

# 957. Prison Cells After N Days [↗ \(/problems/prison-cells-after-n-days/\)](/problems/prison-cells-after-n-days/)

June 10, 2020 | 7K views

Average Rating: 4.25 (16 votes)

There are 8 prison cells in a row, and each cell is either occupied or vacant.

Each day, whether the cell is occupied or vacant changes according to the following rules:

- If a cell has two adjacent neighbors that are both occupied or both vacant, then the cell becomes occupied.
- Otherwise, it becomes vacant.

(Note that because the prison is a row, the first and the last cells in the row can't have two adjacent neighbors.)

We describe the current state of the prison in the following way: `cells[i] == 1` if the `i`-th cell is occupied, else `cells[i] == 0`.

Given the initial state of the prison, return the state of the prison after `N` days (and `N` such changes described above.)

## Example 1:

**Input:** cells = [0,1,0,1,1,0,0,1], N = 7

**Output:** [0,0,1,1,0,0,0,0]

**Explanation:**

The following table summarizes the state of the prison on each day:

Day 0: [0, 1, 0, 1, 1, 0, 0, 1]

Day 1: [0, 1, 1, 0, 0, 0, 0, 0]

Day 2: [0, 0, 0, 0, 1, 1, 1, 0]

Day 3: [0, 1, 1, 0, 0, 1, 0, 0]

Day 4: [0, 0, 0, 0, 0, 1, 0, 0]

Day 5: [0, 1, 1, 1, 0, 1, 0, 0]

Day 6: [0, 0, 1, 0, 1, 1, 0, 0]

Day 7: [0, 0, 1, 1, 0, 0, 0, 0]

### Example 2:

**Input:** cells = [1,0,0,1,0,0,1,0], N = 1000000000

**Output:** [0,0,1,1,1,1,1,0]

### Note:

1. cells.length == 8
2. cells[i] is in {0, 1}
3.  $1 \leq N \leq 10^9$

## Solution

### Overview

First of all, one can consider this problem as a simplified version of the Game of Life

([https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)) invented by the British mathematician John Horton Conway in 1970.

By simplification, this problem is played on one dimensional array (compared to 2D in Game of Life),

and it has less rules.

Due to the nature of game, one of the most intuitive solutions to solve this problem is *playing the game*, i.e. we can simply run the **simulation**.

Starting from the initial state of the prison cells, we could evolve the states following the rules defined in the problem *step by step*.

In the following sections, we will give some approaches on how to run the simulation efficiently.

## Approach 1: Simulation with Fast Forwarding

### Intuition

One important observation from the Game of Life is that we would encounter some already-seen state over the time, simply due to the fact that there are limited number of states.

The above observation applies to our problem here as well. Given  $K$  number of cells, there could be at most  $2^K$  possible states. If the number of steps is larger than all possible states (i.e.  $N > 2^K$ ), we are destined to repeat ourselves sooner or later.

In fact, we would encounter the repetitive states **sooner** than the theoretical boundary we estimated above. For instance, with the initial state of  $[1, 0, 0, 0, 1, 0, 0, 1]$ , just after 15 steps, we would encounter a previously seen state. Once we encounter a state seen before, the history would then repeat itself again and again, assuming that time is infinite.

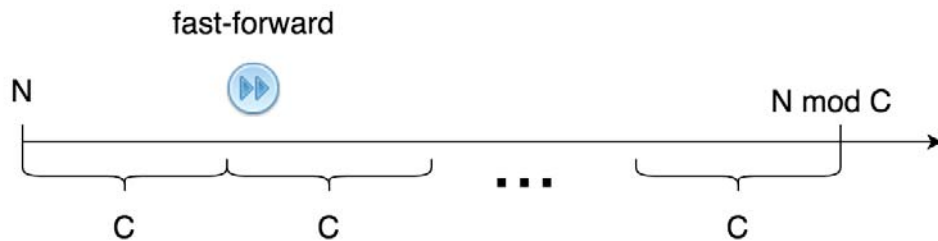
All states between two repetitive states form a cycle, which would repeat itself over the time. Therefore, based on this observation, we could **fast-forward** the simulation rather than going step by step, once we encounter any repetitive state.

### Algorithm

Here is the overall idea to implement our fast-forward strategy.

- First of all, we record the state at each step, with the index of the current step, i.e. `state -> step_index`.
- Once we discover a repetitive state, we can then determine the **length** (denoted as  $C$ ) of the cycle, with the help of hashmap that we recorded.

- Starting from this repetitive state, the prison cells would play out the states within the cycle over and over, until we run out of steps.
- In other words, if the remaining steps is  $N$ , at least we could **fast-forward** to the step of  $N \bmod C$ .
- And then from the step of  $N \bmod C$ , we continue the simulation step by step.




Note: we only need to do the fast-forward once, if there is any.

Here are some sample implementations based on the above idea.

Java

Python3

 Copy

```

1 class Solution:
2     def prisonAfterNDays(self, cells: List[int], N: int) -> List[int]:
3
4         seen = dict()
5         is_fast_forwarded = False
6
7         while N > 0:
8             # we only need to run the fast-forward once at most
9             if not is_fast_forwarded:
10                 state_key = tuple(cells)
11                 if state_key in seen:
12                     # the length of the cycle is seen[state_key] - N
13                     N %= seen[state_key] - N
14                     is_fast_forwarded = True
15                 else:
16                     seen[state_key] = N
17
18             # check if there is still some steps remained,
19             # with or without the fast-forwarding.
20             if N > 0:
21                 N -= 1
22                 next_day_cells = self.nextDay(cells)
23                 cells = next_day_cells
24
25         return cells
26
27
28     def nextDay(self, cells: List[int]):
29         ret = [0] # head
30         for i in range(1, len(cells)-1):
31             ret.append(int(cells[i-1] == cells[i+1]))
32         ret.append(0) # tail
33         return ret

```

## Complexity Analysis

Let  $K$  be the number of cells, and  $N$  be the number of steps.

- Time Complexity:  $\mathcal{O}(K \cdot \min(N, 2^K))$ 
  - As we discussed before, at most we could have  $2^K$  possible states. While we run the simulation with  $N$  steps, we might need to run  $\min(N, 2^K)$  steps without fast-forwarding in the worst case.
  - For each simulation step, it takes  $\mathcal{O}(K)$  time to process and evolve the state of cells.
  - Hence, the overall time complexity of the algorithm is  $\mathcal{O}(K \cdot \min(N, 2^K))$ .
- Space Complexity:

- The main memory consumption of the algorithm is the hashmap that we used to keep track of the states of the cells. The maximal number of entries in the hashmap would be  $2^K$  as we discussed before.
- In the Java implementation, we encode the state as a single integer value. Therefore, its space complexity would be  $\mathcal{O}(2^K)$ , assuming that  $K$  does not exceed 32 so that a state can fit into a single integer number.
- In the Python implementation, we keep the states of cells as they are in the hashmap. As a result, for each entry, it takes  $\mathcal{O}(K)$  space. In total, its space complexity becomes  $\mathcal{O}(K \cdot 2^K)$ .

## Approach 2: Simulation with Bitmap

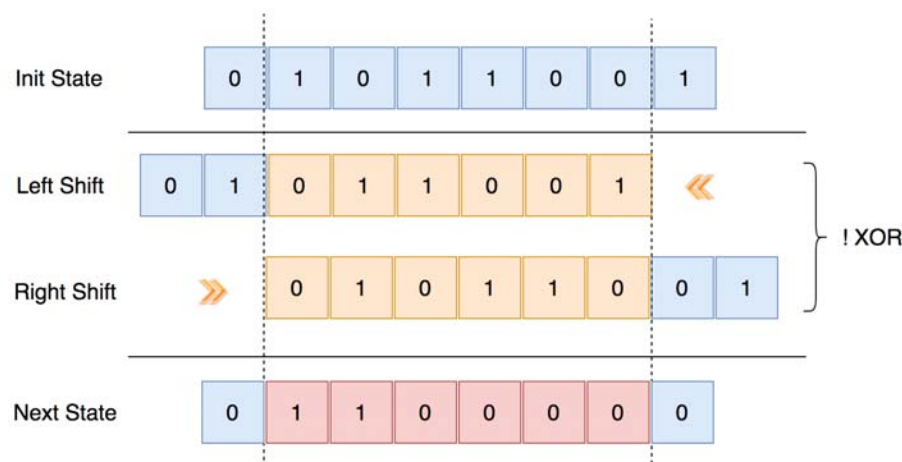
### Intuition

In the above approach, we implemented the function `nextDay(state)`, which runs an **iteration** to calculate the next state of cells, given the current state.

Given that we have already encoded the state as a bitmap in the Java implementation of the previous approach, a more efficient way to implement the `nextDay` function would be to apply the **bit operations** (e.g *AND*, *OR*, *XOR* etc.).

The next state of a cell depends on its left and right neighbors. To align the states of its neighbors, we could make a *left* and a *right* shift respectively on the bitmap. Upon the shifted bitmaps, we then apply the *XOR* and *NOT* operations sequentially, which would lead to the next state of the cell.

Here we show how it works with a concrete example.



Note that, the head and tail cells are particular, which would remain vacant once we start the simulation. Therefore, we should reset the head and tail bits by applying the bit *AND* operation with the **bitmask** of `01111110` (i.e. `0x7e`).


## Algorithm

We could reuse the bulk of the previous implementations, and simply rewrite the `nextDay` function with the bit operations as we discussed.

Additionally, at the end of the simulation, we should *decode* the states of the cells from the final bitmap.

Java

Python3

 Copy

```

1 class Solution:
2     def prisonAfterNDays(self, cells: List[int], N: int) -> List[int]:
3
4         seen = dict()
5         is_fast_forwarded = False
6
7         # step 1). convert the cells to bitmap
8         state_bitmap = 0x0
9         for cell in cells:
10             state_bitmap <<= 1
11             state_bitmap = (state_bitmap | cell)
12
13         # step 2). run the simulation with hashmap
14         while N > 0:
15             if not is_fast_forwarded:
16                 if state_bitmap in seen:
17                     # the length of the cycle is seen[state_key] - N
18                     N %= seen[state_bitmap] - N
19                     is_fast_forwarded = True
20                 else:
21                     seen[state_bitmap] = N
22             # if there is still some steps remained,
23             # with or without the fast-forwarding.
24             if N > 0:
25                 N -= 1
26                 state_bitmap = self.nextDay(state_bitmap)
27
28         # step 3). convert the bitmap back to the state cells
29         ret = []
30         for i in range(len(cells)):
31             ret.append(state_bitmap & 0x1)
32             state_bitmap = state_bitmap >> 1
33
34         return reversed(ret)
35
36
37     def nextDay(self, state_bitmap: int):
38         state_bitmap = ~ (state_bitmap << 1) ^ (state_bitmap >> 1)
39         state_bitmap = state_bitmap & 0x7e # set head and tail to zero
40         return state_bitmap

```

## Complexity Analysis

Let  $K$  be the number of cells, and  $N$  be the number of steps.

- Time Complexity:  $\mathcal{O}(\min(N, 2^K))$  assuming that  $K$  does not exceed 32.
  - As we discussed before, at most we could have  $2^K$  possible states. While we run the simulation, we need to run  $\min(N, 2^K)$  steps without fast-forwarding in the worst case.
  - For each simulation step, it takes a constant  $\mathcal{O}(1)$  time to process and evolve the states of cells, since we applied the bit operations rather than iteration.
  - Hence, the overall time complexity of the algorithm is  $\mathcal{O}(\min(N, 2^K))$ .
- Space Complexity:  $\mathcal{O}(2^K)$ 
  - The main memory consumption of the algorithm is the hashmap that we used to keep track of the states of the cells. The maximal number of entries in the hashmap would be  $2^K$  as we discussed before.
  - This time we adopted the bitmap for both Java and Python implementation, so that each state consumes a constant  $\mathcal{O}(1)$  space.
  - To sum up, the overall space complexity of the algorithm is  $\mathcal{O}(2^K)$ .

Rate this article:

◀ Previous (/articles/sum-of-two-integers/)

Next ▶ (/articles/two-sum-less-than-k/)

Comments: 9

Sort By ▼



Type comment here... (Markdown is supported)

👁 Preview

Post





(/rajatag03)

rajatag03 (/rajatag03) ★ 24 ⌚ 8 hours ago

Why such difficulty solution when it just repeat itself after 14 steps?

The solutions here made me more confused than question itself....

```
class Solution {
    public int[] prisonAfterNDays(int[] cells, int N) {

        if(N>14){
            N=(N%14);
        }
        if(N==0){
            N=14;
        }
        for(int k=0;k<N;k++){
            {
                int[] temp=new int[cells.length];
                for(int i=1;i<cells.length-1;i++){
                    if((cells[i-1]==0 && cells[i+1]==0)|| (cells[i-1]==1 && cells[i+1]==1)){
                        temp[i]=1;
                    }
                    else{
                        temp[i]=0;
                    }
                }

                cells=temp;
            }

            return cells;
        }
    }
}
```

1 ^ v | 📄 Share | ↩ Reply

SHOW 5 REPLIES



(/cathy0517)

cathy0517 (/cathy0517) ★ 12 ⌚ June 13, 2020 4:35 AM

what is the meaning of '0x7e'?

0 ^ v | 📄 Share | ↩ Reply

SHOW 4 REPLIES



(/zzznotsomuch)

zzznotsomuch (/zzznotsomuch) ★ 34 36 minutes ago

I will be happy to fail the interview where this question is asked and the candidate is expected to come up with all of this logic in 45 mins! I had a gut feeling that there should be a pattern but it wasn't intuitive that the pattern will repeat. Well, today I learned! Good puzzle question but a bad interview question in my opinion.

0 ^ v | Share | Reply



(/meowlicious99)

meowlicious99 (/meowlicious99) ★ 470 4 hours ago

log(n) for people who aren't able to figure out mod 14 or XOR tricks

```
def prisonAfterNDays(self, input: List[int], N: int) -> List[int]:
    @lru_cache(None)
```

[Read More](#)

0 ^ v | Share | Reply



(/indrajohn)

indrajohn (/indrajohn) ★ 19 6 hours ago

Without the trick of div = 14, its leading to TLE for  $N = 10^9$ , for a normal memoized caching solution.

Solution shall not be restricted that way.

0 ^ v | Share | Reply

[SHOW 1 REPLY](#)

(/hermanurikh)

hermanurikh (/hermanurikh) ★ 20 8 hours ago

I wonder how many candidates would invent the XOR approach in 45 mins. So elegant!

0 ^ v | Share | Reply

[SHOW 1 REPLY](#)

(/leetcode\_master)

LeetCode\_Master (/leetcode\_master) ★ 112 8 hours ago

```
class Solution
{
    public int[] prisonAfterNDays(int[] cells, int N)
    {
```

[Read More](#)

0 ^ v | Share | Reply



(/akshaynathr)

akshaynathr (/akshaynathr) ★6 ⓘ 2 days ago

Why is the first and last cells made to 0 in the nextDay() function? (Approach 1):

```
protected int[] nextDay(int[] cells) {  
    int[] newCells = new int[cells.length];  
    newCells[0] = 0;
```

[Read More](#)0 ^ v | [Share](#) | [Reply](#)[SHOW 1 REPLY](#)

(/tpt5cu)

tpt5cu (/tpt5cu) ★110 ⓘ June 11, 2020 12:43 AM

why is the second if N&gt;0 required in the first solution? Doesn't the for loop already check for n?

0 ^ v | [Share](#) | [Reply](#)[SHOW 1 REPLY](#)