672. Bulb Switcher II Medium Bit Manipulation **Depth-First Search Breadth-First Search** Math **Leetcode Link** 

# There is a room equipped with n light bulbs, each with a distinct label ranging from 1 to n. Initially, all bulbs are in the 'on' state.

**Problem Description** 

Within this room, there is a wall-mounted control panel that includes four distinct buttons, each with a unique function that affects the state of one or more bulbs: Button 1: Toggles the state (on/off) of all the bulbs.

- Button 2: Toggles the state of all bulbs with an even-numbered label.
- Button 3: Toggles the state of all bulbs with an odd-numbered label.
- . . . .
- Button 4: Toggles the state of bulbs labeled with numbers of the form j=3k+1 (for k=0, 1, 2, ...), which includes 1, 4, 7, 10, The task is to make exactly presses number of button presses, and you can choose to press any button at each step. The question
- For example, with n bulbs and presses allowed presses, what are the different ways the bulbs can be lit or unlit? Intuition

requires us to calculate the total number of distinct lighting configurations that can be achieved after making exactly presses number

To solve this problem, we notice that toggling the lights in certain patterns can lead to the same outcome. This realization can significantly reduce the number of possible configurations we need to check.

## Here's the intuitive approach to find the solution:

of button presses.

1. Optimization of n: Since each press of a button flips the status of the bulbs in a certain pattern, and the patterns repeat every 6 bulbs, we can reduce any n greater than 6 to n % 6, which simplifies the problem.

2. Unique States Analysis: We note that after a certain number of presses, the patterns begin to repeat, and thus the number of

3. Bitmask Technique: Implementing the idea of using a bitmask to represent the state of the light bulbs and the actions of the buttons, where 1 represents a bulb turned on and 0 a bulb turned off, we use bitwise operations to simulate the pressing of buttons and record the resulting state of the bulbs.

unique states is finite. By analyzing the effects of each button press, we can establish that there aren't many different states the

bulbs can be in after presses button presses. This is due to the symmetries and overlaps in the patterns created by each button.

- 4. Iterating Over Possibilities: By iterating over all possible combinations of button presses (which can be up to 2^4 because there are four buttons), we can check which combinations of button presses will result in a unique final state of the bulbs. At each iteration, we apply the respective toggles based on the pressed buttons and store the outcome in a set to avoid duplicates.
- 5. Parity Check: We can further trim down the possibilities by recognizing that the number of presses must have the same parity (even or odd) as the number of operations to reach a specific state. This comes from the insight that toggling an even number of times will lead to the original state, while an odd number will result in a change.

6. Counting Unique States: After generating all possible states, the size of the set will be the number of unique states since a set

states. Solution Approach

The implementation of the solution involves a clever use of bit manipulation and set data structures to efficiently determine the

Implementing these insights into code, we simulate the button presses, record the possible states, and return the count of distinct

The overall approach can be outlined as follows: 1. Minimizing n: The first line of action in our approach is to minimize the value of n (the number of bulbs) to 6 or below, as the

patterns repeat every 6 bulbs. This is achieved with  $n = \min(n, 6)$ . Beyond 6 bulbs, the states just repeat and do not introduce

2. **Defining Operations**: An array (or tuple) ops is defined, representing the outcome of each button press as a bit mask. This is a

## Button 1 toggles all bulbs, hence 0b111111.

button presses.

Example Walkthrough

number 1) to left, is 111.

(flips the last bit).

5. **Applying Toggles**:

any new combinations.

does not contain duplicates.

unique states of the light bulbs after a certain number of presses.

compact way to encode which bulbs are affected by each button:

XOR operation  $t \sim p$  to toggle the state of the bulbs affected by that button.

represents the state of the light bulbs after applying the combination of button presses.

accounting for the symmetry in the buttons' operation to tally the unique final states of the light bulbs.

 Button 2 toggles bulbs at even positions, hence 0b010101. Button 3 toggles bulbs with odd labels, hence 0b101010. Button 4 toggles bulbs with labels 1, 4, 7, ..., hence 0b100100. 3. Iterating Through Button Combinations: We iterate through all possible combinations of button presses using a loop for mask

in range(1 << 4) which generates numbers from 0 to 15 (in binary: 0000 to 1111), each representing a unique combination of

mask.bit\_count() and check whether the count of presses is less than or equal to the allowed number of presses and has the same parity. This eliminates unnecessary checks and focuses our attention on valid scenarios. 5. Applying Toggles: If the combination is valid, we apply the toggles associated with each bit in the mask. The loop for i, op in

6. Recording the State: To ensure that we're only considering the bulbs we have (n may be less than 6), we perform a bit mask t

&= (1 << 6) - 1 of the first six positions and then shift the result to fit our number of bulbs using t >>= 6 - n. This final value

7. Avoiding Duplicate States: We use a set vis to store unique states, preventing duplicate counts. We add the resulting state t to

enumerate(ops) goes through the operations, and if bit i in the mask is set (i.e., the corresponding button is pressed), we use an

4. Parity and Presses Check: For each combination, we count the number of bits (i.e., button presses) using cnt =

this set vis.add(t). 8. Getting the Count: After iterating through all combinations and recording the valid states, the length of the set len(vis) gives us the number of unique possible statuses of the bulbs, which is the solution to the problem.

By using bit masks and sets, the code efficiently simulates all possible scenarios of button presses, while avoiding redundancies and

Let's walk through a small example to illustrate the solution approach using the above method with n = 3 bulbs and presses = 1 allowed press. In this scenario, we are only interested in the toggles that can happen in one press or less, so we will examine what each button does when pressed alone.

The initial state of the bulbs is all "on", which in bit representation (1 for 'on' and 0 for 'off'), starting from the rightmost bit (bulb

2. **Defining Operations**: We define the bitmasks for the given operations, considering only the first 3 bits for our n = 3 bulbs:

 Button 2 toggles even bulbs, bitmask: 010 (flips the middle bit). Button 3 toggles odd bulbs, bitmask: 101 (flips the first and last bits). Button 4 toggles bulbs with labels 1, 4, 7, ..., but since we only have 3 bulbs, it's equivalent to toggling bulb 1, bitmask: 001

3. Iterating Through Button Combinations: We iterate through the combinations of pressing only one button, as we are allowed

4. Parity and Presses Check: Since we are allowed only 1 press, we only consider combinations where the button bit count is 1 or

only one press. We have 0000 through 1000 (in binary), but since the first bit will always be 0, we care only about 0001, 0010, 0100,

 For combination 0001 (Button 1): Initial 111 toggled becomes 000 (all off). • For combination 0010 (Button 2): Initial 111 toggled becomes 101 (middle bulb off).

0. Thus, all combinations are valid because either no button is pressed or exactly one is pressed.

1. Minimizing n: Since n is already less than or equal to 6, we do not need to modify n.

Button 1 toggles all bulbs, bitmask: 111 (flips all bits).

and 1000 representing button presses 1 through 4 respectively.

7. Avoiding Duplicate States: We add each state to a set:

possible with n = 3 bulbs and 1 allowed press.

def flipLights(self, n: int, presses: int) -> int:

# Use set to avoid duplicate configurations

if (mask >> i) & 1:

temp &= (1 << 6) - 1

public int flipLights(int n, int presses) {

int tempPattern = 0;

for (int i = 0; i < 4; ++i) {

tempPattern &= ((1 << 6) - 1);

visitedPatterns.add(tempPattern);

tempPattern >>= (6 - n);

return visitedPatterns.size();

return visitedConfigurations.size();

1 // Limit the number of lights as the pattern repeats every 6 lights.

const operations: number[] = [0b111111, 0b010101, 0b101010, 0b100100];

12 // Flip all the lights based on the number of presses and return different configurations.

// Apply a mask to isolate only the bits corresponding to 'n' lights

// Count the number of bits set (1s) in the binary representation of the number.

// Define the operations corresponding to flipping lights

function flipLights(n: number, presses: number): number {

// A set to keep track of unique configurations.

10 const visitedConfigurations: Set<number> = new Set();

for (let mask = 0; mask < 1 << 4; ++mask) {</pre>

function countSetBits(number: number): number {

let count: number = countSetBits(mask);

configuration ^= operations[i];

if (((mask >> i) & 1) == 1) {

// 0b010101 -> Flip lights with odd indices

// Ob101010 -> Flip lights with even indices

Set<Integer> visitedPatterns = new HashSet<>();

// 0b100100 -> Flip lights at 3k+1 positions (1, 4, ...)

if (count <= presses && count % 2 == presses % 2) {</pre>

tempPattern ^= operations[i];

// Return the number of distinct patterns recorded in the set

// Initialize the temporary light pattern to zero

// Ob111111 -> Flip all lights

temp >>= 6 - n

temp ^= operation

operations = (0b111111, 0b010101, 0b101010, 0b100100)

if count <= presses and count % 2 == presses % 2:</pre>

for i, operation in enumerate(operations):

After Button 1: Add 000.

After Button 4: Add 110.

n = min(n, 6)

visited = set()

for mask in range(1 << 4):</pre>

1 class Solution:

9

10

11

12

13

14

15

16

17

18

19

20

21

23

24

25

26

27

28

2

8

9

10

11

12

13

14

15

16

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

53

52 }

 For combination 0100 (Button 3): Initial 111 toggled becomes 010 (only the middle bulb on). For combination 1000 (Button 4): Initial 111 toggled becomes 110 (last bulb off).

6. Recording the State: Since n=3, no further bit masking or shifting is needed. We record these states as they are.

• After Button 2: Add 101. After Button 3: Add 010.

# Define the operations bitmask for the 4 different types of buttons

temp = 0 # This will store the combination of flips

# Trim the pattern to the least significant 'n' bits

# Since the light pattern repeats every 6 lights, limit n to 6

Thus, the answer for n = 3 bulbs and presses = 1 press is 4 unique lighting configurations. Python Solution

8. Getting the Count: We count the number of unique statuses in our set: {000, 101, 010, 110}. There are 4 unique states

# Iterate over all combinations of button presses (by checking all bitmasks up to the 4th bit)

# Only consider valid sequences of presses: those that match the number of presses and parity

count = bin(mask).count('1') # calculate the number of pressed buttons

# Apply the operation if the corresponding button was pressed

visited.add(temp) # Add this pattern to the visited configurations

// Method to determine the number of different light configurations possible after pressing switches

// Light switch operations, represented in a 6-bit pattern (max number of lights we are considering)

// If this count of operations can be performed within given presses, and the parity (even/odd) matches

// For each operation, check if it is included in the current combination, and if so, apply XOR

return len(visited) # The number of unique configurations is the size of the set

Java Solution

int[] operations = new int[] {0b111111, 0b010101, 0b101010, 0b100100};

// A set to record distinct light patterns after performing operations

// Since the pattern repeats every 6 lights, we only consider the first 6 lights

// Mask off irrelevant bits (we only consider the first 6 lights)

// Add the resulting pattern to the set of visited patterns

// Shift the bits for patterns with lights less than 6, if necessary

#### 17 n = Math.min(n, 6);18 19 // Iterate over all 16 possible combinations of the 4 operations using a bitmask 20 for (int mask = 0; mask < 1 << 4; ++mask) {</pre> 21 22 // Count the number of operations to be performed (number of set bits in mask) int count = Integer.bitCount(mask);

1 class Solution {

```
C++ Solution
 1 class Solution {
 2 public:
        // The flipLights function determines the number of different lighting configurations
        // that can be achieved by operating the switches a given number of times.
        // We limit the lights and operations to minimize calculations due to symmetries.
        int flipLights(int n, int presses) {
 6
            // Limit the number of lights to 6 because the pattern repeats every 6 lights.
            n = min(n, 6);
 8
 9
10
            // Define the operations corresponding to flipping lights:
            // 1. Flip all the lights (0b111111)
11
12
            // 2. Flip lights at even positions (0b010101)
13
            // 3. Flip lights at odd positions (0b101010)
14
            // 4. Flip lights at positions 1, 4, ... (0b100100)
            vector<int> operations = {0b111111, 0b010101, 0b101010, 0b100100};
15
16
17
            // Use a set to keep track of unique configurations after certain presses.
18
            unordered_set<int> visitedConfigurations;
19
20
            // Iterate over all possible combinations of operations.
            // (1 << 4) creates a bitmask with 4 bits, iterating over all combinations of 4 switches.
21
            for (int mask = 0; mask < 1 << 4; ++mask) {</pre>
22
23
                // Count the number of bits set in the mask.
24
                int count = __builtin_popcount(mask);
25
26
                // Skip combinations where the number of operations doesn't match the presses.
27
                // The count of operations must be the same as the number of presses modulo 2.
                if (count > presses || count % 2 != presses % 2) continue;
29
30
                // Initialize the configuration for this combination of operations.
31
                int configuration = 0;
                for (int i = 0; i < 4; ++i) {
32
33
                    // If the i-th operation is to be applied, XOR it with the current configuration.
                    if (mask >> i & 1) {
34
35
                        configuration ^= operations[i];
36
37
38
39
                // Apply a mask to isolate only the bits corresponding to 'n' lights.
                configuration &= (1 << 6) - 1;
40
                // Right-shift the bits to get the correct configuration based on the number of lights.
41
                configuration >>= (6 - n);
42
43
44
                // Insert the final configuration into the set.
45
                visitedConfigurations.insert(configuration);
46
47
48
            // The number of unique configurations is the size of the set.
```

#### 30 configuration &= (1 << n) - 1; 31 32 visitedConfigurations.add(configuration); 33 34 35 // The number of unique configurations is the size of the set. 36 return visitedConfigurations.size;

### The main portion of the code is iterating through all possible combinations of the four light operations, which is done by looping from 0 to 1 << 4 (or 16). This is a constant, as it does not depend on the input variables n or presses. The .bit\_count() function on the mask variable counts the number of set bits in mask's binary representation, and this is done in

Finally, for each valid mask, the code performs a reduction of the state space by only considering the first 6 lights and then shifting to consider the actual number (n) of lights. This operation involves constant-time bitwise manipulation as well.

**Space Complexity** The space complexity is driven by the vis set, which collects the unique states of the lights. In the worst-case scenario, every

constant time, O(1), given modern computer architectures which support this operation natively.

The comparison operations and the bitwise operations inside the loop also take constant time.

possible state of the lights is unique. Because the state space is reduced to min(n, 6), there are at most 2^6 (or 64) unique states. The vis set will, therefore, contain at most 64 elements, regardless of the actual number of lights or presses. Hence, the space

19 // Skip combinations where the number of operations doesn't match the presses 20 if (count > presses || count % 2 !== presses % 2) continue; 21 22 let configuration: number = 0; for (let i = 0; i < 4; ++i) { 23 24 if ((mask >> i) & 1) {

49

50

51

52

4

11

14

15

16

17

18

25

26

27

28

29

37 }

38

41

42

43

44

45

46

48

47 }

};

Typescript Solution

function minLights(n: number): number {

return Math.min(n, 6);

n = minLights(n);

Time and Space Complexity

Time Complexity

return count;

let count: number = 0;

count += number & 1;

number = number >> 1;

while (number) {

Taking all of the above into account, the overall time complexity of the function is O(1), since it is bounded by a constant number of

complexity is also O(1), because the amount of space required does not grow with the input size.

operations that do not change with the size of the input.