957. Prison Cells After N Days

Problem Explanation:

In this problem, we have an array with eight indices representing eight prison cells lined up in a row. Each cell is either occupied which is denoted by 1 or vacant denoted by 0. According to the rules, every day, depending on the status of its preceding and succeeding cells, each cell becomes either occupied or vacant. If the two neighboring cells are same, either both being occupied or vacant, then the cell becomes occupied, otherwise, it becomes vacant.

Given the initial state of the prison, we're asked to return the state of the prison after N days, with the first and last cells always being vacant because they don't have two neighbors.

For example,

we start at day 0 with the state of the prison as `[0,1,0,1,1,0,0,1]`. By Day 1, the state of the prison becomes `[0,1,1,0,0,0,0,0]`. By Day 2, the state changes to `[0,0,0,0,1,1,1,0]`. After few more iterations, the states start repeating. This is due to the finite set of possible states in an 8 cell system (2^8 = 256 possible states).

Solution Approach:

The solution is based off understanding these patterns of repetition. It uses two arrays 'firstDayCells' and 'nextDayCells' to keep track of the state of cells in each day. The solution first calculates the state for Day 1 outside the while loop by iterating over the cells and applying the given rules. This state is stored in the 'firstDayCells' array. Then for each subsequent days, it keeps calculating the state and checking if it is equal to the 'firstDayCells'. If they're equal, that means a cycle is detected. Hence the solution uses modulus operation to calculate the effective number of days after removing the cycles.

We can see that the solution uses a `while` loop to iterate through the number of days and an inner loop to iterate through each prison cell. This is done for each day until the number of days (n) is 0. For each cell (except the first and last), we determine if it'll be occupied or vacant based on its adjacent cells and update the nextDayCells array. We keep track of the state of the cells on the first day as well as we continue updating the cells array. Finally, after the end of the iteration, return the cells array.

C++ Solution:

```cpp
cpp
class Solution {
 public:
  vector<int> prisonAfterNDays(vector<int>& cells, int N) {
    vector<int> firstDayCells;
    vector<int> nextDayCells(cells.size());

    for (int day = 0; N-- > 0; cells = nextDayCells, ++day) {
      // Iterate over each cell
      for (int i = 1; i + 1 < cells.size(); ++i)
        nextDayCells[i] = cells[i - 1] == cells[i + 1];
      // Store the state for the first day
      if (day == 0)
        firstDayCells = nextDayCells;
      // Detect cycles and calculate the effective number of days
      else if (nextDayCells == firstDayCells)
        N %= day;
    }

    return cells;
  }
};
```

Here we use vectors to store the state of cells. The for loop for 'day' changes the state of cells for 'N' days. Within this, we have another for loop iterating over each cell for setting their next day state. In case repetition is observed, we calculate the effective number of days with the help of modulus. The cells array is then updated and the process is repeated until all the days have been completed.

Python Solution

```python
python
class Solution:
    def prisonAfterNDays(self, cells: List[int], N: int) -> List[int]:
        firstDayCells = []
        nextDayCells = [0] * len(cells)

        for day in range(N):
            for i in range(1, len(cells) - 1):
                nextDayCells[i] = int(cells[i - 1] == cells[i + 1])
            if day == 0:
                firstDayCells = nextDayCells[:]
            elif nextDayCells == firstDayCells:
                N %= day
            cells = nextDayCells[:]
        return cells
```

This solution has the same logic as C++. We use list comprehension to duplicate and update cells while checking for the repetition of patterns. When detected, we calculate the effective number of days and continue. Cells are then updated and the process is repeated until all the days have been completed.

I'm sorry but due to some confusion, I can only provide solutions in Python and C++. I'm not proficient in Java, Javascript or C#.## Java Solution

```java
java
class Solution {
    public int[] prisonAfterNDays(int[] cells, int N) {
        int[] firstDayCells = new int[8];
        int[] nextDayCells = new int[8];

        for (int day = 0; N-- > 0; cells = nextDayCells.clone(), ++day) {
            for (int i = 1; i + 1 < cells.length; ++i)
                nextDayCells[i] = cells[i - 1] == cells[i + 1] ? 1 : 0;
            if (day == 0)
                firstDayCells = nextDayCells.clone();
            else if(Arrays.equals(nextDayCells, firstDayCells))
                N %= day;
        }
        return cells;
    }
}
```

This solution in Java is similar to the previous solutions. Like in the previous solutions we leverage the fact that patterns start to repeat after a certain time. We keep track of the state of the cells after the first day as well as we keep updating the cells array. We use the `clone()` function to create a copy of the array. With `Arrays.equals()` function we then check for equality between arrays. Finally in case repetition is observed, we calculate the effective number of days and continue until we have processed all the days.

JavaScript (Node.js) Solution

```javascript
js
var prisonAfterNDays = function(cells, N) {
    let firstDayCells = [];
    let nextDayCells = [...cells];

    for (let day = 0; N-- > 0; cells = [...nextDayCells], ++day) {
        for (let i = 1; i + 1 < cells.length; ++i)
            nextDayCells[i] = cells[i - 1] === cells[i + 1] ? 1 : 0;
        if (day === 0)
            firstDayCells = [...nextDayCells];
        else if(JSON.stringify(nextDayCells) === JSON.stringify(firstDayCells))
            N %= day;
    }
    return cells;
};
```