<https://leetcode.com/problems/non-negative-integers-without-consecutive-ones/description/>

Given a positive integer n, return the number of the integers in the range [0, n] whose binary representations **do not** contain consecutive ones.

**Example 1:**

Input: n = 5

Output: 5

Explanation:

Here are the non-negative integers <= 5 with their corresponding binary representations:

0 : 0

1 : 1

2 : 10

3 : 11

4 : 100

5 : 101

Among them, only integer 3 disobeys the rule (two consecutive ones) and the other 5 satisfy the rule.

**Example 2:**

Input: n = 1

Output: 2

**Example 3:**

Input: n = 2

Output: 3

**Constraints:**

* 1 <= n <= 10^9

**Attempt 1: 2023-11-10**

**Solution 1: Bitwise (30 min, TLE 27/527)**

class Solution {

public int findIntegers(int n) {

int count = 0;

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

if(check(i)) {

count++;

}

}

return count;

}

/\*\*

Refer

https://coderanch.com/t/540847/java/int-int

To really understand why an int goes from 2147483647 to -2147483648

when you add 1, you have to understand how integers work behind the scenes.

In Java (as in many other programming languages), integers are stored

in two's complement format. The type int is 32 bits in Java.

If you look at the bit pattern:

2147483647 = 01111111 11111111 11111111 11111111

If you add 1 to that, you get: 10000000 00000000 00000000 00000000

In two's complement, that is the number -2147483648.

\*/

// Why i increase from 0 till 30 (i < 31) in for loop ?

// E.g check all 1's for max integer 2147483647 with bit operation

// 2^31 - 1 = 2147483647 -> 01111111111111111111111111111111

// (32 bits = 31 bits of 1 and signal bit is 0)

// Let's shift it to right bit by bit with logical right shift

// Logical right shift, does not care that the value could possibly

// represent a signed number, it simply moves everything to the right

// and fills in from the left with 0s

// for(int i = 0; i < 31; i++) {

// 2147483647 >>> 1;

// }

// i = 0: 2147483647 >>> 1 => 1073741823

// 01111111111111111111111111111111 => 00111111111111111111111111111111

// i = 1: 1073741823 >>> 1 => 536870911

// 00111111111111111111111111111111 => 00011111111111111111111111111111

// ...

// ... after 31 times right shift (i increase from 0 till 30) all 1 removed

// ...

// i = 29: 7 >>> 1 => 3

// 00000000000000000000000000000111 => 00000000000000000000000000000011

// i = 30: 3 >>> 1 => 1

// 00000000000000000000000000000011 => 00000000000000000000000000000001

// Which means for max integer 2147483647 after 31 times logic right shift

// we able to filter out all binary bit 1's digit by digit

private boolean check(int num) {

for(int i = 0; i < 31; i++) {

boolean a = (num >>> i & 1) == 1;

boolean b = ((num >>> (i + 1)) & 1) == 1;

if(a && b) {

return false;

}

}

return true;

}

}

Time complexity : O(32∗n). We test the 32 consecutive positions of every number from 0 to n. Here, n refers to given number.

Space complexity : O(1). Constant space is used.

**Refer to**

<https://leetcode.com/problems/non-negative-integers-without-consecutive-ones/>

#### **Approach #1 Brute Force [Time Limit Exceeded]**

The brute force approach is simple. We can traverse through all the numbers from 1 to num. For every current number chosen, we can check all the consecutive positions in this number to check if the number contains two consecutive ones or not. If not, we increment the count of the resultant numbers with no consecutive ones.

To check if a 1 exists at the position x (counting from the LSB side), in the current number n, we can proceed as follows. We can shift a binary 1 x−1 times towards the left to get a number y which has a 111only at the xth position. Now, logical ANDing of n and y will result in a logical 111output only if n contains 1 at the xth position.

public class Solution {

public int findIntegers(int num) {

int count = 0;

for (int i = 0; i <= num; i++)

if (check(i))

count++;

return count;

}

public boolean check(int n) {

int i = 31;

while (i > 0) {

if ((n & (1 << i)) != 0 && (n & (1 << (i - 1))) != 0)

return false;

i--;

}

return true;

}

}

**Complexity Analysis**

* Time complexity : O(32∗n). We test the 32 consecutive positions of every number from 0 to n. Here, n refers to given number.
* Space complexity : O(1). Constant space is used.

**Solution 2: Native DFS (30 min, TLE 527/527)**

class Solution {

public int findIntegers(int n) {

return helper(0, 0, n, false);

}

// One critical setup: why use boolean for 'prevBitIsOne' ?

// Because if we use int to identify the previous bit is 1 or 0, that

// will cause a problem for the LSB (least significant bit), because

// the initial bit will be 0, since we start with number = 0, which

// the LSB is 0, but if we define int prevBitIsOne = 0, then how we

// present number = 1, which requires the LSB = 1 ? So the conflict

// happened, to enable the ability to assign LSB as either 0 or 1, we

// cannot use int to represent 'prevBitIsOne', the alternative way

// is use boolean to enable the ability to assign LSB as either 0 or 1

private int helper(int countOfBits, int curSum, int n, boolean prevBitIsOne) {

// Base condition

if(curSum > n) {

return 0;

}

// Base condition but difficult to get:

// Only when 1 shift count of bits and larger than n, we recognize

// it as one number, e.g 1 <<< 3 = 8 > 5, then all 3 digits potential

// combination [000 ~ 111] which under 8(=1000)'s cover and able to

// go till this condition is a qualified number

if(1 << countOfBits > n) {

return 1;

}

// If the previous bit is 1 we only have one choice to append 0 on left

// e.g previous bit = '1', then append '0' ahead -> '01'

if(prevBitIsOne) {

return helper(countOfBits + 1, curSum, n, false);

}

// If the previous bit is 0 we can append either 0 or 1

// e.g previous bit = '0', then append '0' ahead -> '00', apeend '1' ahead -> '10'

return helper(countOfBits + 1, curSum, n, false) + helper(countOfBits + 1, curSum + (1 << countOfBits), n, true);

}

}

Time complexity : O(x). Only x numbers are generated. Here, x refers to the resultant count to be returned.

Space complexity : O(log(max\_int)=32). The depth of recursion tree can go up to 32.

**Refer to**

<https://leetcode.com/problems/non-negative-integers-without-consecutive-ones/editorial/>

#### **Approach #2 Better Brute Force [Time Limit Exceeded]**

**Algorithm**

In the last approach, we generated every number and then checked if it contains consecutive ones at any position or not. Instead of this, we can generate only the required kind of numbers. e.g. If we generate numbers in the order of the number of bits in the current number, if we get a binary number 110 on the way at the step of 3-bit number generation. Now, since this number already contains two consecutive ones, it is useless to generate number with more number of bits with the current bitstream as the suffix(e.g. numbers of the form 1110 and 0110).

The current approach is based on the above idea. We can start with the LSB position, by placing a 0and a 1at the LSB. These two initial numbers correspond to the 1-bit numbers which don't contain any consecutive ones. Now, taking 0 as the initial suffix, if we want to generate two bit numbers with no two consecutive 1's, we can append a 1 and a 0 both in front of the initial 0generating the numbers 10 and 00as the two bit numbers ending with a 0with no two consecutive 1's.

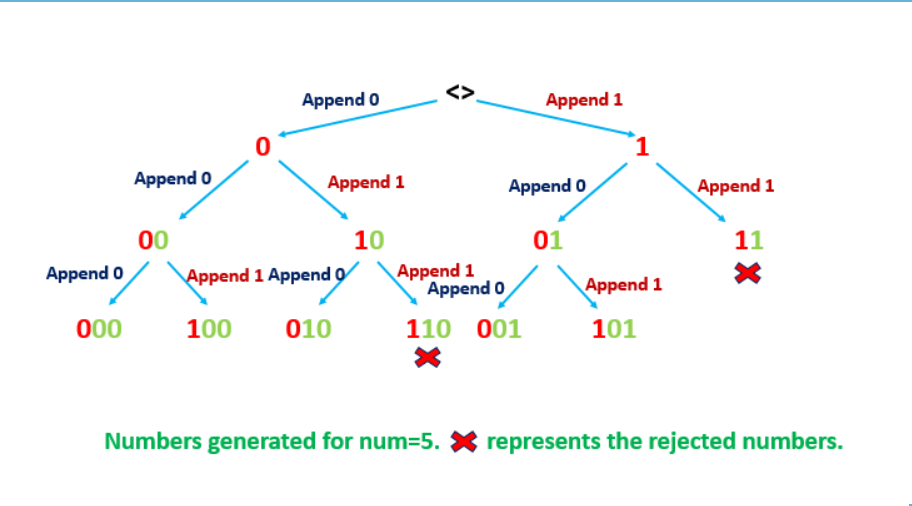
But, when we take 1 as the initial suffix, we can append a 0 to it to generate 01 which doesn't contain any consecutive ones. But, adding a 1 won't satisfy this criteria(11 will be generated). Thus, while generating the current number, we need to keep a track of the point that whether a 1was added as the last prefix or not. If yes, we can't append a new 1 and only 0 can be appended. If a 0 was appended as the last prefix, both 0and 1can be appended in the new bit-pattern without creating a violating number.

Thus, we can continue forward with the 3-bit number generation only with 00, 01 and 10 as the new suffixes in the same manner.

To get a count of numbers lesser than num, with no two consecutive 1's, based on the above discussion, we make use of a recursive function find(i, sum, num, prev). This function returns the count of binary numbers with i bits with no two consecutive 1's. Here, sum refers to the binary number generated till now(the prefix obtained as the input). num refers to the given number. prev is a boolean variable that indicates whether the last prefix added was a 1or a 0.

If the last prefix was a 0, we can add both 1and 0as the new prefix. Thus, we need to make a function call find(i + 1, sum, num, false) + find(i + 1, sum + (1 << i), num, true). Here, the first sub-part refers to a 0being added at the ith position. Thus, we pass a false as the prefix in this case. The second sub-part refers to a 1 being added at the ith position. Thus, we pass true as the prefix in this case. If the last prefix was a 1, we can add only a 0as the new prefix. Thus, only one function call find(i + 1, sum, num, false)is made in this case.

Further, we need to stop the number generation whenever the current input number(sum) exceeds the given number num



public class Solution {

public int findIntegers(int num) {

return find(0, 0, num, false);

}

public int find(int i, int sum, int num, boolean prev) {

if (sum > num)

return 0;

if (1<<i > num)

return 1;

if (prev)

return find(i + 1, sum, num, false);

return find(i + 1, sum, num, false) + find(i + 1, sum + (1 << i), num, true);

}

}

**Complexity Analysis**

* Time complexity : O(x). Only x numbers are generated. Here, x refers to the resultant count to be returned.
* Space complexity : O(log(max\_int)=32). The depth of recursion tree can go up to 32.

**Wrong Solution: Traditional DFS + Memoization (Pass 419/527)**

class Solution {

public int findIntegers(int n) {

// countOfBits -> max = 32 bits

// curSum vs n relation -> boolean

// prevBitIsOne -> boolean

Integer[][][] memo = new Integer[32][2][2];

return helper(0, 0, n, false, memo);

}

// One critical setup: why use boolean for 'prevBitIsOne' ?

// Because if we use int to identify the previous bit is 1 or 0, that

// will cause a problem for the LSB (least significant bit), because

// the initial bit will be 0, since we start with number = 0, which

// the LSB is 0, but if we define int prevBitIsOne = 0, then how we

// present number = 1, which requires the LSB = 1 ? So the conflict

// happened, to enable the ability to assign LSB as either 0 or 1, we

// cannot use int to represent 'prevBitIsOne', the alternative way

// is use boolean to enable the ability to assign LSB as either 0 or 1

private int helper(int countOfBits, int curSum, int n, boolean prevBitIsOne, Integer[][][] memo) {

// Base condition

if(curSum > n) {

return 0;

}

// Base condition but difficult to get:

// Only when 1 shift count of bits and larger than n, we recognize

// it as one number, e.g 1 <<< 3 = 8 > 5, then all 3 digits potential

// combination [000 ~ 111] which under 8(=1000)'s cover and able to

// go till this condition is a qualified number

if(1 << countOfBits > n) {

return 1;

}

int tmp1 = curSum > n ? 1 : 0;

int tmp2 = prevBitIsOne ? 1 : 0;

if(memo[countOfBits][tmp1][tmp2] != null) {

return memo[countOfBits][tmp1][tmp2];

}

// If the previous bit is 1 we only have one choice to append 0 on left

// e.g previous bit = '1', then append '0' ahead -> '01'

if(prevBitIsOne) {

return memo[countOfBits][tmp1][tmp2] = helper(countOfBits + 1, curSum, n,false, memo);

}

// If the previous bit is 0 we can append either 0 or 1

// e.g previous bit = '0', then append '0' ahead -> '00', append '1' ahead -> '10'

return memo[countOfBits][tmp1][tmp2] = (helper(countOfBits + 1, curSum, n, false, memo) + helper(countOfBits + 1, curSum + (1 << countOfBits), n, true, memo));

}

}

**Failed on n = 4, expected 4, actually return 5**

**Analysis:**

1. Currently we have 3 dimensions and the definition, assignment and usage is below:

// countOfBits -> max = 32 bits

// curSum vs n relation -> boolean

// prevBitIsOne -> boolean

int tmp1 = curSum > n ? 1 : 0;

int tmp2 = prevBitIsOne ? 1 : 0;

if(memo[countOfBits][tmp1][tmp2] != null) {

return memo[countOfBits][tmp1][tmp2];

}

The current 2nd dimension memo[countOfBits][tmp1 = curSum > n ? 1 : 0][tmp2] -> boolean for curSum > n always = 0, no change and not create any difference and may because this 2nd dimension not create actual difference to identify

the status which result in error out on 2nd point

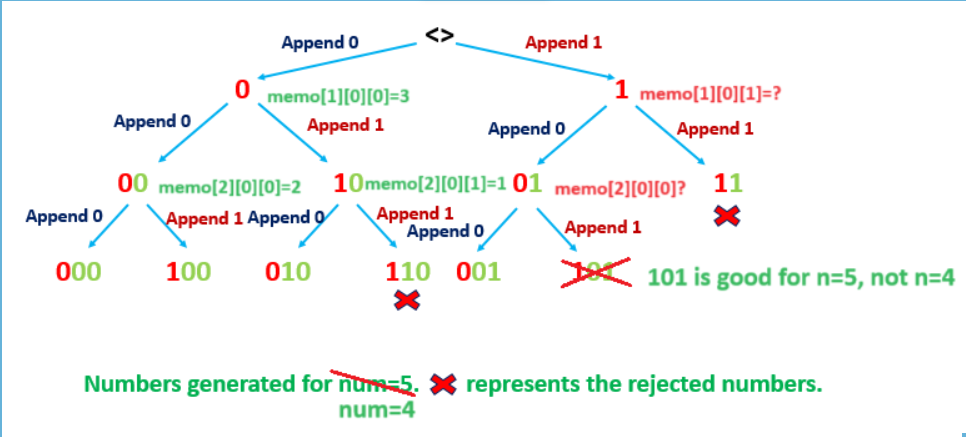
2. After recursion on left subtree which append based on 0, we will obtain memo[2][0][0]=2, but when recursion happen on right subtree which append based on 1,  we will encounter condition as

countOfBits = 2

curSum > n ? 1 : 0 -> always = 0

prevBitIsOne = 0

and it will call at memo[2][0][0]=2 which will return 2 directly, and plus the left subtree total count = 3, total count as 3 + 2 = 5, for n = 4 this is wrong, the actual count should be 4 only (000, 100, 010, 001), 101 represent 5 exceed 4, but since it directly return memo[2][0][0]=2 when condition match, we failed to exclude 101.



No difference between 2 bits scenario when hit 00 and 01, it may caused by 2nd dimension in memo setup not work properly as curSum > n ? 1 : 0 -> always = 0, which cause either 00 or 01 will invoke memo[2][0][0]

3. Is the memo really required in this full binary tree which has no overlapping branch? Looks like no recursion will call the memo part since no overlapping

**One change may help but  test failed (24/527, test case n = 1000000, expected result 17711 but Memo Limit Exceed)**

Directly use curSum instead of curSum > n ? 1 : 0 in memo setup, because curSum will help to identify difference between 00 and 01, and all combination will be unique since we will append 0 or 1 differently which result in unique curSum

class Solution {

public int findIntegers(int n) {

// countOfBits -> max = 32 bits

// curSum -> should not exceed n

// prevBitIsOne -> boolean

Integer[][][] memo = new Integer[32][n + 1][2];

return helper(0, 0, n, false, memo);

}

// One critical setup: why use boolean for 'prevBitIsOne' ?

// Because if we use int to identify the previous bit is 1 or 0, that

// will cause a problem for the LSB (least significant bit), because

// the initial bit will be 0, since we start with number = 0, which

// the LSB is 0, but if we define int prevBitIsOne = 0, then how we

// present number = 1, which requires the LSB = 1 ? So the conflict

// happened, to enable the ability to assign LSB as either 0 or 1, we

// cannot use int to represent 'prevBitIsOne', the alternative way

// is use boolean to enable the ability to assign LSB as either 0 or 1

private int helper(int countOfBits, int curSum, int n, boolean prevBitIsOne, Integer[][][] memo) {

// Base condition

if(curSum > n) {

return 0;

}

// Base condition but difficult to get:

// Only when 1 shift count of bits and larger than n, we recognize

// it as one number, e.g 1 <<< 3 = 8 > 5, then all 3 digits potential

// combination [000 ~ 111] which under 8(=1000)'s cover and able to

// go till this condition is a qualified number

if(1 << countOfBits > n) {

return 1;

}

//int tmp1 = curSum > n ? 1 : 0;

int tmp2 = prevBitIsOne ? 1 : 0;

if(memo[countOfBits][curSum][tmp2] != null) {

return memo[countOfBits][curSum][tmp2];

}

// If the previous bit is 1 we only have one choice to append 0 on left

// e.g previous bit = '1', then append '0' ahead -> '01'

if(prevBitIsOne) {

return memo[countOfBits][curSum][tmp2] = helper(countOfBits + 1, curSum, n,false, memo);

}

// If the previous bit is 0 we can append either 0 or 1

// e.g previous bit = '0', then append '0' ahead -> '00', append '1' ahead -> '10'

return memo[countOfBits][curSum][tmp2] = (helper(countOfBits + 1, curSum, n, false, memo) + helper(countOfBits + 1, curSum + (1 << countOfBits), n, true, memo));

}

}

And we didn't actually resolve the 3rd point mentioned above, looks like all status are unique in this full binary tree, its worthless to create a memo to reserve the status because no same call will happen. Maybe that's why the traditional memoization or 0/1 knapsack won't work here, we need to introduce new thought such as Fibonacci DP way to resolve it

**Solution 3:  DP Fibonacci Sequence + Bit Manipulation (120 min)**

class Solution {

public int findIntegers(int n) {

int[] f = new int[32];

// 0 bit is nothing but a make up for supporting Fibonacci sequence

// e.g f[2] = f[1] + f[0] need to represent how many combinations

// can be represent with 2 bits: 00, 01, 10(no 11 because consecutive

// ones), result count as 3, and we know f[1] = 2, so f[0] set as 1

f[0] = 1;

// 1 bit can represent either 0 or 1, result count as 2

f[1] = 2;

// Build Fibonacci sequence

for(int i = 2; i < f.length; i++) {

f[i] = f[i - 1] + f[i - 2];

}

int count = 0;

// keep a track of the last bit checked to see if its 1

int prev\_bit = 0;

// We find 1 on each bit from MSB (most significant bit)

// and it should only start with i = 30 to i = 0 which is enough

// to cover all bits till max integer (2^31 - 1 = 2147483647 ->

// 01111111111111111111111111111111 = 31 bits of 1 and signal bit 0)

for(int i = 30; i >= 0; i--) {

if((n & (1 << i)) != 0) {

count += f[i];

if(prev\_bit == 1) {

count--;

break;

}

prev\_bit = 1;

} else {

prev\_bit = 0;

}

}

return count + 1;

}

}

Time complexity : O(log2(max\_int)=32). One loop to fill f array and one loop to check all bits of num.

Space complexity : O(log2(max\_int)=32). f array of size 32 is used.

**Refer to**

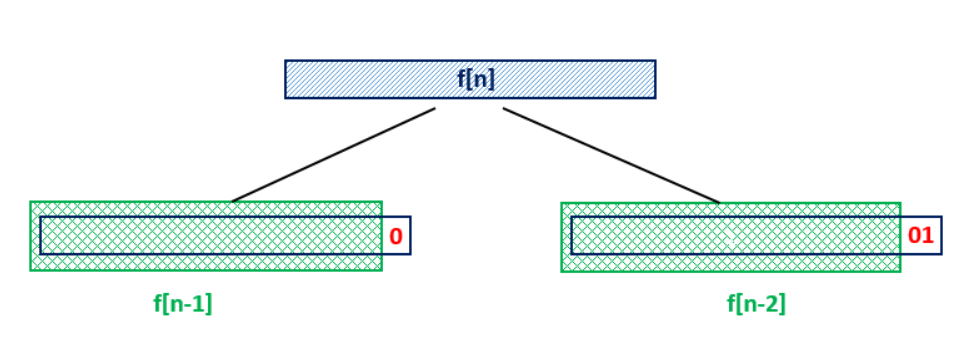
<https://leetcode.com/problems/non-negative-integers-without-consecutive-ones/editorial/>

#### **Approach #3 Using Bit Manipulation [Accepted]**

**Algorithm**

Before we discuss the idea behind this approach, we consider another simple idea that will be used in the current approach.

Suppose, we need to find the count of binary numbers with n bits such that these numbers don't contain consecutive 1's. In order to do so, we can look at the problem in a recursive fashion. Suppose f[i] gives the count of such binary numbers with i bits. In order to determine the value of f[n], which is the requirement, we can consider the cases shown below:



From the above figure, we can see that if we know the value of f[n−1] and f[n−2], in order to generate the required binary numbers with n bits, we can append a 0 to all the binary numbers contained in f[n−1] without creating an invalid number. These numbers give a factor of f[n−1] to be included in f[n]. But, we can't append a 1 to all these numbers, since it could lead to the presence of two consecutive ones in the newly generated numbers. Thus, for the currently generated numbers to end with a 1, we need to ensure that the second last position is always 0. Thus, we need to fix a 01 at the end of all the numbers contained in f[n−2]. This gives a factor of f[n−2] to be included in f[n]. Thus, in total, we get f[n]=f[n−1]+f[n−2].

Now, let's look into the current approach. We'll try to understand the idea behind the approach by taking two simple examples. Firstly, we look at the case where the given number doesn't contain any consecutive 1's.Say,num=1010100(7 bit number). Now, we'll see how we can find the numbers lesser than num ith no two consecutive 1's. We start off with the MSB of nums. If we fix a 0 at the MSB position, and find out the count of 6 bit numbers(corresponding to the 6 LSBs) with no two consecutive 1's, these 6-bit numbers will lie in the range **0**000000 −> **0**111111. For finding this count we can make use of f[6] which we'll have already calculated based on the discussion above.

But, even after doing this, all the numbers in the required range haven't been covered yet. Now, if we try to fix 1 at the MSB, the numbers considered will lie in the range **1**000000 −> **1**111111. As we can see, this covers the numbers in the range **1**000000 −> **1**010100, but it covers the numbers in the range beyond limit as well. Thus, we can't fix 1 at the MSB and consider all the 6-bit numbers at the LSBs.

For covering the pending range, we fix 1 at the MSB, and move forward to proceed with the second digit(counting from MSB). Now, since we've already got a 0 at this position, we can't substitute a 1 here, since doing so will lead to generation of numbers exceeding num. Thus, the only option left here is to substitute a 0 at the second position. But, if we do so, and consider the 5-bit numbers(at the 5 LSBs) with no two consecutive 1's, these new numbers will fall in the range

**10**00000 −> **10**11111. But, again we can observe that considering these numbers leads to exceeding the required range. Thus, we can't consider all the 5-bit numbers for the required count by fixing 0 at the second position.

Thus, now, we fix 0 at the second position and proceed further. Again, we encounter a 1 at the third position. Thus, as discussed above, we can fix a 0 at this position and find out the count of 4-bit consecutive numbers with no two consecutive 1's(by varying only the 4 LSB bits). We can obtain this value from f[4]. Thus, now the numbers in the range

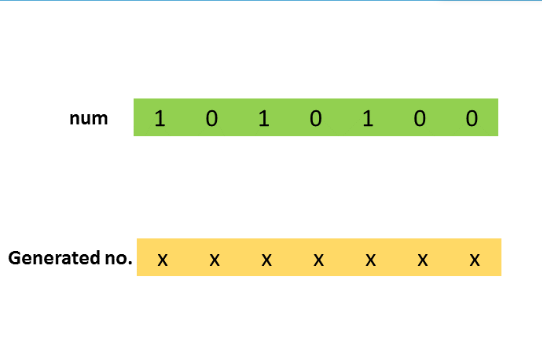
**100**0000 −> **100**1111 have been covered up.

Again, as discussed above, now we fix a 1 at the third position, and proceed with the fourth bit. It is a 0. So, we need to fix it as such as per the above discussion, and proceed with the fifth bit. It is a 1. So, we fix a 0 here and consider all the numbers by varying the two LSBs for finding the required count of numbers in the range **10101**00 −> **10101**11. Now, we proceed to the sixth bit, find a 0 there. So, we fix 0 at the sixth position and proceed to the seventh bit which is again 0. So, we fix a 0 at the seventh position as well.

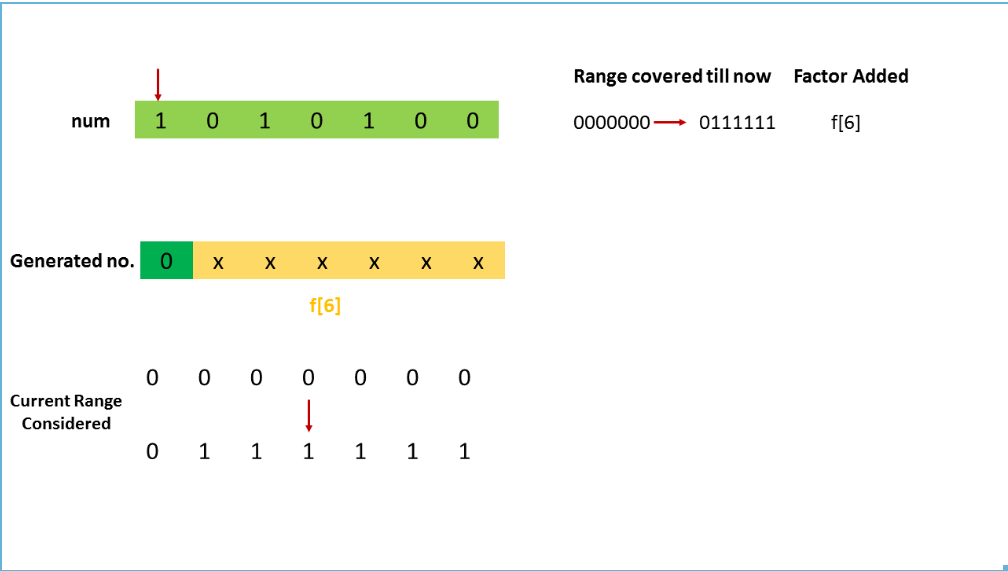
Now, we can see, that based on the above procedure, the numbers in the range **1**000000 −> **1**111111,**100**0000−>**100**1111,**10000**00−>**10000**11 have been considered and the counts for these ranges have been obtained as

f[6], f[4] and f[2] respectively. Now, only 1010100 is pending to be considered in the required count. Since, it doesn't contain any consecutive 1's, we add a 1 to the total count obtained till now to consider this number. Thus, the result returned is f[6]+f[4]+f[2]+1

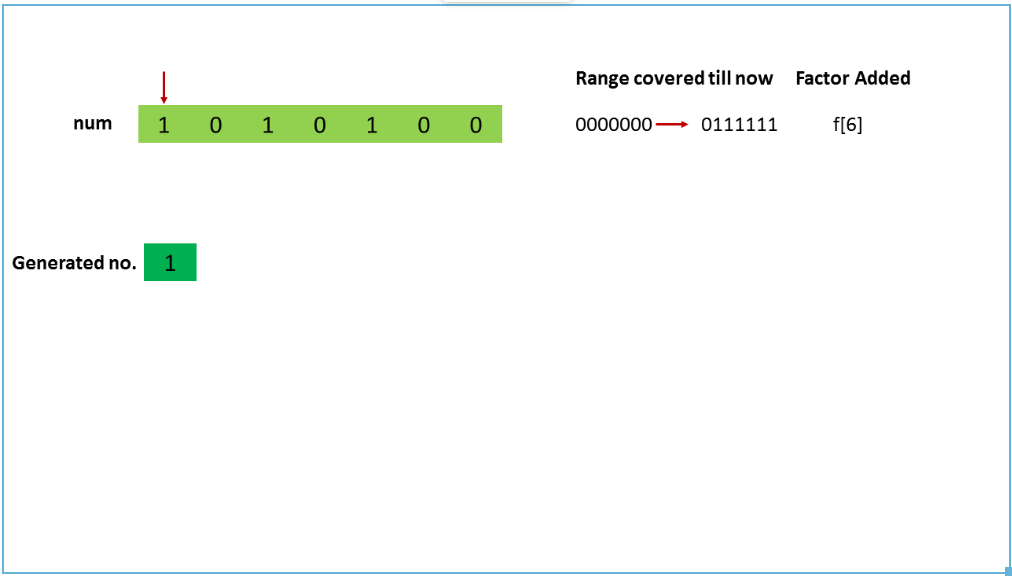
**Step 1:**



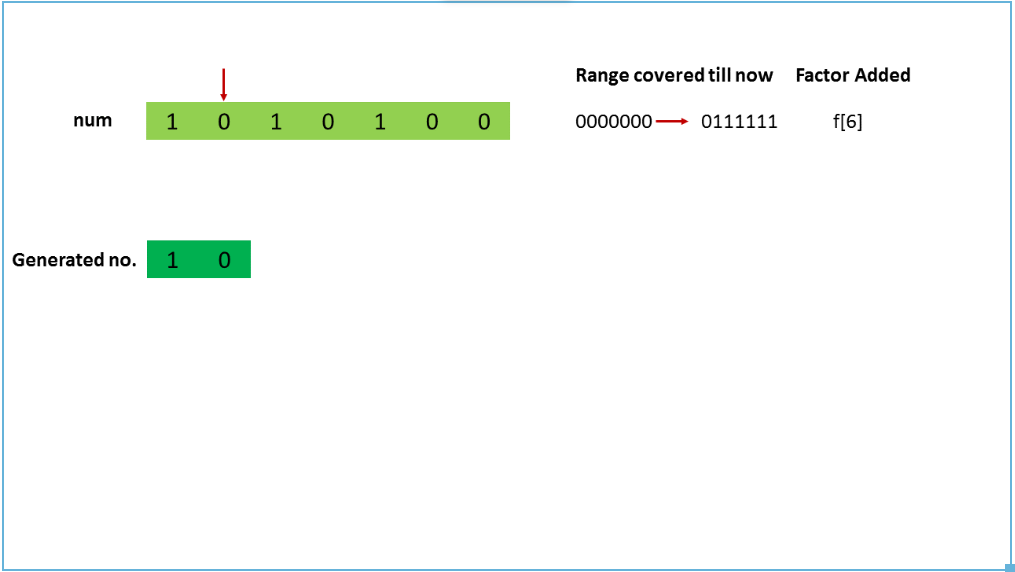
**Step 2:**



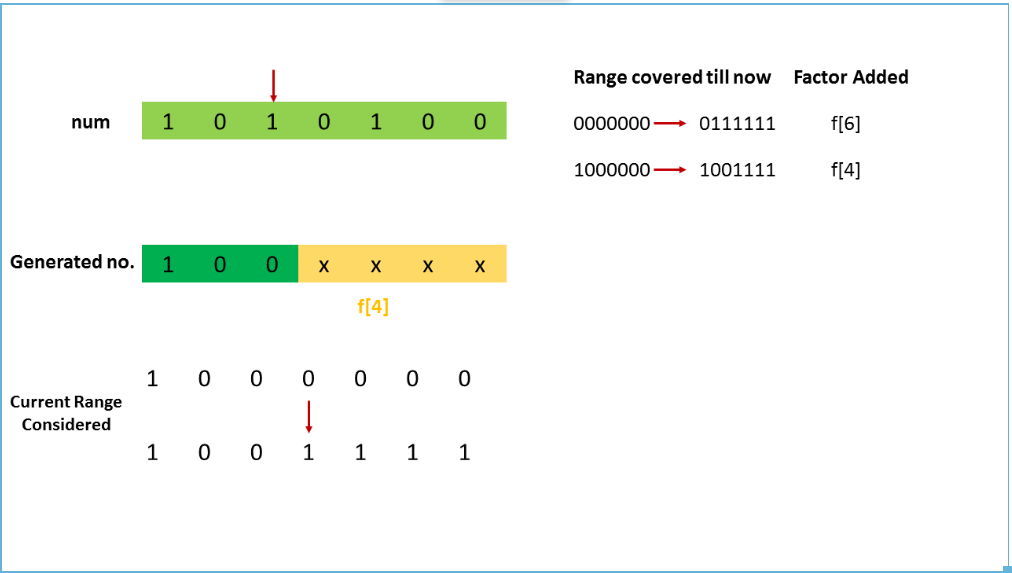
**Step 3:**



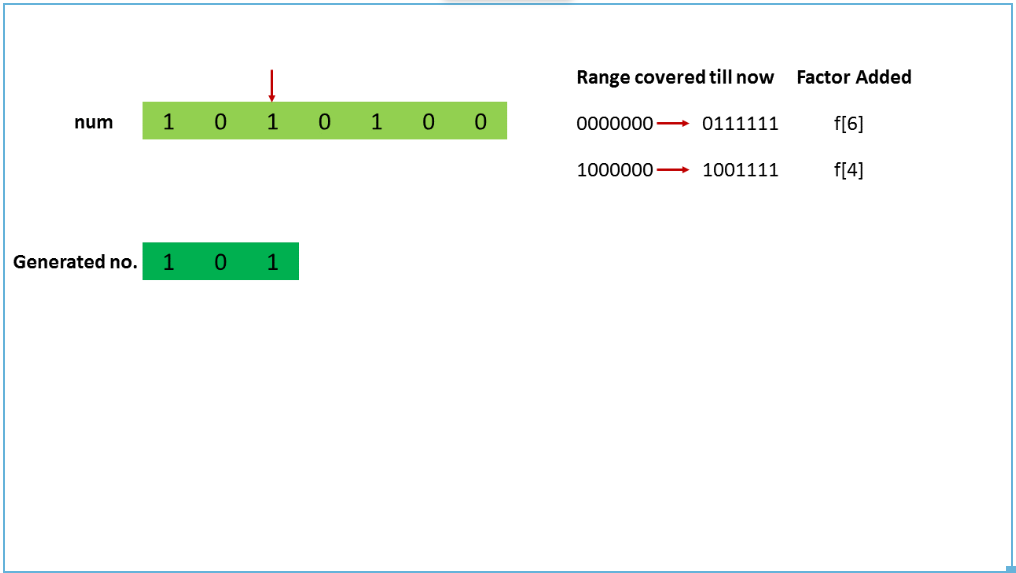
**Step 4:**



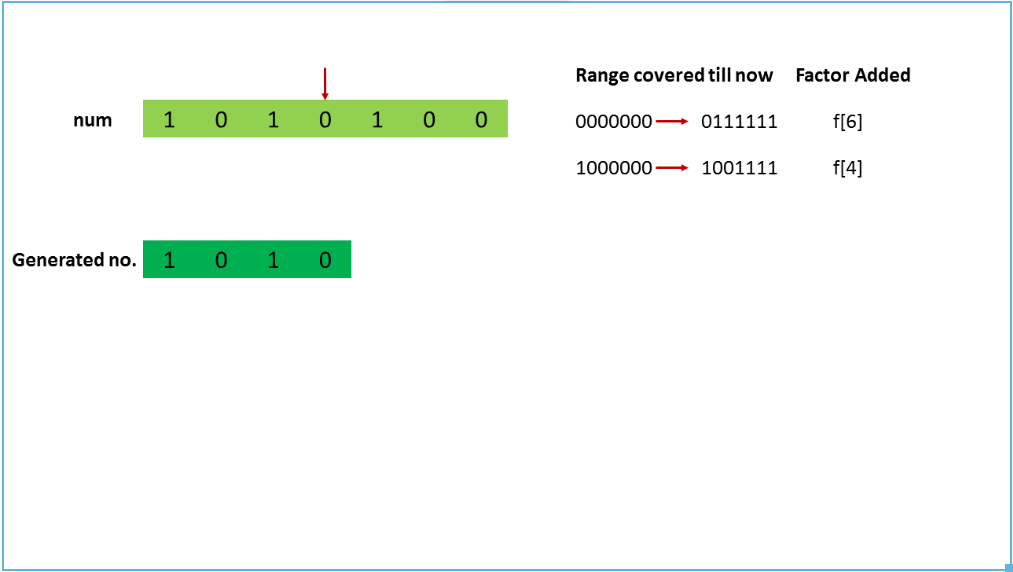
**Step 5:**



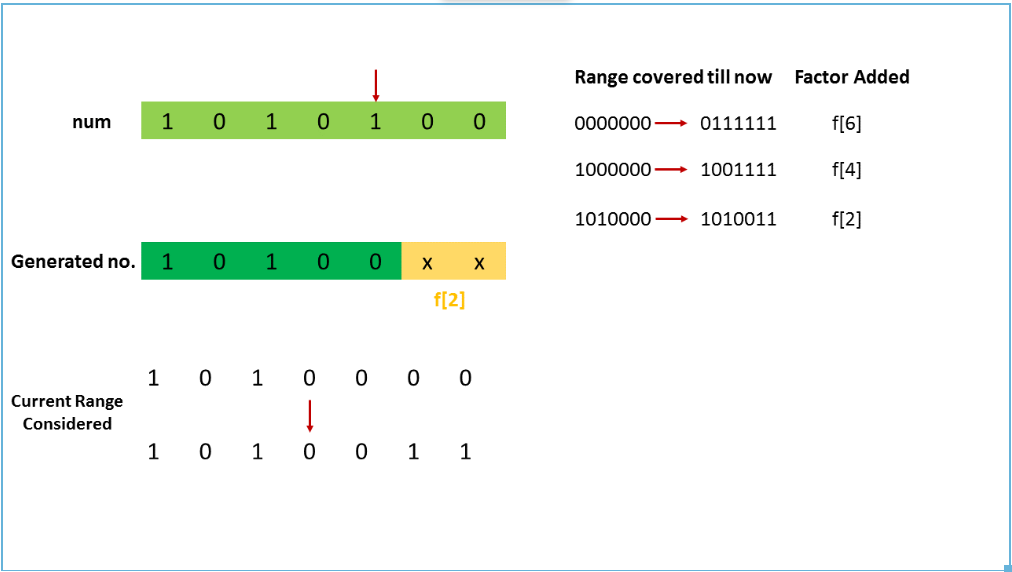
**Step 6:**



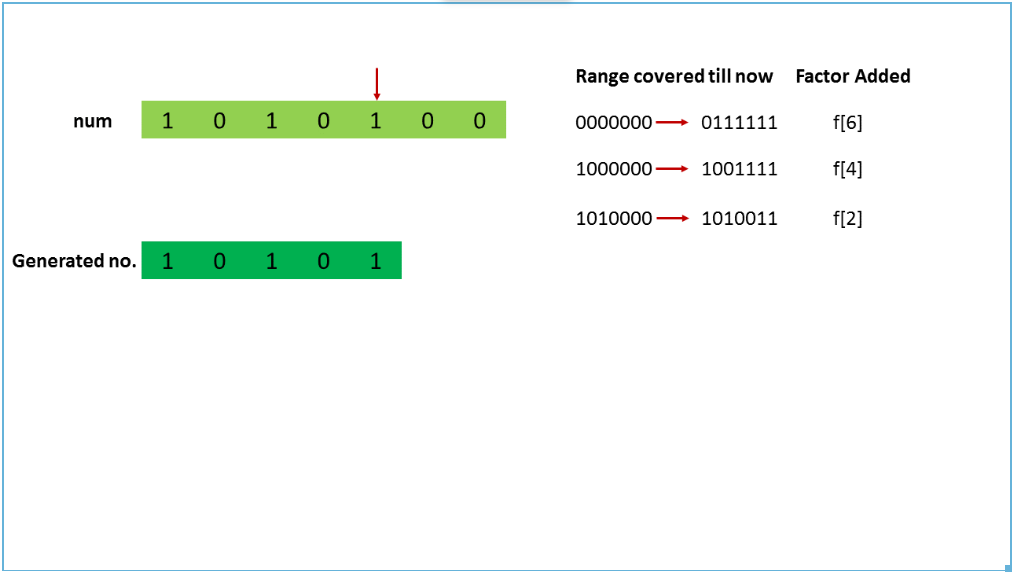
**Step 7:**



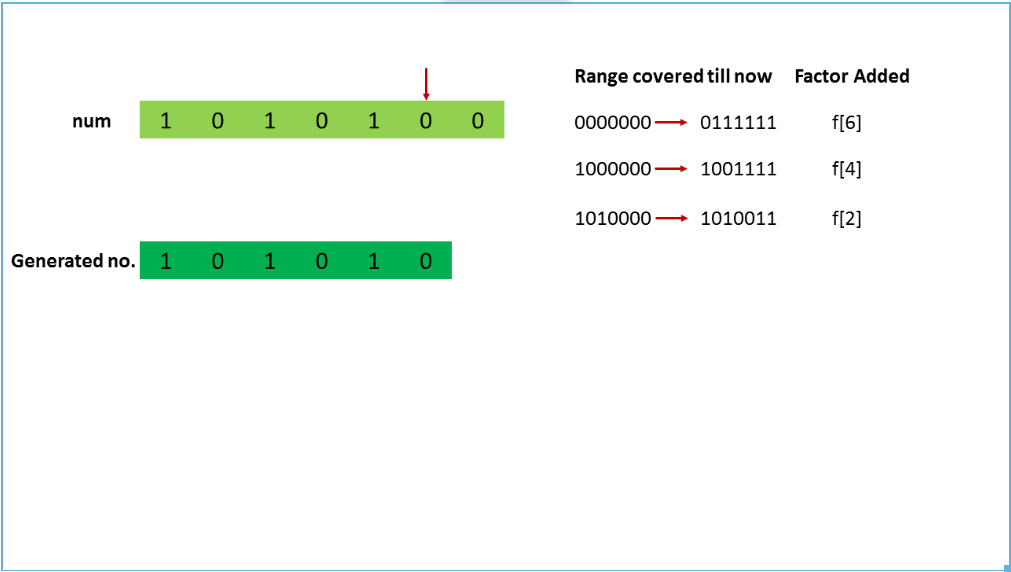
**Step 8:**



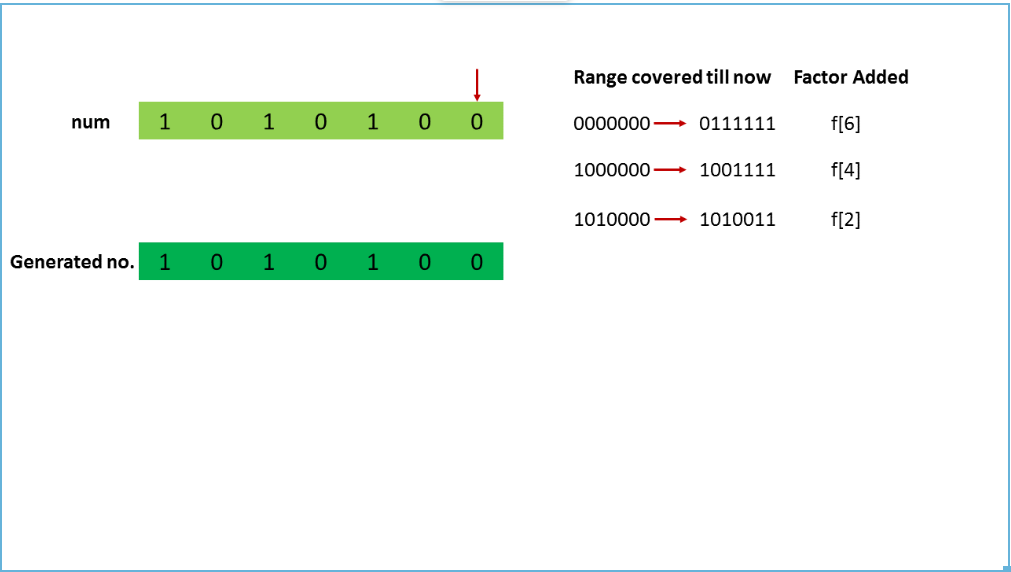
**Step 9:**



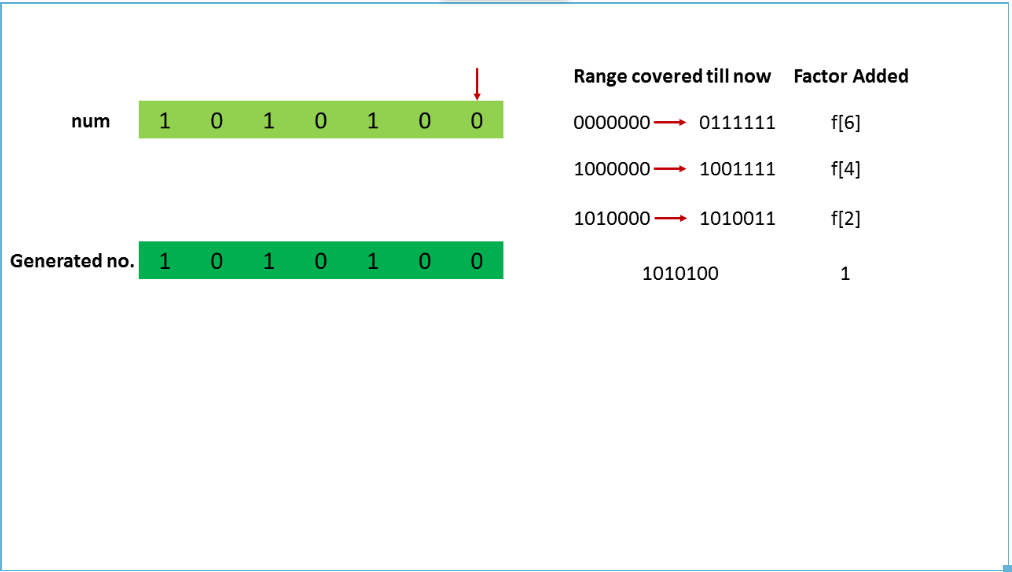
**Step 10:**



**Step 11:**



**Step 12:**



Now, we look at the case, where num contains some consecutive 1's. The idea will be the same as the last example, with the only exception taken when the two consecutive 1's are encountered. Let's say, num=1011010(7 bit number). Now, as per the last discussion, we start with the MSB. We find a 1 at this position. Thus, we initially fix a 0 at this position to consider the numbers in the range **0**000000 −> **0**111111, by varying the 6 LSB bits only. The count of the required numbers in this range is again given by f[6].

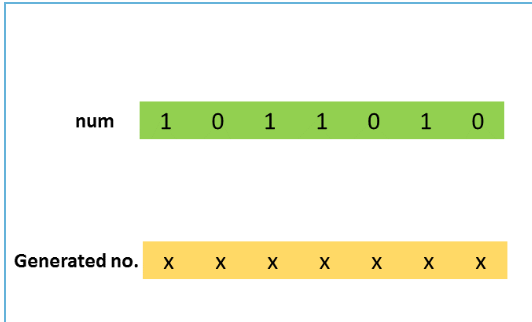
Now, we fix a 1 at the MSB and move on to the second bit. It is a 0, so we have no choice but to fix 0 at this position and to proceed with the third bit. It is a 1, so we fix a 0 here, considering the numbers in the range **100**0000 −> **100**1111. This accounts for a factor of f[4]. Now, we fix a 1 at the third position, and proceed with the fourth bit. It is a 1(consecutive to the previous 1). Now, initially we fix a 0 at the fourth position, considering the numbers in the range **1010**000−>**1010**111. This adds a factor of f[3] to the required count.

Now, we can see that till now the numbers in the range **0**000000 −> **0**111111, **100**0000−>**100**1111, **1010**000−>**1010**111

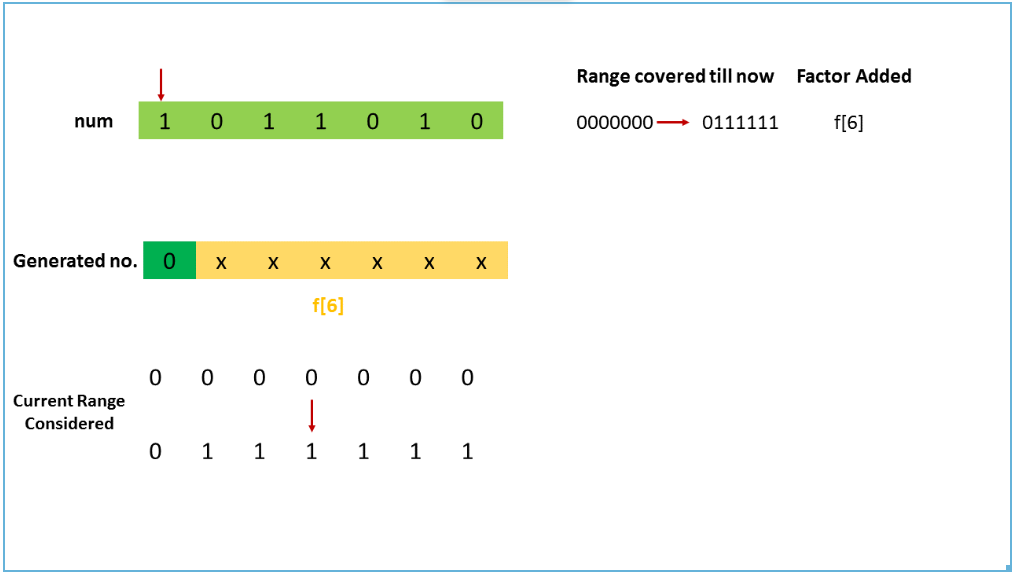
have been considered. But, if we try to consider any number larger than 1010111, it leads to the presence of two consecutive 1's in the new number at the third and fourth position. Thus, all the valid numbers upto num

have been considered with this, giving a resultant count of f[6]+f[4]+f[3].

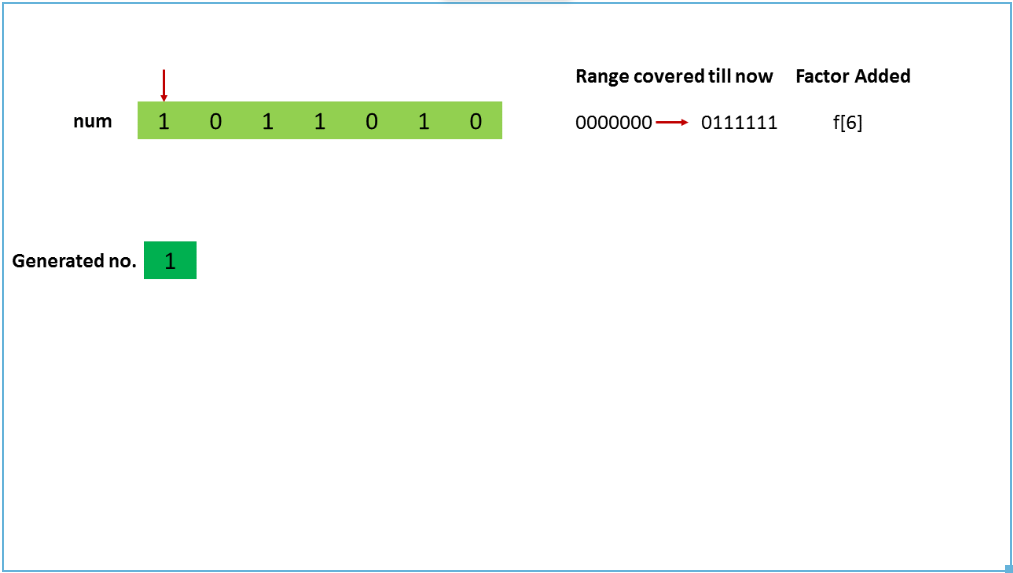
**Step 1:**



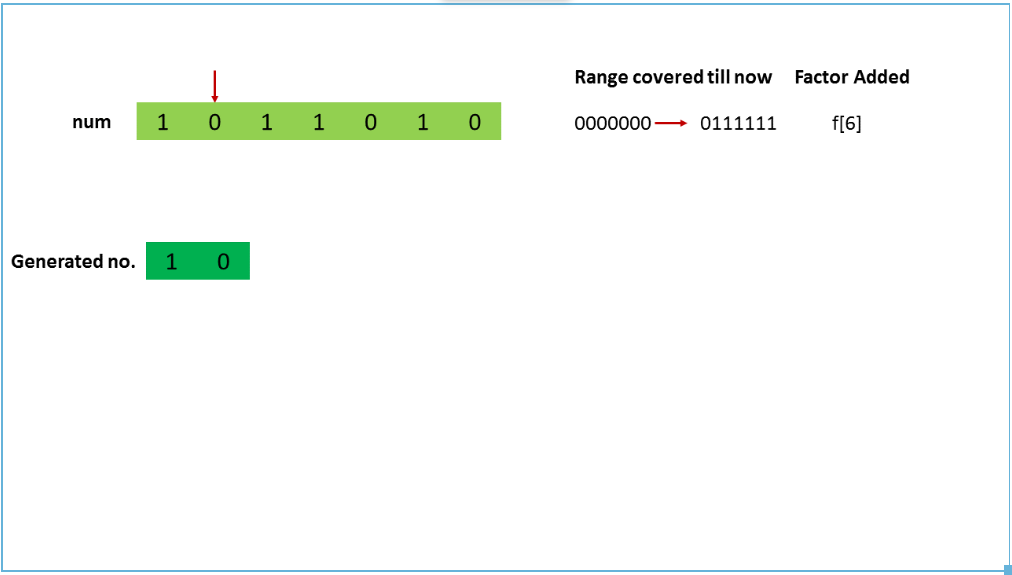
**Step 2:**



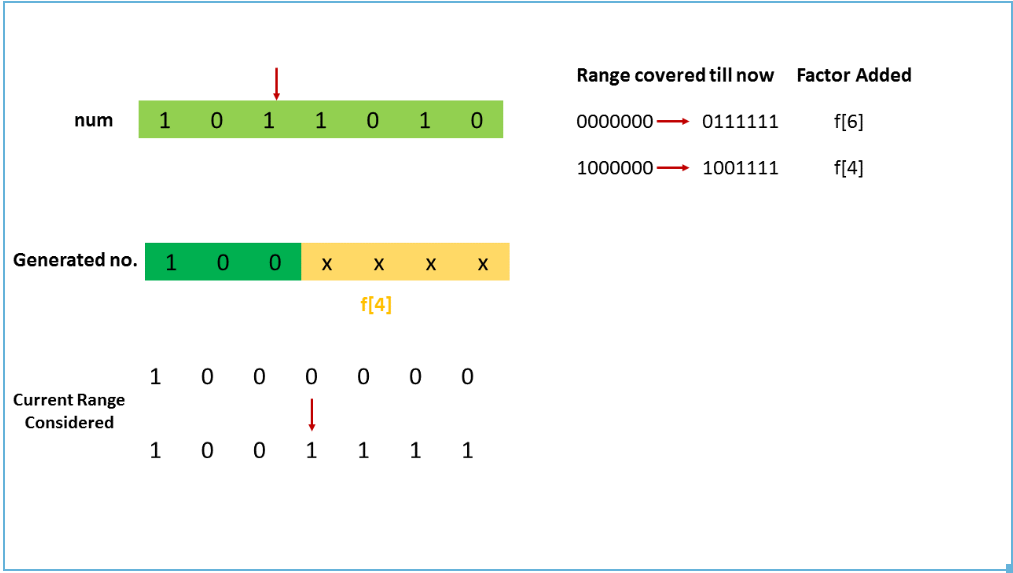
**Step 3:**



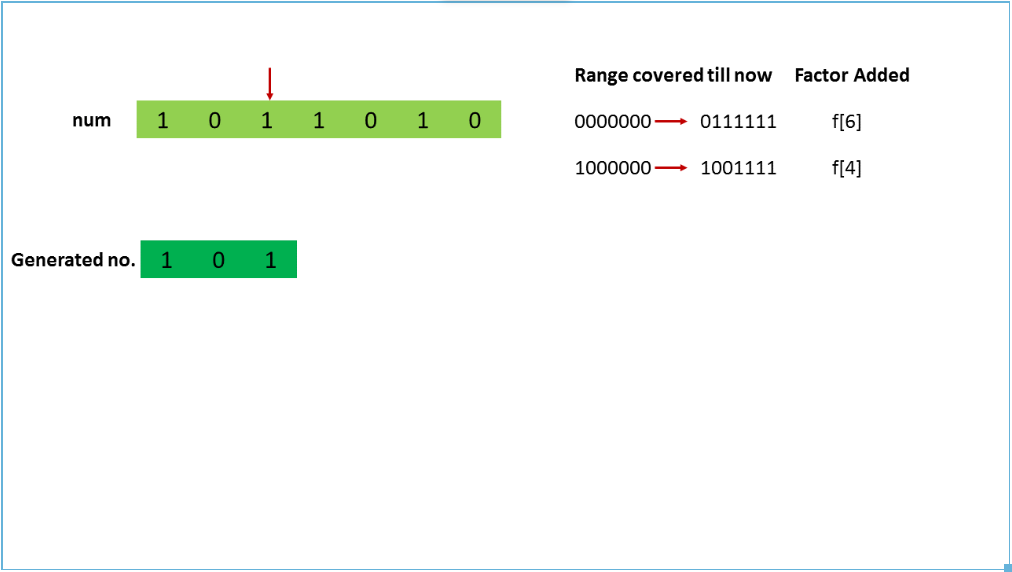
**Step 4:**



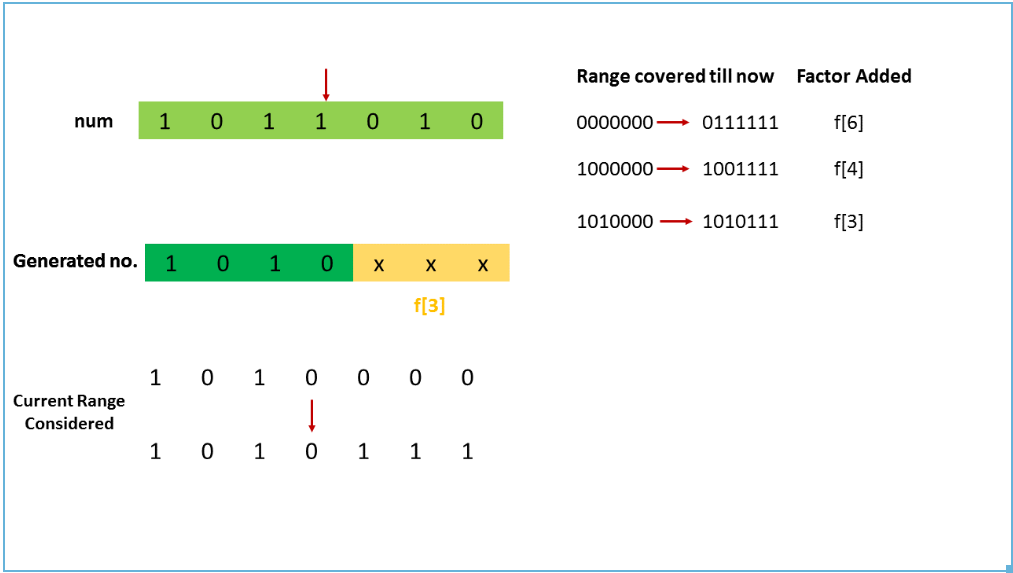
**Step 5:**



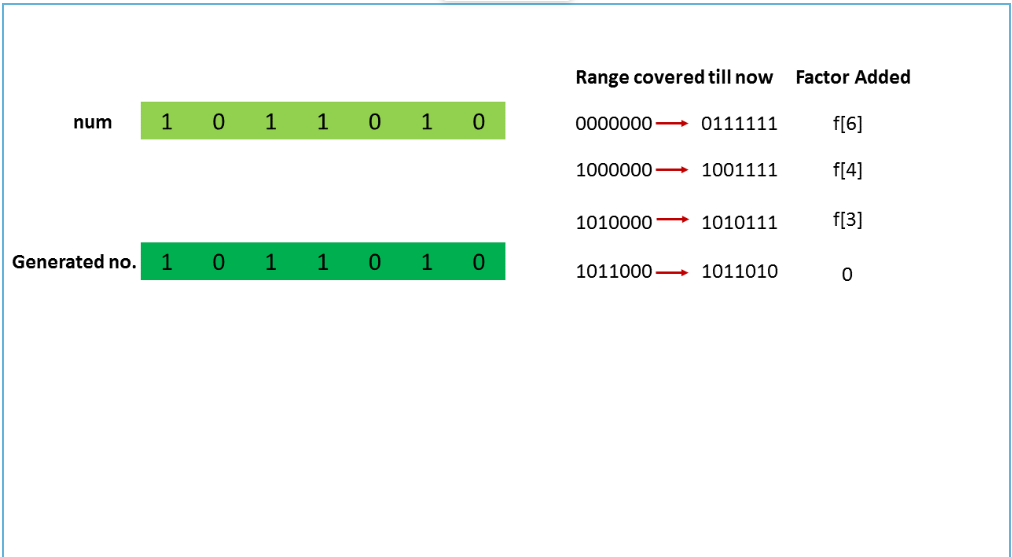
**Step 6:**



**Step 7:**



**Step 8:**



Thus, summarizing the above discussion, we can say that we start scanning the given number num from its MSB. For every 1 encountered at the ith bit position(counting from 0 from LSB), we add a factor of f[i] to the resultant count. For every 0 encountered, we don't add any factor. We also keep a track of the last bit checked. If we happen to find two consecutive 1's at any time, we add the factors for the positions of both the 1's and stop the traversal immediately. If we don't find any two consecutive 1's, we proceed till reaching the LSB and add an extra 1 to account for the given number

num as well, since the procedure discussed above considers numbers upto num without including itself.

public class Solution {

public int findIntegers(int num) {

int[] f = new int[32];

f[0] = 1;

f[1] = 2;

for (int i = 2; i < f.length; i++)

f[i] = f[i - 1] + f[i - 2];

int i = 30, sum = 0, prev\_bit = 0;

while (i >= 0) {

if ((num & (1 << i)) != 0) {

sum += f[i];

if (prev\_bit == 1) {

sum--;

break;

}

prev\_bit = 1;

} else

prev\_bit = 0;

i--;

}

return sum + 1;

}

}

Complexity Analysis

* Time complexity : O(log2(max\_int)=32). One loop to fill f array and one loop to check all bits of num.
* Space complexity : O(log2(max\_int)=32). f array of size 32 is used.

**Refer to**

<https://leetcode.com/problems/non-negative-integers-without-consecutive-ones/solutions/1361794/python3-official-solution-explained-simply-with-diagrams/>

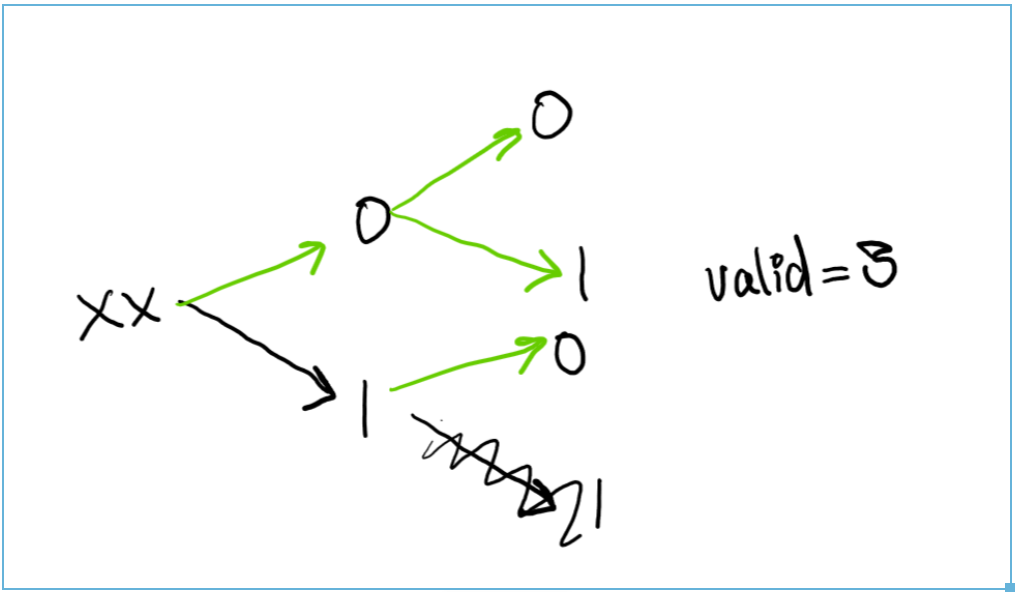
## **0. Introduction**

The problem is hard, but looks deceptively easy. The solution is easy, but hard to understand. I'll attempt to explain the [official solution](https://leetcode.com/problems/non-negative-integers-without-consecutive-ones/solution/) with diagrams.

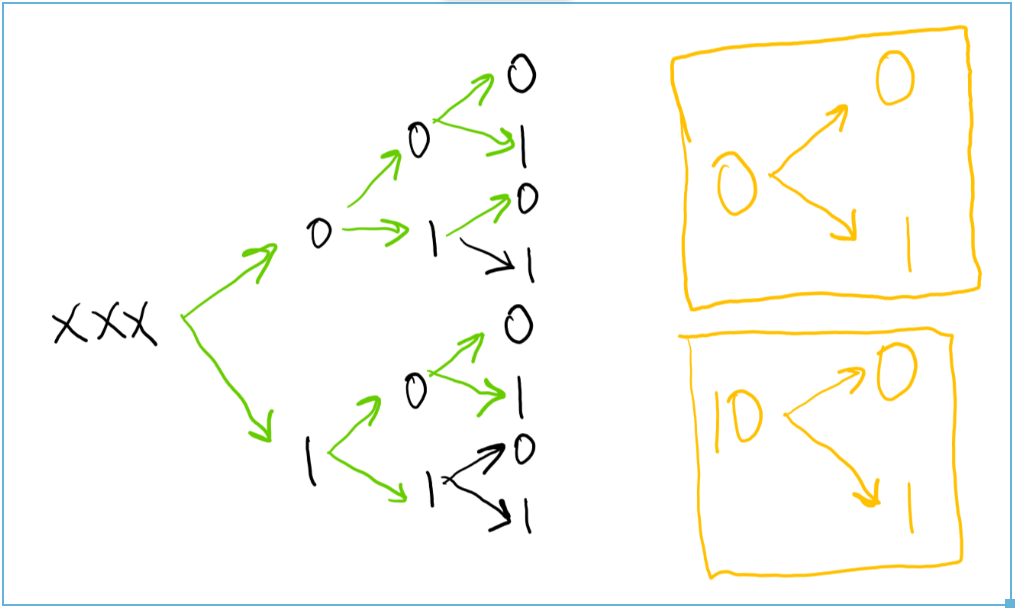
## **1. The World of Bits**

The single constraint given to us is of **no consecutive set bits**. Set bits meaning bits with value of 1. Since the question is in terms of bits, let's think in terms of bits.

Say we have two bits worth of space, xx. What all are the valid combinations? 00, 01, 10.



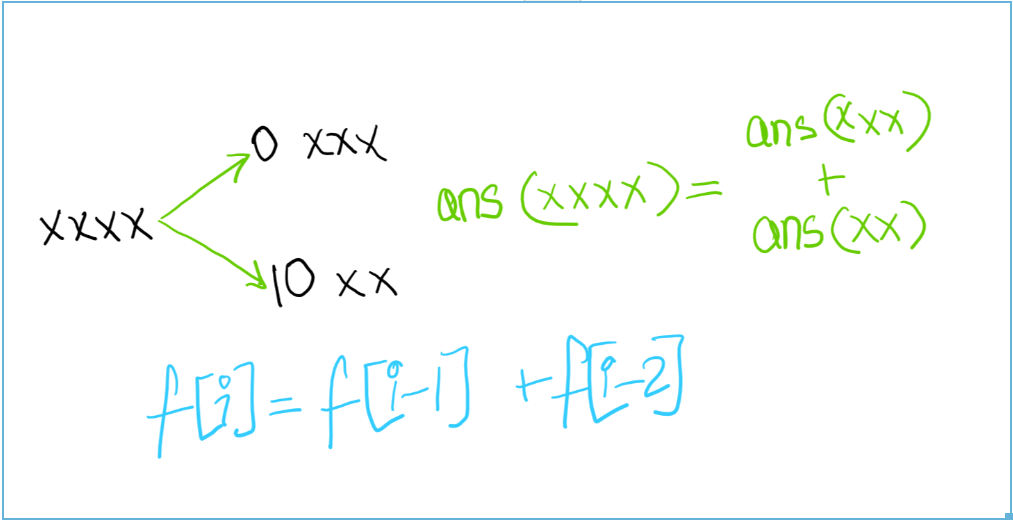
Similarly, for 3 bits as xxx:



The paths reaching till the end with green are counted as valid.

## **2. Observations**

* One thing is glaringly obvious: Only a sequence of 0 and 10 is able to produce both 0, 1 as the output, giving us full freedom over what to pick next. This is marked as yellow.
* Note how the trees are similar. Take the case of 4 bits as: xxxx



That's right, its a fibonacci sequence! This is the core of the solution.

## **3. Examples**

Now we know that its a world of bits with 0 and 10 as the minimum viable sequences giving us fibonacci numbers as the answer.

Since all the numbers can be formed using 1s and 0s, let's take a look at some of the samples.

input: n = "1010"output: In this case we can think of "1010" as "1000+10". We add the answers of "1000" and "10", which can be given by f[3]+f[1]+1. We add the additional 1 because "1010" itself is a valid number!

input: n = "1110"output: Something intersting happends here. Consider "1110 = 1000+110". Answer for "1000" is f[3].What about "110"? Can we break it down as "110 = 100+10"? Not so fast!The issue is, "110" has a hidden 1 behind it. Meaning, we can consider numbers **upto** "100", sure (*not* inlcuding "100"), but any number beyond that is simply invalid by the condition. So, we only consider "1110 as 1000+100". The answer is thus f[3]+f[2]. Note how we didn't do +1 because "1100" is an invalid number.

input: n = "101 1010"output: "101 1010 = 100 0000 + 1 0000 + 1000" => f[6]+f[4]+f[3]. The rest would become invalid. If you are confued, reread! This combines elements of both of the above.

## **4. Code**

This code is a direct translation of the official solution, but this time, it should make sense!

class Solution:

def findIntegers(self, n: int) -> int:

# f stores the fibonacci numbers

f = [1, 2]

for i in range(2, 30):

f.append(f[-1]+f[-2])

# last\_seen tells us if there was a one right before.

# If that is the case, we are done then and there!

# ans is the answer

ans, last\_seen = 0, 0

for i in reversed(range(30)):

if (1 << i) & n: # is the ith bit set?

ans += f[i]

if last\_seen:

ans -= 1

break

last\_seen = 1

else:

last\_seen = 0

return ans+1

<https://leetcode.com/problems/non-negative-integers-without-consecutive-ones/solutions/1361794/python3-official-solution-explained-simply-with-diagrams/comments/1024712>

***Q: Can you please elaborate this why is this happening??***

input: n = "1110"

output: Something intersting happends here. Consider "1110 = 1000+110". Answer for "1000" is f[3].

What about "110"? Can we break it down as "110 = 100+10"? Not so fast!

The issue is, "110" has a hidden 1 behind it. Meaning, we can consider numbers upto "100", sure (not inlcuding "100"), but any number beyond that is simply invalid by the condition. So, we only consider "1110 as 1000+100". The answer is thus f[3]+f[2]. Note how we didn't do +1 because "1100" is an invalid number.

***A:***

So for a number n, at every bit that is a set to a 1, we are replacing it with a 0 then counting the number remaining bits to the right of it.

Example:

"[1]110" -> "[0]XXX"

[1] indicates the bit we are at, there are 3 bits to the right hence 3 X's

XXX = F(3) = third fibonacci number --> There are F(3) ways to make XXX have non-consecutive 1's --> Add F(3) to count

"1[1]10" -> "1[0]XX"

XX = F(2) --> Add F(2) to the count

"11[1]0" -> "11[0]X"

Note that here, X = F(1) --> There are F(1) ways to make X have non-consecutive 1's but we cannot add F(1) to the count because no matter what you change X to, the number is already invalid.