**The countSetBits method counts the number of 1-bits (set bits) in the binary representation of an integer n. Here's a step-by-step explanation with an example:**

**How It Works**

1. **Initialization:**
   * **count is initialized to 0. This variable will keep track of the number of set bits.**
2. **While Loop:**
   * **The loop continues as long as n is greater than 0.**
3. **Count Set Bits:**
   * **count += n & 1: This operation checks if the least significant bit of n is 1. If it is, n & 1 evaluates to 1; otherwise, it evaluates to 0. This value is added to count.**
4. **Right Shift:**
   * **n >>= 1: This operation performs a right bitwise shift on n, effectively removing the least significant bit. The next iteration of the loop will then process the new least significant bit.**
5. **Return Result:**
   * **After exiting the loop, count contains the total number of set bits in the original value of n.**

**Example**

**Let's walk through the method with an example where n = 29.**

**Binary Representation of 29: 11101**

1. **Initial State:**
   * **n = 29 (11101 in binary)**
   * **count = 0**
2. **First Iteration:**
   * **n & 1 → 11101 & 00001 → 1 (least significant bit is 1)**
   * **count += 1 → count = 1**
   * **n >>= 1 → 11101 >> 1 → 01110 (removes the least significant bit)**
3. **Second Iteration:**
   * **n & 1 → 01110 & 00001 → 0 (least significant bit is 0)**
   * **count += 0 → count = 1**
   * **n >>= 1 → 01110 >> 1 → 00111**
4. **Third Iteration:**
   * **n & 1 → 00111 & 00001 → 1 (least significant bit is 1)**
   * **count += 1 → count = 2**
   * **n >>= 1 → 00111 >> 1 → 00011**
5. **Fourth Iteration:**
   * **n & 1 → 00011 & 00001 → 1 (least significant bit is 1)**
   * **count += 1 → count = 3**
   * **n >>= 1 → 00011 >> 1 → 00001**
6. **Fifth Iteration:**
   * **n & 1 → 00001 & 00001 → 1 (least significant bit is 1)**
   * **count += 1 → count = 4**
   * **n >>= 1 → 00001 >> 1 → 00000 (now n is 0)**
7. **Loop Ends:**
   * **Since n is now 0, the loop exits.**
8. **Return Result:**
   * **The final value of count is 4.**

**Result: The number of set bits (1s) in the binary representation of 29 is 4.**

**Summary**

* **The method iteratively checks each bit of n, counts the number of set bits, and returns this count.**
* **It effectively examines each bit from least significant to most significant by repeatedly right-shifting n until it becomes 0.**

**Problem Statement**

Shawn, a computer science enthusiast, is curious about binary strings and their patterns. He wants to find out how many binary strings of length N contain consecutive 1s. Help Shawn by writing a program to calculate the number of such binary strings.

**Example**

**Input:** N = 2

**Output:** 1

**Explanation:** There are 4 strings of length 2, the strings are 00, 01, 10, and 11. Only the string 11 has consecutive 1's.

**Input format :**

The first line of input consists of an integer **N**, representing the length of the binary strings.

**Output format :**

The first line of output displays the count of binary strings of length N that contain consecutive 1s.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

2 ≤ N ≤ 30

**Sample test cases :**

**Input 1 :**

2

**Output 1 :**

1

**Input 2 :**

5

**Output 2 :**

19

**Solution:**

// You are using Java

import java.util.Scanner;

class BinaryStringWithConsecutive1s {

    // Function to count binary strings of length N without consecutive 1s

    public static int countNonConsecutive1s(int N) {

        // Base cases:

        // If length is 1, there are 2 valid binary strings: "0" and "1"

        if (N == 1) return 2;

        // If length is 2, there are 3 valid binary strings: "00", "01", and "10"

        if (N == 2) return 3;

        // Create an array to store the count of binary strings for each length up to N

        int[] dp = new int[N + 1];

        // Initialize base cases in the dp array

        dp[1] = 2; // For length 1

        dp[2] = 3; // For length 2

        // Fill the dp array for lengths from 3 to N

        // dp[i] is derived from dp[i - 1] and dp[i - 2]

        for (int i = 3; i <= N; i++) {

            // The number of binary strings of length i without consecutive 1s

            // is the sum of binary strings of length (i-1) and (i-2)

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

        }

        // Return the number of valid binary strings of length N

        return dp[N];

    }

    public static void main(String[] args) {

        // Create a Scanner object for user input

        Scanner scanner = new Scanner(System.in);

        // Read the length N of the binary strings

        int N = scanner.nextInt();

        // Handle the case when N is less than 2

        if (N < 2) {

            System.out.println("0");

            return; // No valid strings for length less than 2

        }

        // Calculate the total number of binary strings of length N

        // 2^N is equivalent to 1 << N (bitwise left shift)

        int totalBinaryStrings = 1 << N; // 2^N

        // Call the function to count binary strings without consecutive 1s

        int nonConsecutive1s = countNonConsecutive1s(N);

        // Calculate the number of binary strings with consecutive 1s

        // by subtracting the count of non-consecutive strings from the total

        int consecutive1s = totalBinaryStrings - nonConsecutive1s;

        // Output the count of binary strings with consecutive 1s

        System.out.println(consecutive1s);

    }

}

**Base Cases**

Base cases are special conditions that handle the smallest or simplest instances of a problem. In your dynamic programming approach, base cases are necessary to kickstart the recursion or iteration.

Here’s how the base cases are used in this problem:

**1. Base Case for Length 1 (N = 1)**:

* **Explanation**: For binary strings of length 1, there are only two possible strings: "0" and "1". Both of these do not contain consecutive 1s.
* **Base Case**: if (N == 1) return 2;
* **Reason**: There are exactly 2 strings: "0" and "1", both are valid.

**2. Base Case for Length 2 (N = 2)**:

* **Explanation**: For binary strings of length 2, you have the following possible strings: "00", "01", "10", and "11".
  + "00" (no consecutive 1s)
  + "01" (no consecutive 1s)
  + "10" (no consecutive 1s)
  + "11" (has consecutive 1s)
* **Base Case**: if (N == 2) return 3;
* **Reason**: There are 3 valid strings without consecutive 1s: "00", "01", "10".

**Dynamic Programming Transition**

For lengths greater than 2, the number of valid binary strings can be built up from smaller lengths:

* **Transition Formula**:
  + dp[i] = dp[i - 1] + dp[i - 2]
  + **Explanation**:
    - **Strings Ending in "0"**: A valid string of length iii ending in "0" can be derived from a valid string of length i−1i - 1i−1 (since adding "0" to any valid string of length i−1i - 1i−1 does not create consecutive 1s).
    - **Strings Ending in "10"**: A valid string of length iii ending in "10" can be derived from a valid string of length i−2i - 2i−2 (since adding "10" to any valid string of length i−2i - 2i−2 also does not create consecutive 1s).

**Putting It All Together**

Here’s a brief summary of the approach:

1. **Handle Small Cases Directly**:
   * Use base cases to quickly return results for lengths 1 and 2.
2. **Use Dynamic Programming for Larger Cases**:
   * Build up solutions for larger strings based on previously computed results.
3. **Calculate and Print Results**:
   * Compute the total number of strings, subtract those without consecutive 1s to get those with consecutive 1s, and print the results.

This approach efficiently handles the problem by leveraging dynamic programming to avoid redundant calculations and by providing base cases for the simplest scenarios.

Certainly! Let’s dive into how the dynamic programming formula dp[i] = dp[i - 1] + dp[i - 2] works with a concrete example. We’ll use the example of binary strings of length N = 5 to demonstrate the calculation.

**Example: Length N = 5**

**Objective**: Calculate the number of binary strings of length 5 that do not have consecutive 1s.

**Steps:**

1. **Base Cases**:
   * dp[1] (length 1): There are 2 valid binary strings: "0", "1".
     + Thus, dp[1] = 2.
   * dp[2] (length 2): There are 3 valid binary strings: "00", "01", "10".
     + Thus, dp[2] = 3.
2. **Recursive Formula**:
   * The formula dp[i] = dp[i - 1] + dp[i - 2] is used to build the solution for strings of length i based on the solutions for strings of length i - 1 and i - 2.

**Explanation**:

* + **Strings Ending in "0"**: For length i, a valid string can end in "0", and this is derived from valid strings of length i - 1.
  + **Strings Ending in "10"**: For length i, a valid string can end in "10", and this is derived from valid strings of length i - 2.

**Calculation:**

1. **Calculate dp[3]**:
   * **Valid strings of length 3**:
     + Adding "0" to valid strings of length 2: "000", "010", "100"
     + Adding "10" to valid strings of length 1: "010", "110"
   * **Count**:
     + Valid strings: "000", "001", "010", "100", "101"
   * Thus, dp[3] = 5.
2. **Calculate dp[4]**:
   * **Valid strings of length 4**:
     + Adding "0" to valid strings of length 3: "0000", "1000", "0100", "0001", "0101"
     + Adding "10" to valid strings of length 2: "0010", "1010", "1100", "0110"
   * **Count**:
     + Valid strings: "0000", "0001", "0010", "0100", "0101", "1000", "1001", "1010", "1100", "0110"
   * Thus, dp[4] = 8.
3. **Calculate dp[5]**:
   * **Valid strings of length 5**:
     + Adding "0" to valid strings of length 4: "00000", "00010", "01000", "10000", "00100", "10100"
     + Adding "10" to valid strings of length 3: "00100", "01010", "10010", "11000"
   * **Count**:
     + Valid strings: "00000", "00001", "00010", "00100", "01000", "01001", "01010", "10000", "10001", "10010", "10100", "11000", "01100"
   * Thus, dp[5] = 13.

**Summary of Values:**

* dp[1] = 2
* dp[2] = 3
* dp[3] = 5
* dp[4] = 8
* dp[5] = 13

**Visualization:**

To visualize, here’s a summary table for dp values:

| **Length (i)** | **Number of Valid Strings (dp[i])** |
| --- | --- |
| 1 | 2 |
| 2 | 3 |
| 3 | 5 |
| 4 | 8 |
| 5 | 13 |

The recurrence relation dp[i] = dp[i - 1] + dp[i - 2] helps build the solution for larger lengths using previously computed results, making the approach efficient and avoiding redundant calculations.

Let’s break down the use and purpose of the main method and the initial conditions in the provided code. This will clarify how the code operates with different input values and how it processes the calculation of binary strings.

**Overview of the main Method**

**Purpose**:

* The main method reads user input, processes it to calculate the number of binary strings of a given length that contain consecutive 1s, and outputs the result.

**Breakdown of main Method Steps**

1. **Read User Input**:

java

Copy code

Scanner scanner = new Scanner(System.in);

int N = scanner.nextInt();

* + Creates a Scanner object to read input from the console.
  + Reads an integer N, which represents the length of the binary strings to be analyzed.

1. **Handle Special Case for N < 2**:

java

Copy code

if (N < 2) {

System.out.println("0");

return;

}

* + Checks if N is less than 2.
  + If N < 2, prints "0" and exits. This is because for binary strings of length 0 or 1, there are no valid binary strings with consecutive 1s (length 1 has no consecutive 1s, and length 0 doesn’t exist).

1. **Calculate Total Number of Binary Strings**:

java

Copy code

int totalBinaryStrings = 1 << N; // 2^N

* + Calculates the total number of binary strings of length N.
  + Uses bitwise left shift (1 << N) which is equivalent to 2N2^N2N.

1. **Count Binary Strings Without Consecutive 1s**:

java

Copy code

int nonConsecutive1s = countNonConsecutive1s(N);

* + Calls the countNonConsecutive1s method to get the number of binary strings of length N that do not have consecutive 1s.

1. **Calculate Binary Strings With Consecutive 1s**:

java

Copy code

int consecutive1s = totalBinaryStrings - nonConsecutive1s;

* + Subtracts the count of non-consecutive strings from the total number of strings to get the count of strings with consecutive 1s.

1. **Output the Result**:

java

Copy code

System.out.println(consecutive1s);

* + Prints the number of binary strings of length N that contain consecutive 1s.

**Initial Conditions**

**Initial Conditions**:

* **For N < 2**:
  + **Condition**: If the length is less than 2, no valid strings with consecutive 1s exist. Therefore, the result is directly "0".

**Dynamic Programming Array Initialization**:

* **Base Cases**:
  + dp[1] = 2: Represents the binary strings of length 1: "0", "1".
  + dp[2] = 3: Represents the binary strings of length 2: "00", "01", "10".

**Recursive Calculation**:

* For lengths greater than 2, the number of valid strings is calculated using the recurrence relation:
  + dp[i] = dp[i - 1] + dp[i - 2].

**Example Scenario**

Suppose the user inputs N = 4. Here’s how the main method processes this input:

1. **Calculate Total Binary Strings**:
   * totalBinaryStrings = 1 << 4 = 16.
2. **Count Strings Without Consecutive 1s**:
   * nonConsecutive1s = countNonConsecutive1s(4) = 8 (as calculated using the countNonConsecutive1s function).
3. **Calculate Strings With Consecutive 1s**:
   * consecutive1s = 16 - 8 = 8.
4. **Output**:
   * The program prints 8, indicating there are 8 binary strings of length 4 with at least one occurrence of consecutive 1s.

**Summary**

* **Initial Condition**: Handles cases where N<2N < 2N<2 by directly outputting 0.
* **Main Calculation**: Uses the total count of binary strings and subtracts the count of those without consecutive 1s to find those with consecutive 1s.
* **Dynamic Programming**: Calculates the number of binary strings without consecutive 1s for any given length N based on previously computed values.

This process ensures that the program efficiently calculates and outputs the desired result for any valid input length N.

**Problem Statement**

Mandy is working on a project that involves comparing binary representations of integers. She needs a program to determine how many bits need to be flipped to convert one integer to another. Help Mandy by writing a program that calculates and displays the number of bit flips required.

**Example**

**Input:** x= 10, y= 20

**Output:** 4

**Explanation:** The binary representation of x is 000**0101**0. The binary representation of y is 00010100. We need to flip the highlighted four bits in x to make it y.

**Input format :**

The first line of input consists of an integer **x**, representing the first number.

The second line consists of an integer **y**, representing the second number.

**Output format :**

The first line of output displays "x in binary is z" where z is the binary representation of x.

The second line displays "y in binary is A" where A is the binary representation of y.

The third line of output displays "The total number of bits to be flipped is " followed by an integer representing the number of bits that need to be changed to convert x to y.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ x, y ≤ 100

**Sample test cases :**

**Input 1 :**

10

20

**Output 1 :**

10 in binary is 1010

20 in binary is 10100

The total number of bits to be flipped is 4

**Input 2 :**

7

10

**Output 2 :**

7 in binary is 111

10 in binary is 1010

The total number of bits to be flipped is 3

**Solution:**

import java.util.Scanner;

class BitFlipCalculator {

    // Function to convert an integer to its binary representation as a string

    public static String toBinaryString(int number) {

        // Convert the integer to a binary string representation

        return Integer.toBinaryString(number);

    }

    // Function to count the number of bits set to 1 in the binary representation

    public static int countSetBits(int number) {

        int count = 0; // Initialize count of bits set to 1

        // Process each bit in the number

        while (number != 0) {

            // Add the least significant bit (1 or 0) to the count

            count += (number & 1);

            // Right shift the number to process the next bit

            number >>= 1;

        }

        // Return the total count of bits set to 1

        return count;

    }

    public static void main(String[] args) {

        // Create a Scanner object for user input

        Scanner scanner = new Scanner(System.in);

        // Read two integers x and y from user input

        int x = scanner.nextInt();

        int y = scanner.nextInt();

        // Get binary representations of x and y

        String binaryX = toBinaryString(x);

        String binaryY = toBinaryString(y);

        // Print the binary representations of the input integers

        System.out.println(x + " in binary is " + binaryX);

        System.out.println(y + " in binary is " + binaryY);

        // Calculate the XOR of x and y to find differing bits

        int xor = x ^ y;

        // Count the number of bits set to 1 in the XOR result

        int bitsToFlip = countSetBits(xor);

        // Print the number of bits that need to be flipped

        System.out.println("The total number of bits to be flipped is " + bitsToFlip);

    }

}

**Explanation of Each Part:**

1. **Imports and Class Declaration**:

java

Copy code

import java.util.Scanner;

class BitFlipCalculator {

* + Imports the Scanner class for reading user input.
  + Defines the BitFlipCalculator class.

1. **Binary Conversion Function**:

java

Copy code

public static String toBinaryString(int number) {

return Integer.toBinaryString(number);

}

* + Converts an integer to its binary string representation using Integer.toBinaryString().

1. **Counting Set Bits Function**:

java

Copy code

public static int countSetBits(int number) {

int count = 0; // Initialize count of bits set to 1

while (number != 0) {

count += (number & 1); // Add the least significant bit (1 or 0) to count

number >>= 1; // Right shift number to process the next bit

}

return count; // Return the total count of bits set to 1

}

* + Counts the number of bits set to 1 by checking each bit of the integer using bitwise operations.

1. **Main Method**:

java

Copy code

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in); // Create Scanner for user input

int x = scanner.nextInt(); // Read integer x

int y = scanner.nextInt(); // Read integer y

String binaryX = toBinaryString(x); // Get binary representation of x

String binaryY = toBinaryString(y); // Get binary representation of y

System.out.println(x + " in binary is " + binaryX); // Print binary representation of x

System.out.println(y + " in binary is " + binaryY); // Print binary representation of y

int xor = x ^ y; // Compute XOR of x and y to identify differing bits

int bitsToFlip = countSetBits(xor); // Count the number of bits that differ

System.out.println("The total number of bits to be flipped is " + bitsToFlip); // Output the result

}

* + Reads two integers from user input.
  + Converts these integers to binary strings and prints them.
  + Computes the XOR of the two integers to find differing bits.
  + Counts the bits set to 1 in the XOR result, which represents the number of bits that need to be flipped.
  + Prints the result.

This detailed commentary provides a clear understanding of what each part of the code does and how the entire program works to solve the problem.

**Example Walkthrough**

Let's use the number 29 as an example. The binary representation of 29 is 11101.

**Step-by-Step Execution**

1. **Initial State**:
   * number = 29 (binary: 11101)
   * count = 0
2. **First Iteration**:
   * number & 1 yields 1 (least significant bit is 1)
   * Add 1 to count: count = 1
   * Right shift number by 1 bit: number = 14 (binary: 1110)
3. **Second Iteration**:
   * number & 1 yields 0 (least significant bit is 0)
   * Add 0 to count: count = 1
   * Right shift number by 1 bit: number = 7 (binary: 111)
4. **Third Iteration**:
   * number & 1 yields 1 (least significant bit is 1)
   * Add 1 to count: count = 2
   * Right shift number by 1 bit: number = 3 (binary: 11)
5. **Fourth Iteration**:
   * number & 1 yields 1 (least significant bit is 1)
   * Add 1 to count: count = 3
   * Right shift number by 1 bit: number = 1 (binary: 1)
6. **Fifth Iteration**:
   * number & 1 yields 1 (least significant bit is 1)
   * Add 1 to count: count = 4
   * Right shift number by 1 bit: number = 0 (binary: 0)
7. **End of Loop**:
   * The while loop exits because number is now 0.
8. **Return Value**:
   * The function returns count, which is 4.

**Summary**

For the number 29:

* **Binary Representation**: 11101
* **Number of 1s (set bits)**: 4

**Problem Statement**

Sheldon, a computer science professor, is teaching his students about Gray code sequences. He wants a program to generate and display the Gray code sequence for a given number of bits.

Help Sheldon by writing a program that generates the Gray code sequence for n bits.

**Example**

**Input:**

2

**Output:**

Gray Code Sequence: 0 1 3 2

**Explanation:**

The binary representation of [0,1,3,2] is [00,01,11,10].

00 and 01 differ by one bit.

01 and 11 differ by one bit.

11 and 10 differ by one bit.

10 and 00 differ by one bit.

**Input format :**

The input consists of an integer **n**, representing the number of bits for the Gray code sequence.

**Output format :**

The output displays "Gray Code Sequence: " followed by a list of integers representing the Gray code sequence, where each integer is the decimal equivalent of the binary Gray code.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ n ≤ 6

**Sample test cases :**

**Input 1 :**

2

**Output 1 :**

Gray Code Sequence: 0 1 3 2

**Solution:**

import java.util.Scanner;

import java.util.ArrayList;

class GrayCodeGenerator {

    // Function to generate Gray code sequence for n bits

    public static ArrayList<Integer> generateGrayCode(int n) {

        // Create a list to store the Gray code sequence

        ArrayList<Integer> grayCodeSequence = new ArrayList<>();

        // Total number of Gray codes is 2^n

        int numCodes = 1 << n; // Equivalent to 2^n

        // Generate each Gray code

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

            // Calculate the Gray code for the integer i

            // Gray code is obtained by XORing i with i shifted right by 1 bit

            int grayCode = i ^ (i >> 1);

            // Add the Gray code to the list

            grayCodeSequence.add(grayCode);

        }

        // Return the list containing the Gray code sequence

        return grayCodeSequence;

    }

    public static void main(String[] args) {

        // Create a Scanner object to read input from the user

        Scanner scanner = new Scanner(System.in);

        // Read the number of bits for the Gray code sequence

        int n = scanner.nextInt();

        // Generate the Gray code sequence for n bits

        ArrayList<Integer> grayCodeSequence = generateGrayCode(n);

        // Print the generated Gray code sequence

        System.out.print("Gray Code Sequence: ");

        for (int i = 0; i < grayCodeSequence.size(); i++) {

            // Print each Gray code in the sequence

            System.out.print(grayCodeSequence.get(i));

            // Print a space between numbers, but not after the last number

            if (i < grayCodeSequence.size() - 1) {

                System.out.print(" ");

            }

        }

        // Move to the next line after printing the sequence

        System.out.println();

    }

}

Certainly! Let’s go through an example to understand how the generateGrayCode function works, focusing on generating the Gray code sequence for n = 3 bits.

**Example: Generating Gray Code Sequence for n = 3**

**Gray Code** is a binary numeral system where two successive values differ in only one bit. The sequence can be generated using the formula:  
GrayCode(i)=i⊕(i≫1)\text{GrayCode}(i) = i \oplus (i \gg 1)GrayCode(i)=i⊕(i≫1)  
where ⊕\oplus⊕ denotes the bitwise XOR operation and ≫\gg≫ denotes the bitwise right shift.

**Steps for Generating Gray Code Sequence**

1. **Initialize Variables:**
   * n = 3 (number of bits)
   * numCodes = 1 << n which is 23=82^3 = 823=8
2. **Loop Through All Values from 0 to 7:**

For each integer iii from 0 to 7, compute the Gray code as follows:

* + **i = 0**:
    - **Binary Representation**: 000
    - **Gray Code Calculation**: 0⊕(0≫1)=0⊕0=00 \oplus (0 \gg 1) = 0 \oplus 0 = 00⊕(0≫1)=0⊕0=0
    - **Gray Code**: 000
    - **Decimal Value**: 0
  + **i = 1**:
    - **Binary Representation**: 001
    - **Gray Code Calculation**: 1⊕(1≫1)=1⊕0=11 \oplus (1 \gg 1) = 1 \oplus 0 = 11⊕(1≫1)=1⊕0=1
    - **Gray Code**: 001
    - **Decimal Value**: 1
  + **i = 2**:
    - **Binary Representation**: 010
    - **Gray Code Calculation**: 2⊕(2≫1)=2⊕1=32 \oplus (2 \gg 1) = 2 \oplus 1 = 32⊕(2≫1)=2⊕1=3
    - **Gray Code**: 011
    - **Decimal Value**: 3
  + **i = 3**:
    - **Binary Representation**: 011
    - **Gray Code Calculation**: 3⊕(3≫1)=3⊕1=23 \oplus (3 \gg 1) = 3 \oplus 1 = 23⊕(3≫1)=3⊕1=2
    - **Gray Code**: 010
    - **Decimal Value**: 2
  + **i = 4**:
    - **Binary Representation**: 100
    - **Gray Code Calculation**: 4⊕(4≫1)=4⊕2=64 \oplus (4 \gg 1) = 4 \oplus 2 = 64⊕(4≫1)=4⊕2=6
    - **Gray Code**: 110
    - **Decimal Value**: 6
  + **i = 5**:
    - **Binary Representation**: 101
    - **Gray Code Calculation**: 5⊕(5≫1)=5⊕2=75 \oplus (5 \gg 1) = 5 \oplus 2 = 75⊕(5≫1)=5⊕2=7
    - **Gray Code**: 111
    - **Decimal Value**: 7
  + **i = 6**:
    - **Binary Representation**: 110
    - **Gray Code Calculation**: 6⊕(6≫1)=6⊕3=56 \oplus (6 \gg 1) = 6 \oplus 3 = 56⊕(6≫1)=6⊕3=5
    - **Gray Code**: 101
    - **Decimal Value**: 5
  + **i = 7**:
    - **Binary Representation**: 111
    - **Gray Code Calculation**: 7⊕(7≫1)=7⊕3=47 \oplus (7 \gg 1) = 7 \oplus 3 = 47⊕(7≫1)=7⊕3=4
    - **Gray Code**: 100
    - **Decimal Value**: 4

**Resulting Gray Code Sequence**

The sequence generated by the function for n = 3 bits is:

* 0 (binary 000)
* 1 (binary 001)
* 3 (binary 011)
* 2 (binary 010)
* 6 (binary 110)
* 7 (binary 111)
* 5 (binary 101)
* 4 (binary 100)

**How It Maps to the Function**

In the generateGrayCode function:

* **int numCodes = 1 << n** calculates 2^n to determine how many Gray codes to generate.
* **Loop** iterates from 0 to 7 (for n = 3), computes the Gray code for each value using i ^ (i >> 1), and adds it to grayCodeSequence.

**Problem Statement**

Saravanan, an aspiring mathematician, is fascinated by the concept of power sets and wants to explore all distinct subsets of a given set of integers including possible duplicates in the input set and the empty set. He needs a program that generates and displays all unique subsets of a given set in a sorted manner using bit manipulation.

Help Saravanan by writing a program to generate the power set.

**Note:** Use HashSet and ArrayList.

**Input format :**

The first line of input consists of an integer **n**, representing the number of elements in the set.

The second line contains **n** space-separated integers, representing the elements of the set.

**Output format :**

The output displays all unique subsets of the given set, each subset represented as a list of integers enclosed in square brackets [].

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ n ≤ 5

1 ≤ set elements ≤ 100

**Sample test cases :**

**Input 1 :**

3

1 2 1

**Output 1 :**

[[1], [], [1, 1], [2], [1, 1, 2], [1, 2]]

**Input 2 :**

4

1 2 3 4

**Output 2 :**

[[1], [], [2], [1, 2, 3], [1, 3, 4], [1, 2], [3], [2, 3], [1, 2, 4], [3, 4], [2, 3, 4], [1, 3], [4], [2, 4], [1, 4], [1, 2, 3, 4]]

**Input 3 :**

3

1 2 2

**Output 3 :**

[[1], [1, 2, 2], [], [2], [2, 2], [1, 2]]

**Solution:**

import java.util.\*;

public class Main {

    // Iterative function to print all distinct subsets of `S`

    public static void findPowerSet(int[] S) {

        // `N` stores the total number of subsets

        int N = (int)Math.pow(2, S.length);

        Set<List<Integer>> set = new HashSet<>();

        // sort the set

        Arrays.sort(S);

        // generate each subset one by one

        for (int i = 0; i < N; i++) {

            List<Integer> subset = new ArrayList<>();

            // check every bit of `i`

            for (int j = 0; j < S.length; j++) {

                // if j'th bit of `i` is set, append `S[j]` to the subset

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

                    subset.add(S[j]);

                }

            }

            // insert the subset into the set

            set.add(subset);

        }

        // print all subsets present in the set

        System.out.println(set);

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        int n = scanner.nextInt();

        int[] S = new int[n];

        for (int i = 0; i < n; i++) {

            S[i] = scanner.nextInt();

        }

        findPowerSet(S);

        scanner.close();

    }

}

**Problem Statement**

Enid Sinclair, a computer science student, is exploring the fascinating world of binary numbers and their concatenations. She wants to concatenate the binary representations of integers from 1 to n and compute the result modulo 109 +7.

Help Enid by writing a program that performs this task efficiently.

**Example**

**Input:** 3

**Output:** 27

**Explanation:** In binary, 1, 2, and 3 corresponds to "1", "10", and "11". After concatenating them, we have "11011", which corresponds to the decimal value 27.

**Input format :**

The input consists of an integer **n**, representing the upper limit of the range for concatenation.

**Output format :**

The output displays an integer representing the concatenated binary number modulo 109 +7.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ n ≤ 8

**Sample test cases :**

**Input 1 :**

3

**Output 1 :**

27

**Input 2 :**

5

**Output 2 :**

1765

**Solution:**

import java.util.Scanner;

class BinaryConcatenation {

    // Define a constant for the modulo value to prevent overflow

    private static final long MOD = 1000000007;

    /\*\*

     \* Function to calculate the decimal value of concatenated binary representations of numbers from 1 to n, modulo MOD.

     \*

     \* @param n The upper limit of numbers to concatenate.

     \* @return The decimal value of the concatenated binary string modulo MOD.

     \*/

    public static long calculateConcatenatedBinaryModulo(int n) {

        // Create a StringBuilder to efficiently concatenate binary strings

        StringBuilder binaryString = new StringBuilder();

        // Iterate through numbers from 1 to n

        for (int i = 1; i <= n; i++) {

            // Convert each number to its binary representation and append it to binaryString

            binaryString.append(Integer.toBinaryString(i));

        }

        // Convert the concatenated binary string to a decimal number

        long decimalValue = 0; // Initialize decimal value to 0

        // Convert each character of the binary string to its numeric value

        for (char c : binaryString.toString().toCharArray()) {

            // Update decimalValue by shifting left (multiply by 2) and adding the current bit value

            decimalValue = (decimalValue \* 2 + (c - '0')) % MOD;

        }

        // Return the result modulo MOD to ensure the value does not overflow

        return decimalValue;

    }

    public static void main(String[] args) {

        // Create a Scanner object to read input from the console

        Scanner scanner = new Scanner(System.in);

        // Read an integer value from the console, which represents the upper limit of the range

        int n = scanner.nextInt();

        // Call the function to calculate the concatenated binary modulo and print the result

        System.out.println(calculateConcatenatedBinaryModulo(n));

        // Close the scanner to free up resources

        scanner.close();

    }

}

**Detailed Explanation**

1. **Constant Declaration:**

java

Copy code

private static final long MOD = 1000000007;

* + MOD is a large prime number used to take results modulo this value. This helps in preventing integer overflow and keeping numbers manageable.

1. **calculateConcatenatedBinaryModulo(int n) Method:**

java

Copy code

public static long calculateConcatenatedBinaryModulo(int n) {

* + This method calculates the decimal value of concatenating binary representations of numbers from 1 to n, and returns the result modulo MOD.
  + **Concatenate Binary Representations:**

java

Copy code

StringBuilder binaryString = new StringBuilder();

for (int i = 1; i <= n; i++) {

binaryString.append(Integer.toBinaryString(i));

}

* + - A StringBuilder is used to build a single string representing the concatenation of binary representations of numbers from 1 to n.
  + **Convert Binary String to Decimal:**

java

Copy code

long decimalValue = 0;

for (char c : binaryString.toString().toCharArray()) {

decimalValue = (decimalValue \* 2 + (c - '0')) % MOD;

}

* + - The binary string is processed character by character.
    - decimalValue is updated by shifting the current value left (equivalent to multiplying by 2) and adding the value of the current bit (0 or 1).
    - % MOD ensures the result remains within the bounds of MOD to prevent overflow.
  + **Return Result:**

java

Copy code

return decimalValue;

1. **main(String[] args) Method:**

java

Copy code

public static void main(String[] args) {

* + The entry point of the program.
  + **Read Input:**

java

Copy code

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

* + - Reads the integer n from the console, which indicates the range for generating binary strings.
  + **Print Result:**

java

Copy code

System.out.println(calculateConcatenatedBinaryModulo(n));

* + **Close Scanner:**

java

Copy code

scanner.close();

* + - Closes the Scanner object to free up resources.

**Example**

For n = 3, the binary representations of numbers from 1 to 3 are:

* 1 → 1
* 2 → 10
* 3 → 11

Concatenating these gives the binary string 11011.

Converting 11011 to decimal:

* 1 \* 2^4 + 1 \* 2^3 + 0 \* 2^2 + 1 \* 2^1 + 1 \* 2^0 = 16 + 8 + 2 + 1 = 27

If the result were larger, modulo 1000000007 would be applied to ensure it fits within the specified range.

**Example**

Consider the binary string "11011":

1. Initialize decimalValue = 0.
2. Iterate over each character:
   * For c = '1':
     + decimalValue = (0 \* 2 + (1 - '0')) % MOD
     + decimalValue = (0 \* 2 + 1) % MOD
     + decimalValue = 1
   * For c = '1':
     + decimalValue = (1 \* 2 + (1 - '0')) % MOD
     + decimalValue = (2 + 1) % MOD
     + decimalValue = 3
   * For c = '0':
     + decimalValue = (3 \* 2 + (0 - '0')) % MOD
     + decimalValue = (6 + 0) % MOD
     + decimalValue = 6
   * For c = '1':
     + decimalValue = (6 \* 2 + (1 - '0')) % MOD
     + decimalValue = (12 + 1) % MOD
     + decimalValue = 13
   * For c = '1':
     + decimalValue = (13 \* 2 + (1 - '0')) % MOD
     + decimalValue = (26 + 1) % MOD
     + decimalValue = 27

The final result is 27, which is the decimal equivalent of the binary string "11011".