Please explain the following code so I may use it in an interview.

public class B0050 {

    public double **myPow**(double x, int n) {

        if(n < 0){

            n = -n;

            x = 1 / x;

        }

        double pow = 1;

        while(n != 0){

            if((n & 1) != 0){

                pow \*= x;

            }

            x \*= x;

            n >>>= 1;

        }

        return pow;

    }

}

This code defines a class **B0050** which has a single method, **myPow**. The method takes two arguments, a double **x** and an integer **n**, and returns a double.

The method first checks if **n** is less than 0, and if so, sets **n** to its absolute value and sets **x** to the reciprocal of **x** (i.e. 1/x).

A variable **pow** is initialized to 1. The method then enters a while loop that continues as long as **n** is not equal to 0. Within the loop, if the binary AND of **n** and 1 is not equal to 0 (i.e. if the least significant bit of **n** is 1), **pow** is multiplied by **x**.

Then, **x** is multiplied by itself and **n** is assigned the result of a bitwise unsigned right shift by 1 (**>>>= 1**)

Finally, the method returns **pow**

This code is using bit manipulation and exponentiation by squaring to compute x^n in O(log n) time. It is more efficient than a simple for loop or recursion.

What is absolute value and reciprocal of x?

The absolute value of a number is its distance from 0 on the number line, regardless of whether the number is positive or negative. For example, the absolute value of -5 is 5, and the absolute value of 5 is also 5.

Reciprocal of x, is the multiplicative inverse of x, which is 1/x. For example, the reciprocal of 2 is 1/2, and the reciprocal of 1/3 is 3. In the code provided, when n<0, x is set to the reciprocal of x, this is done in order to calculate the power of x when the exponent is negative.

Explain to me like I am 12 about absolute value

Okay, imagine a number line, like the one you might see in a math book, with negative numbers on the left, zero in the middle, and positive numbers on the right. The absolute value of a number is how far that number is from zero, no matter whether it's on the left side or the right side of zero.

For example, if the number is -5, it is 5 spaces away from zero on the left side, so the absolute value of -5 is 5. If the number is 5, it is also 5 spaces away from zero on the right side. So, no matter if the number is negative or positive, the absolute value is always positive.

Does that make sense?

If the absolute value is always positive then why n = -n ?

You're correct that the absolute value of a number is always positive. The line **n = -n** is used to change the sign of **n** from negative to positive when **n** is less than zero. This is done because the rest of the code is written to calculate the power of x when the exponent is positive, So, to use the same code for negative exponents, the sign of the exponent must be flipped to positive before proceeding. The result is then multiplied by 1/x, which is equivalent to taking the reciprocal of x to get the correct output for negative exponents.

Why you need to use this code

x \*= x;

n >>>= 1;

The code **x \*= x;** is used to square the value of **x**. This is important because it allows the method to use a technique called "exponentiation by squaring" to calculate the power of **x** to the exponent **n**.

Exponentiation by squaring is a way to calculate the power of a number more efficiently than by repeatedly multiplying the number by itself. The basic idea is that if you need to calculate **x^n**, instead of multiplying **x** by itself **n** times, you can repeatedly square **x**, and at each step, if the current exponent is odd, you also multiply by **x** one more time.

The line **n >>>= 1;** is used to shift the bits of **n** to the right by 1 bit. This is the same as dividing **n** by 2, but it's more efficient to do it this way. This is done to keep track of the exponent after each step of the while loop, since we are doing the exponentiation by squaring, in each step we are dividing the exponent by 2.

By using both of these operations together, the method is able to calculate the power of **x** to the exponent **n** more efficiently than by using a simple loop or recursion, by reducing the number of multiplications that need to be done.

Why you need to use this code

if((n & 1) != 0){ pow \*= x; }

The line **if((n & 1) != 0)** is used to check if the least significant bit of **n** is 1. The **&** operator is a bitwise AND operator, which compares each bit of the first operand (**n**) to the corresponding bit of the second operand (1) and returns 1 if both bits are 1, and 0 otherwise. So, **n & 1** will be 1 if the least significant bit of **n** is 1, and 0 otherwise. Then, the != 0 check is used to check if the result of n & 1 is not equal to zero, which is true if the least significant bit of n is 1.

If the least significant bit of **n** is 1, this means that the current value of **n** is odd, and that **x** should be multiplied by **pow**. The line **pow \*= x;** is used to do this. This is done because of the exponentiation by squaring, as I mentioned before, we are dividing the exponent by 2 in each step, by checking if the exponent is odd or even, we know if we need to multiply the current value of pow by x or not.

So, this line of code is used to check if the current exponent is odd, and if it is, it multiplies the **pow** by **x** one more time.

In summary, this part of the code is used to check if the current exponent is odd, and if it is, it multiplies the **pow** by **x** one more time, which is needed to correctly calculate the power of x to the exponent n using the exponentiation by squaring.

Explain to me “least significant bit of n is 1” in below code especially

“if((n & 1) != 0){ pow \*= x; }”

public class A0050 { public static void main(String[] args) {

B0050 b0050 = new B0050();

double x = 2.0;

int n = 10;

double result = b0050.myPow(x, n);

System.out.println(result); } }

The code is calculating the value of x raised to the power of n. The if statement "if((n & 1) != 0)" is checking if the least significant bit (LSB) of the variable "n" is 1 or not.

"n & 1" performs a bitwise AND operation on the binary representation of "n" and the number 1. The binary representation of 1 is "0001" and if the LSB of "n" is 1, the result of this operation will also be 1. If the LSB is 0, the result will be 0.

The inequality operator "!= 0" is checking if the result of the bitwise AND operation is not equal to 0. If it is not equal to 0, then it means the LSB of "n" is 1 and the code inside the if statement gets executed. The line "pow \*= x" is multiplying the current value of pow by x.

So the if statement is checking if the LSB is 1 and multiply the pow by x if it is 1.

Explain to me like I am 12

The code is trying to figure out what happens when you multiply a number by itself a certain number of times. Like if you have 2 \* 2 \* 2 \* 2 \* 2 \* 2 \* 2 \* 2 \* 2 \* 2 = 1024. It's checking if the last digit of the number of times it needs to multiply is a 1 or not. If it's a 1, it will multiply the number one more time. If it's not, it won't. It's like checking if the last number of a number of groups of 10 is a 1. Like 21 groups of 10 is 210, and the last digit is 1, so it will add one more group of 10 to it. But 20 groups of 10 is 200, and the last digit is 0, so it won't add any more groups.

What do you mean by below statemennt?

"n & 1" performs a bitwise AND operation on the binary representation of "n" and the number 1. The binary representation of 1 is "0001" and if the LSB of "n" is 1, the result of this operation will also be 1. If the LSB is 0, the result will be 0.

A bitwise operation is a way for a computer to process numbers by looking at each individual "bit" (or binary digit) of the number.

The binary representation of a number is a way of writing a number using only 0s and 1s. For example, the binary representation of the number 10 is 1010.

The bitwise AND operation takes two numbers and compares each bit of the numbers. If both bits are 1, the resulting bit is 1. If either bit is 0, the resulting bit is 0.

So in this case, the "n & 1" is taking the binary representation of the number 'n' and the number 1 (which is always 0001), and doing the bitwise AND operation. If the LSB of 'n' is 1, the result will be 1, otherwise it will be 0. This is used to check if the LSB of 'n' is 1 or 0.

So what is bit for number 5 ?

The binary representation of the decimal number 5 is "00000101" in 8-bit. Each bit in a binary number can either be a 0 or a 1. The rightmost bit (the bit farthest to the right) is known as the least significant bit (LSB) and the leftmost bit is the most significant bit (MSB). In the case of the number 5, the LSB is 1 and the rest of the bits are 0.

Also, you can express the number 5 in 4-bit binary representation, which is "0101" , LSB is still 1.

Why using n >>>= 1 ? Why the code didn’t use n =/ 2 ?

The statement "n >>>= 1" is a shorthand way of writing "n = n >>> 1". It is a bitwise operator that shifts the bits of the number to the right by 1 position, discarding the rightmost bit and adding a 0 to the left. This effectively divides the number by 2, discarding any remainder.

On the other hand, the statement "n =/ 2" is not a valid statement in Java, it should be "n = n / 2" which also divide the number by 2 but the difference between using "n >>>= 1" and "n = n / 2" is that

* the first one doesn't take into account the sign of the number, it always shifts the bits to the right discarding the rightmost bit,
* but the second one will consider the sign of the number, if the number is negative it will round towards 0.

Using "n >>>= 1" in this context will ensure that the number is always divided by 2 without considering the sign of the number, making the operation faster and more efficient.

Explain to me how 10 can be divided in half to yield 5. Explain to me in bitwise operation

In order to understand how the number 10 can be divided by 2 using bitwise operations, it is first important to understand how the number 10 is represented in binary. The binary representation of the decimal number 10 is "00001010" in 8-bit.

To divide a number by 2 using bitwise operations, we can shift all of its bits to the right by 1 position. This is equivalent to dividing the number by 2 and discarding any remainder. The right shift operator ">>" or ">>> " can be used to accomplish this. In this case, the ">>" operator is used, which keeps the sign of the number when shifting.

When we apply the operator ">>" on the binary representation of 10 "00001010", we get "00000101" which is the binary representation of 5.

To be more clear, we can see the following:

10 in decimal = 1010 in binary

shift the bits one step to the right: 0101 in binary

0101 in binary = 5 in decimal

So by shifting the bits to the right by 1 position, we effectively divide the number by 2 and get the result of 5.