# Question

On the first row, we write a 0. Now in every subsequent row, we look at the previous row and replace each occurrence of 0 with 01, and each occurrence of 1 with 10.

Given row N and index K, return the K-th indexed symbol in row N. (The values of K are 1-indexed.) (1 indexed).

**Examples:**

**Input:** N = 1, K = 1

**Output:** 0

**Input:** N = 2, K = 1

**Output:** 0

**Input:** N = 2, K = 2

**Output:** 1

**Input:** N = 4, K = 5

**Output:** 1

**Explanation:**

row 1: 0

row 2: 01

row 3: 0110

row 4: 01101001

**Note:**

1. N will be an integer in the range [1, 30].
2. K will be an integer in the range [1, 2^(N-1)].

# Solution

#### **Approach 1: Brute Force**

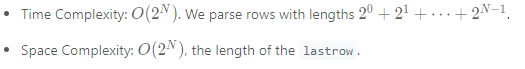
**Intuition and Algorithm**

We'll make each row exactly as directed in the problem statement. We only need to remember the last row.

Unfortunately, the strings could have length around 1 billion, as they double on each row, so this approach is not efficient enough.

|  |
| --- |
| class Solution {  public int kthGrammar(int N, int K) {  int[] lastrow = new int[1 << N];  for (int i = 1; i < N; ++i) {  for (int j = (1 << (i-1)) - 1; j >= 0; --j) {  lastrow[2\*j] = lastrow[j];  lastrow[2\*j+1] = 1 - lastrow[j];  }  }  return lastrow[K-1];  }  } |

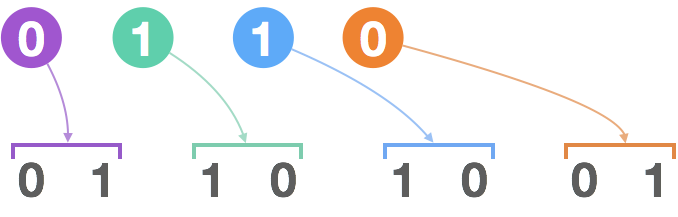
**Complexity Analysis**



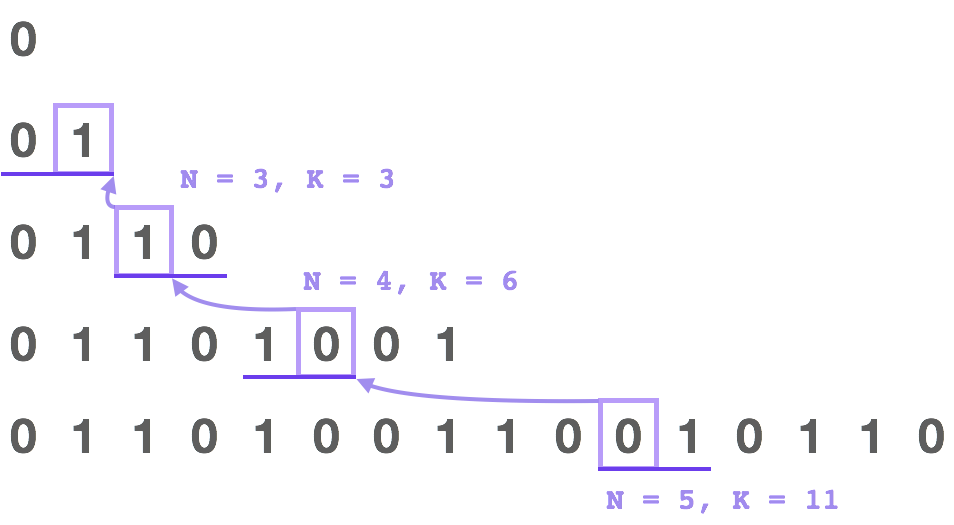
#### **Approach 2: Recursion (Parent Variant)**

**Intuition and Algorithm**

Since each row is made only using information from the previous row, let's try to write the answer in terms of bits from the previous row.



In particular, if we write say "0110" which generates "01101001", then the first "0" generates the first "01" in the next row; the next digit "1" generates the next "10", the next "1" generates the next "10", and the last "0" generates the last "01".



In general, the Kth digit's parent is going to be (K+1) / 2. If the parent is 0, then the digit will be the same as 1 - (K%2). If the parent is 1, the digit will be the opposite, ie. K%2.

|  |
| --- |
| class Solution {  public int kthGrammar(int N, int K) {  if (N == 1) return 0;  return (~K & 1) ^ kthGrammar(N-1, (K+1)/2);  }  } |

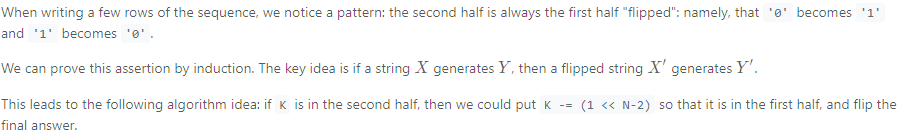
**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*). It takes N-1*N*−1 steps to find the answer.
* Space Complexity: O(1)*O*(1).

#### **Approach 3: Recursion (Flip Variant)**

**Intuition and Algorithm**

As in Approach #2, we could try to write the bit in terms of it's previous bit.



|  |
| --- |
| class Solution {  public int kthGrammar(int N, int K) {  if (N == 1) return 0;  if (K <= 1 << N-2)  return kthGrammar(N-1, K);  return kthGrammar(N-1, K - (1 << N-2)) ^ 1;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*). It takes N-1*N*−1 steps to find the answer.
* Space Complexity: O(1)*O*(1).

#### **Approach 4: Binary Count**

**Intuition and Algorithm**

As in Approach #3, the second half of every row is the first half flipped.

When the indexes K are written in binary (now indexing from zero), indexes of the second half of a row are ones with the first bit set to 1.

This means when applying the algorithm in Approach #3 virtually, the number of times we will flip the final answer is just the number of 1s in the binary representation of K-1.

|  |
| --- |
| class Solution {  public int kthGrammar(int N, int K) {  return Integer.bitCount(K - 1) % 2;  }  } |

**Complexity Analysis**

* Time Complexity: O(\log N)*O*(log*N*), the number of binary bits in N. If \log Nlog*N* is taken to be bounded, this can be considered to be O(1)*O*(1).
* Space Complexity: O(1)*O*(1). (In Python, bin(X) creates a string of length O(\log X)*O*(log*X*), which could be avoided.)