Modular Exponentiation

Exponentiation is a mathematical operation that is defined as a^b, where a is known as the *base* and b is the *exponent*. So, a^b = a.a.a....a (b times).

Finding the value of a^b can be easily done recursively as $a*a^{b-1}$. However, the number of steps it takes to find the value is proportional to the value of exponent b. So, if the value of b is quite large like 1e18, then the method is not suitable to find the value of a^b.

Instead, we can use what we call **binary exponentiation**, where we break down the exponents by dividing them by 2.

- If b is even, then a^b can be written as: $(a^{b/2})*(a^{b/2})$, where we are only required to compute the value of $a^{b/2}$.
- If b is odd, the a^b can be written as: $a*a^{b-1}$, now b-1 is even and you can simply apply the above case to find the value of a^{b-1}.

Since at every step b is divided by 2, the time complexity becomes $O(\log_2 b)$, which is fast enough to calculate the values of a^b, for large exponents(b).

Pseudocode (Recursive):

```
/* Input a and b are non-negative integers , returns a^b */
function binaryExpo(a,b)

    //  Base Case
    if a equals 0
        return 0
    if b equals 0
        return 1

    //  If b is even
    if b mod 2 equals 0
        res = binaryExpo(a,b/2)
        return res * res
    //  Else if b is odd
    else
        res = binaryExpo(a,(b - 1)/2)
        return res * res * a
```

Time complexity: $O(\log_2 b)$, where b is the exponent

Pseudocode (Iterative):

```
/* Input a and b are non-negative integers, returns a^b */
function binaryExpo(a,b)
```

```
if a equals 0
        return 0

//   Initialize the value of result to 1
res = 1
//   Iterate until the exponent is positive
while(b > 0)
        //   If b is odd
        if b mod 2 equals 1
                res = res * a
        a = a * a
        b = b / 2

return res
```

Time complexity: O($\log_2 b$), where b is the exponent

However, several cases may arise when a(base) and b(exponent) are quite large enough that the result can no longer be stored within the permissible limits of the values of the data type, or in other cases multiplying huge numbers may lead to taking longer times for execution.
So, we generally use the modulo operation (%M), to find the answer to computing huge powers such that the value remains within the limits of the data type and also does not lead to the multiplication of huge numbers.

For example, let’s consider 2^{1024}, the answer of this can be large enough that it will not fit any integer or long-range. So we calculate 2^{1024} % M, where M is some integer so that the answer will lie in the range 0 to M-1.

Pseudocode (Recursive):

```
/* Input a,b and are non-negative integers and m is the modulo, returns $a^b$ mod m*/
function modularExpo(a,b,m)

        //   Base Case
        if a equals 0
                return 0
        if b equals 0
                return 1

        //   If b is even
        if b mod 2 equals 0
                res = modularExpo(a,b/2) mod m
                return (res * res) mod m
        //   Else if b is odd
        else
                res = modularExpo(a,(b - 1)/2) mod m
```

```
                    return ((a mod m) * (res * res) mod m) mod m
```

Time complexity: O($\log_2 b$), where b is the exponent

Pseudocode (Iterative):

```
/* Input a and b are non-negative integers and m is the modulo,  returns a^b mod m*/
function binaryExpo(a,b,m)

          //   If a is greater than or equal to m, update a
          a = a mod m
          if a equals 0
                    return 0

          //   Initialize the value of result to 1
          res = 1
          //   Iterate until the exponent is positive
          while(b > 0)
                    //   If b is odd
                    if b mod 2 equals 1
                              res = (res * a) mod m
                    a = (a * a) mod m
                    b = b / 2

          return res
```

Time complexity: O($\log_2 b$), where b is the exponent