Please insert comment in below code

class B0052 {

    int count;

    public int **totalNQueens**(int n) {

        count = 0;

**backtrack**(0, 0, 0, 0, n);

        return count;

    }

    private void **backtrack**(int row, int ld, int col, int rd, int n) {

        if (row == (1 << n) - 1) {

            count++;

            return;

        }

        int pos = (~(row | ld | col | rd)) & ((1 << n) - 1);

        while (pos > 0) {

            int p = pos & -pos;

            pos -= p;

**backtrack**((row | p), (ld | p) << 1, (col | p), (rd | p) >> 1, n);

        }

    }

}

This code is a solution to the N-Queens problem, which is a classic problem in computer science and mathematics. The goal is to place N queens on an NxN chessboard such that no two queens threaten each other.

The class B0052 has an int variable 'count' which will keep track of the number of solutions found. The 'totalNQueens' function takes an int 'n' as an input, initializes the 'count' variable to zero, calls the 'backtrack' function and returns the 'count' variable at the end.

The 'backtrack' function takes five input parameters:

* 'row': a bitmask representing the rows where queens are placed.
* 'ld': a bitmask representing the diagonals from top-left to bottom-right where queens are placed.
* 'col': a bitmask representing the columns where queens are placed.
* 'rd': a bitmask representing the diagonals from top-right to bottom-left where queens are placed.
* 'n': the size of the chessboard.

The function uses a backtracking algorithm to find all possible solutions. If the number of rows where queens are placed is equal to (1 << n) - 1, then a solution has been found and the 'count' variable is incremented. Otherwise, the function generates a bitmask 'pos' representing all the empty positions on the chessboard where a queen can be placed.

It then goes through a loop where the function uses bit manipulation to place a queen at the lowest set bit in 'pos' and then recursively calls itself with the updated bitmasks for the rows, diagonals and columns.

The function continues until it finds all possible solutions and the 'count' variable will have the total number of solutions found.

what is 1 << n

**1 << n** is a bitwise left shift operator. It shifts the bits of the number 1 to the left by n places, effectively multiplying it by 2^n.

For example, **1 << 2** would be equivalent to 1 \* 2^2 = 4.

Why you need to use private void in java

In Java, the **private** access modifier is used to restrict access to a class, method, or variable to only within the same class.

When a method or variable is declared as private, it can only be accessed or called from within the same class. This means that other classes or objects cannot directly access or modify the private method or variable.

This is useful for encapsulation, which is a fundamental concept in object-oriented programming. By keeping the implementation of a class or method private, you can hide the details of how it works from other classes or objects that use it. This allows you to change the implementation without affecting other parts of your code, and it also helps to prevent other code from making invalid or unintended changes to the class or method.

Additionally, declaring a method as private void, means it is a private method with no return type. It can be used for internal method calls or for certain functionalities which are not supposed to be exposed to the outer world.

Explain to code line by line Code that need to be explain :

if (row == (1 << n) - 1) { // increment count, a solution is found count++; return; }

The code is taken from this :

class B0052 {

    // variable to keep track of number of solutions found

    int count;

    public int **totalNQueens**(int n) {

        // initialize count to 0

        count = 0;

        // call backtrack function with initial bitmasks

**backtrack**(0, 0, 0, 0, n);

        // return the number of solutions found

        return count;

    }

    private void **backtrack**(int row, int ld, int col, int rd, int n) {

        // if all rows are filled

        if (row == (1 << n) - 1) {

            // increment count, a solution is found

            count++;

            return;

        }

        // generate bitmask for empty positions

        int pos = (~(row | ld | col | rd)) & ((1 << n) - 1);

        while (pos > 0) {

            // get lowest set bit in pos

            int p = pos & -pos;

            // mark this position as filled

            pos -= p;

            // recursive call with updated bitmasks

**backtrack**((row | p), (ld | p) << 1, (col | p), (rd | p) >> 1, n);

        }

    }

}

The code is checking if all rows of the chessboard have been filled.

1. **if (row == (1 << n) - 1)**: Here, the variable 'row' is being compared to the value of **(1 << n) - 1**. The **1 << n** is a bitwise left shift operator that shifts the bits of the number 1 to the left by n places, effectively multiplying it by 2^n. In this case, it is used to generate a bitmask of length 'n' with all bits set to 1. Subtracting 1 from this value gives a bitmask with n-1 bits set to 1.
2. **// increment count, a solution is found**: This is a comment describing that a solution has been found, and the count of solutions will be incremented.
3. **count++;**: This increments the 'count' variable by 1, which is keeping track of the number of solutions found.
4. **return;**: This returns control back to the calling function.

In summary, this code checks if all rows of the chessboard have been filled by comparing the value of 'row' to a bitmask of all 1's. If it is true, it increments the 'count' variable, which keeps track of the number of solutions found and returns control back to the calling function.

1 << 4 mean?

**1 << 4** is a bitwise left shift operator. It shifts the bits of the number 1 to the left by 4 places, effectively multiplying it by 2^4. In binary representation, 1 is represented as '0001' and by left shifting by 4 places, it becomes '00010000' which is 16 in decimal representation. So, 1 << 4 is equal to 16. This is equivalent to the mathematical expression 1 \* 2^4 = 16. It is commonly used in programming when working with binary numbers and bit manipulation.

(1 << 4) - 1 mean?

**(1 << 4) - 1** is a bitwise left shift operator followed by a subtraction of 1. It first shifts the bits of the number 1 to the left by 4 places, effectively multiplying it by 2^4. In binary representation, 1 is represented as '0001' and by left shifting by 4 places, it becomes '00010000' which is 16 in decimal representation. Then subtracting 1 from the value, it becomes 15. So, (1 << 4) - 1 is equal to 15. This is equivalent to the mathematical expression (1 \* 2^4) - 1 = 15. It is commonly used in programming when working with binary numbers and bit manipulation. It is used to generate a bitmask of length 4 with all bits set to 1 except the first one.

Why you need to the this code?

This code : (~(row | ld | col | rd))

The code is taken from this :

class B0052 {

    // variable to keep track of number of solutions found

    int count;

    public int **totalNQueens**(int n) {

        // initialize count to 0

        count = 0;

        // call backtrack function with initial bitmasks

**backtrack**(0, 0, 0, 0, n);

        // return the number of solutions found

        return count;

    }

    private void **backtrack**(int row, int ld, int col, int rd, int n) {

        // if all rows are filled

        if (row == (1 << n) - 1) {

            // increment count, a solution is found

            count++;

            return;

        }

        // generate bitmask for empty positions

        int pos = (~(row | ld | col | rd)) & ((1 << n) - 1);

        while (pos > 0) {

            // get lowest set bit in pos

            int p = pos & -pos;

            // mark this position as filled

            pos -= p;

            // recursive call with updated bitmasks

**backtrack**((row | p), (ld | p) << 1, (col | p), (rd | p) >> 1, n);

        }

    }

}

The code is used to solve the n-queens problem, which is a classic problem in computer science that involves placing n queens on an nxn chessboard in such a way that no two queens threaten each other. The goal is to find all possible ways to place n queens on an nxn chessboard such that no two queens threaten each other.

The code uses bitmasks and backtracking to efficiently find all solutions to the problem.

The variable "pos" is a bitmask representing the empty positions on the chessboard where a queen can be placed.

The bitwise negation operator (~) is used to invert the bits of the bitmask, effectively finding all the empty positions on the chessboard.

The bitwise OR operator (|) is used to combine the bitmasks for the occupied positions on the chessboard (stored in the variables "row", "ld", "col", and "rd").

The bitwise AND operator (&) is then used to find the empty positions by taking the intersection of the inverted bitmask and the total number of positions on the chessboard.

This is used to find the next available position to place a queen on the chessboard during the backtracking process.

Explain to me below code :

private void **backtrack**(int row, int ld, int col, int rd, int n) {

        // if all rows are filled

        if (row == (1 << n) - 1) {

            // increment count, a solution is found

            count++;

            return;

        }

        // generate bitmask for empty positions

        int pos = (~(row | ld | col | rd)) & ((1 << n) - 1);

        while (pos > 0) {

            // get lowest set bit in pos

            int p = pos & -pos;

            // mark this position as filled

            pos -= p;

            // recursive call with updated bitmasks

**backtrack**((row | p), (ld | p) << 1, (col | p), (rd | p) >> 1, n);

        }

    }

This code is a backtracking algorithm for solving the N-Queens problem, which is the problem of placing N queens on an NxN chessboard such that no two queens threaten each other. The function "backtrack" is used to search for solutions by exploring all possible ways of placing queens on the board.

The function takes in five parameters:

* "row" is a bitmask representing which rows have queens placed on them.
* "ld" is a bitmask representing which positions on the left diagonal have queens placed on them.
* "col" is a bitmask representing which columns have queens placed on them.
* "rd" is a bitmask representing which positions on the right diagonal have queens placed on them.
* "n" is the size of the chessboard (number of queens).

The function first checks if all rows have been filled with queens (row == (1 << n) - 1), if so, it increments the count and returns.

Otherwise, it generates a bitmask for the empty positions on the chessboard using the bitwise OR operator (~) and the AND operator (&). The bitmask is then stored in the variable pos.

It then enters a loop and starts processing the lowest set bit in pos. It marks the position as filled and then makes a recursive call to the function with updated bitmasks for row, ld, col, and rd. This continues until all possible solutions have been found or the empty positions bitmask becomes zero.