PROBLEM STATEMENT (COMMON BASE)

We want to compute:

$$x^n \text{or} a^b \, mod \, m$$

Where:

- n / b can be **very large**
- n can be **negative**
- Multiplication can **overflow**
- Sometimes b is given as a **vector of digits**

1️⃣ Naive Power Calculation

✅ Concept

The most straightforward idea is:

$$x^n = x \times x \times x \dots (n \text{ times})$$

If n is negative:

$$x^{-n} = \frac{1}{x^n}$$

💻 Code (Naive Approach)

```
double solve(int n, double x)
{
   double ans = 1.0;

   if (n < 0) {
      n = -n;
      x = 1 / x;
   }

   while (n--) {
      ans *= x;
```

```
    }
    return ans;
}
```

❌ Drawbacks

- ⏱️ **Time Complexity:** O(n)
- ❌ Too slow for large n
- ❌ Fails for n ≈ 10^9
- ❌ Not accepted in competitive programming

➡️ Need a faster approach

2️⃣ Binary Exponentiation (Recursive)

✅ Concept

Use the mathematical identity:

$$x^n = \begin{cases} (x^2)^{n/2}, & n \text{ even} \\ x \cdot x^{n-1}, & n \text{ odd} \end{cases}$$

This **cuts the problem in half** each time.

💻 Code (Recursive Binary Exponentiation)

```
double solve(double x, int n)
{
    if (n == 0) return 1;

    if (n < 0)
        return solve(1 / x, -n);

    if (n % 2 == 0)
        return solve(x * x, n / 2);
    else
```

```
    return x * solve(x, n - 1);
}
```

❌ **Drawbacks**

- 🧠 Uses recursion → **stack space**
- ⚠️ Risk of stack overflow for deep recursion
- ❌ Not optimal in memory usage

➡️ **We want the same speed but without recursion**

3️⃣ **Binary Exponentiation (Iterative – Optimal)**

✅ **Concept**

Convert recursion into iteration.

Key idea:

- Process exponent **bit by bit**
- Square the base
- Halve the exponent

💻 **Code (Iterative Binary Exponentiation)**

```
long long binexp(long long a, long long b, long long m)
{
  long long ans = 1;
  a %= m;

  while (b > 0)
  {
    if (b & 1)
      ans = (ans * a) % m;

    a = (a * a) % m;
```

```
    b >>= 1;
  }
  return ans;
}
```

❌ Drawbacks

- ❌ a * a may overflow for very large numbers
- ❌ Unsafe when a ≈ 10^18

➡️ We need overflow-safe multiplication

🔹 Binary Multiplication (Overflow-Safe)

✅ Concept

Replace multiplication with **repeated addition**:

$$a \times b = \sum a \times 2^i$$

This avoids overflow completely.

💻 Code (Binary Multiplication)

```
long long binmul(long long a, long long b, long long m)
{
  long long ans = 0;
  a %= m;

  while (b > 0)
  {
    if (b & 1)
      ans = (ans + a) % m;

    a = (a + a) % m;
    b >>= 1;
```

```
    }
    return ans;
}
```

❌ Drawbacks

- Slower than normal multiplication
- Used only when overflow risk exists

➡️ Combine safe multiplication with fast exponentiation

5️⃣ Binary Exponentiation with Safe Multiplication

✅ Concept

Use:

- Binary exponentiation
- Binary multiplication instead of *

💻 Code (Fully Safe Power)

```
long long binexp(long long a, long long b, long long m)
{
    long long ans = 1;
    a %= m;

    while (b > 0)
    {
        if (b & 1)
            ans = binmul(ans, a, m);

        a = binmul(a, a, m);
        b >>= 1;
    }
    return ans;
```

}

❌ **Drawbacks**

- Slightly slower than normal binary exponentiation
- Still cannot handle **huge exponent given as digits**

➡️ **Exponent itself is too large now**

6️⃣ **Euler's Totient Theorem (Reducing Large Exponent)**

✅ **Concept**

If:

$$\gcd(a, m) = 1$$

Then:

$$a^{\varphi(m)} \equiv 1 (\mathrm{mod} m)$$

So:

$$a^b \, mod \, m = a^{(b \, mod \, \varphi(m))} \, mod \, m$$

For:

1337 = 7 × 191

ϕ(1337) = 1140

💻 **Code (SuperPow with Euler Theorem)**

```
class Solution {
public:
    int solve(int a, int b, int m)
    {
        long long ans = 1;
        a %= m;

        while (b > 0)
```

```
    {
        if (b & 1)
            ans = (ans * a) % m;

        a = (a * a) % m;
        b >>= 1;
    }
    return ans;
}

int superPow(int a, vector<int>& b)
{
    int bmod = 0;
    for (int digit : b)
        bmod = (bmod * 10 + digit) % 1140;

    if (bmod == 0)
        bmod = 1140;

    return solve(a, bmod, 1337);
}
};
```

❌ **Drawbacks (VERY IMPORTANT ⚠️)**

- ❌ **Fails when gcd(a, m) ≠ 1**
- ❌ Example: a = 7, m = 1337
- ❌ Euler theorem becomes invalid

➡️ **Final universally correct approach is digit-by-digit recursion**

🔚 **Final Summary Table**

Approach	Time	Space	Safe	Notes
Naive	O(n)	O(1)	❌	Too slow
Recursive Binary	O(log n)	O(log n)	❌	Stack use
Iterative Binary	O(log n)	O(1)	❌	Overflow risk
Safe Binary	O(log n)	O(1)	✅	Large numbers
Euler SuperPow	O(len(b))	O(1)	⚠️	Needs gcd = 1