Level: Easy

Power Function: Implement a function to calculate X^N. Both X and N can be positive or negative. You can assume that overflow doesn't happen.

(Try doing it in *O(log(N))* time)

For example:

```
 2 ^ 2  = 4
 2 ^ -2 = 0.25
-2 ^ 3  = -8
```

Questions to Clarify:
Q. Are both X and N integers?
A. Yes

Q. Can the number be both negative and positive?
A. Yes

Q. Can the power be negative?
A. Yes

Q. Can we assume that the result won't overflow?
A. Yes, you can assume that.

Q. 0^0 or 0^{-3} (or any negative number) is undefined. How do we return that?
A. Throw an exception.

Solution:
In this problem, there are two portions:

1. Figuring out all the cases - how to handle negative signs for both X and N.
2. Finding an algorithm to calculate powers.

You can calculate the result (X^N) for absolute values of X and N, and modify that to handle negative signs. Let's say this result is called positivePower.

$-2^3 = 2^3 * -1$ *(power is an odd number)*
$-2^2 = 2^2$ *(power is an even number)*

If X is negative, return -1 * positivePower if the power (N) is odd.
Otherwise just return positivePower.

$2^{-3} = 1 / 2^3$

If the power (N) is negative, just return 1 / positivePower.

That takes care of the signs. Now, we just need to calculate positive powers, i.e., both X and N are positive.

To do this, the O(N) approach is obvious. Just loop N times and multiply X.

However, we can do better. We can divide our work by 2 each time.

$X^N = X^{N/2} * X^{N/2}$ (if N is even)

If N is odd, we modify it a bit:

$X^N = X^{N/2} * X^{N/2} * X$ (if N is odd)

Remember, in most languages, if N is an odd integer (e.g, 5), N/2 will be floor of the half:

=> 5/2 = 2, not 2.5 or 3

If we do this recursively, we can do it in $O(\log_2 N)$ steps. For example:

$2^{16} = 2^8 * 2^8$
$2^8 = 2^4 * 2^4$
$2^4 = 2^2 * 2^2$
$2^2 = 2^1 * 2^1$

That's 4 steps (which is also $\log_2 16$).

Note: Here, we did not use memoization, because we are creating only one branch per function call and we're not repeating any call.

Memoization is only useful when we're repeating the same work again.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Except if you're writing
a few lines to quickly plan your solution. Your actual solution should be in a
real language and use good syntax.)

power(X, power)
     if X is 0 and power <= 0
         undefined, throw error

     result = positivePower(absolute_value(X), absolute_value(power))

     if power is -ve: result = 1 / result
     if X is -ve: result = -result

     return result

positivePower(X, power):
     // base cases
     if power is 0, return 1
     if power is 1, return X

     halfPower = X ^ power/2
     if power is even:
         return halfPower * halfPower
     else:
         return X * halfPower * halfPower
```

Test Cases:

Base Cases: power is 0, power is 1, x is 0, x is 1

Regular Cases: both power & x are +ve, both are -ve, both different signs

Time Complexity: O(log(N))

Space Complexity: O(log(N)) on call stack

```
public static float power(int x, int power) {
    if (x == 0 && power <= 0) {
        throw new ArithmeticException("undefined");
    }

    float result = positivePower(Math.abs(x), Math.abs(power));

    // handle negative power
    if (power < 0)
        result = 1 / result;
```

```
    // handle negative x
    if (x < 0 && power % 2 != 0)
        result = -1 * result;

    return result;
}

public static int positivePower(int x, int power) {
    if (power == 0)
        return 1;
    if (power == 1)
        return x;

    int halfPower = positivePower(x, power/2);
    if (power % 2 == 0) {
        return halfPower * halfPower;
    } else {
        return x * halfPower * halfPower;
    }
}
```