🔥 Modular Exponentiation Master Guide 🎯

🎯 Problem Statement

> Given integers `x`, `n`, and `M`, compute **(x^n mod M)** efficiently.

⚡ Why Not Just Use `pow()` or `x^n`?

Imagine raising `x = 5` to power `n = 1e18` 😱. The number becomes *astronomically huge* and will definitely **overflow** even `long long int`. Hence, we need smarter approaches!

🧠 Analogy Time: "📦 Folding Paper"

Imagine you're folding a large paper (exponent) repeatedly:

- Every fold (÷2) makes it smaller!
- Use the folds (binary form) to remember *when* to multiply! 💡

☑️ Brute Force Approach (❌ Not Recommended)

```
long long int PowMod(long long x, long long n, long long M) {
    long long int ans = 1;
    for (long long i = 0; i < n; i++) {
        ans = (ans * x) % M;
    }
    return ans;
}
```

⏱️ Time Complexity: O(n)

📦 Space Complexity: O(1)

> ❌ **Not feasible for large** `n` like 1e18. Too slow!

☑️ Efficient Approach: **Binary Exponentiation** (a.k.a. Fast Power)

> We exploit the fact that:

```
x^n = x^(n/2) * x^(n/2)     if n is even
x^n = x * x^(n/2) * x^(n/2) if n is odd
```

💡 Intuition

- Convert n to **binary** 🔢
- Use each bit to decide whether to multiply
- Reduce n by half at each step

🧪 Dry Run Example

```
x = 3, n = 5, M = 100
Binary of 5 = 101
Step-by-step:
- ans = 1
- bit 0 (1): ans = ans * 3 = 3
- bit 1 (0): skip multiply
- bit 2 (1): ans = ans * 81 = 243
Final ans = 243 % 100 = 43
```

🧑‍💻 Final C++ Code

```cpp
class Solution {
public:
  long long int PowMod(long long int x, long long int n, long long int M) {
    long long int ans = 1;
    while (n > 0) {
      if (n & 1) { // if current bit is 1
        ans = (ans * x) % M;
      }
      x = (x * x) % M; // square base
      n = n >> 1;      // move to next bit
    }
    return ans % M;
  }
};
```

📊 Time & Space Complexity

Metric	Value
⏱ Time	**O(log n)**
📦 Space	**O(1)**

🔁 Recursive Version (💧 Same Logic)

```
long long int PowModRecursive(long long x, long long n, long long M) {
    if (n == 0) return 1;
    long long half = PowModRecursive(x, n / 2, M);
    long long result = (half * half) % M;
    if (n % 2 == 1) result = (result * x) % M;
    return result;
}
```

> But watch out for **stack overflow** for huge `n`.

🛡️ When Do You Use This?

- Cryptography (RSA 🔐)
- Hashing
- Competitive Programming
- Large exponent calculations without overflow

🚀 Summary Table

Method	Time Complexity	Space	Comment
Brute Force	O(n)	O(1)	❌ Not for large `n`
Iterative Binary	O(log n)	O(1)	✅ Best for large inputs
Recursive Binary	O(log n)	O(log n)	❌ Risk of stack overflow

📚 Tags

`modular exponentiation`, `binary exponentiation`, `fast power`, `math`, `modulo arithmetic`, `competitive programming`
