Modular exponentiation

X^N%M needs to be found.

BRUTE FORCE :

Mutliply X n times modulo M and return ans. The value can be significatly large for power so we multiply by 1LL for number to be in range.

```
#include <bits/stdc++.h>

int modularExponentiation(int x, int n, int m) {
  // Write your code here.
    int ans=1;
    for(int i=1;i<=n;i++)
    {
        ans=(1LL*ans*x)%m;
    }
    return ans%m;
}
```

- Time Complexity : O(n)
- Space Complexity : O(1)

Optimal Approach : Recursive

2^5 can be rewritten as 2*2^4 —> 2*(2)^2*2 which is 2*(2*2)^2—> 2*(4*4)^1 so from this we know that if we can recursively divide the power by 2 and keep multiplying ans then the complexity can be reduced to O(logN).

So we store the recursive calls in var ans after dividing power by 2.Whenever we multiply by ans or x we take care for overflow and multiply by 1LL and %m in each step where the value can overflow.

We have two paths to follow now :

1. when n is even then we multiply ans * ans
2. Else when n is odd then we also need to multiply ans*ans %m *x

```
#include <bits/stdc++.h>

int modularExponentiation(int x, int n, int m) {
  // Write your code here.
    if(n==0)
        return 1;
    int ans=modularExponentiation(x,n/2,m);
    if(n%2==0)
    {
        return (1LL*ans*ans)%m;
    }
    return (1LL*(1LL*ans*ans)%m*x%m)%m;
}
```

- Time Complexity : O(log N)
- Space Complexity : O(logN)

Optimal Approach : Binary Exponentiation (iterative)

Whenever n is odd we multiply x into ans otherwise we keep multiplying x with x and keep storing it. We chcek for even odd using bit manipulation we perform bitwise and with n and then keep shifting n by 1 bit to right to check for next set bit ex: 10 can be written in binary as 1010 firstly n&1 gives 0 so we just multiply x*x and store in x then we shift n so it becomes 101 now ans=ans*x then x=x*x everytime we mutiply x and store in x but we store ans*x only when nth bit is set.

```
#include <bits/stdc++.h>

int modularExponentiation(int x, int n, int m) {
  // Write your code here.
    int ans=1;
    while(n>0)
    {
        if(n&1)
        {
            ans=(1LL*ans*x)%m;
        }
        x=(1LL*x*x)%m;
        n>>=1;
    }
    return ans;
}
```

- Time Complexity : O(logN) since operating with bits and shifting bits reduces it to logN
- Space Complexity : O(1)