Binary Exponentiation

We want to compute:

$$a^n$$

For example:

$$3^{13}, 5^{1000000}, 2^{10^9}$$

The naive approach is to multiply a by itself n times:

```
result = 1
repeat n times:
    result *= a
```

This takes O(n) operations.
If n = 10 ^ 9, that's impossible in time.

Binary exponentiation reduces the time to O(log n).
This is huge in programming contests.

How Binary Exponentiation Thinks About Powers

Every number can be written in binary:

Example:

$$13 = 1101_2$$

What does that notation mean?

13 = 1·8 + 1·4 + 0·2 + 1·1

So:

$$a^{13} = a^{8+4+1} = a^8 . a^4 . a^1$$

That's just using:

$$a^{b+c} = a^b . a^c$$

rather than multiplying a one at a time.

How and Why It Works

Instead of computing all powers from scratch, binary exponentiation uses two rules:

1. If the exponent is even:

$$a^n = (a^{n/2})^2$$

2. If the exponent is odd:

$$a^n = a . (a^{(n-1)/2})^2$$

Why is this valid?

If n is even, say 10:

$$a^{10} = a^5 . a^5$$

So computing a^5 once and squaring it gets the job done.

If n is odd, say 13:

$$a^{13} = a . a^{12} = a . (a^6)^2$$

Convert the problem into smaller exponent problems and re-use previously computed values instead of doing everything from scratch.

This shrinks the problem very fast: each step roughly halves the remaining exponent.

Why This Is So Efficient

In each step you reduce n by at least half, so:

T(n) = T(n/2) + O(1)

This gives:

T(n) = O(log n)

Instead of O(n) multiplications, we only need about log(n).

This massive drop is what makes exponentiation feasible in competitive problems.

Binary exponentiation =

"Write exponent in binary.
Square the base for each bit.
Multiply only for bits that are 1."

or even simpler:

"Use squaring to grow powers exponentially fast,
and pick only the ones we need."

Recursive Binary Exponentiation

```
long long binpow_rec(long long a, long long n) {
    if (n == 0) return 1;           // base case
    long long half = binpow_rec(a, n / 2);

    if (n % 2 == 0)
        return half * half;         // even power
    else
        return half * half * a;     // odd power
}
```

Iterative Binary Exponentiation

This uses bit logic and is preferred in contests.

```
long long binpow_iter(long long a, long long n) {
    long long result = 1;

    while (n > 0) {
        if (n & 1)              // if current bit is 1
            result *= a;
        a *= a;                 // square base
        n >>= 1;                // divide exponent by 2
    }
    return result;
}
```

Modular Fast Exponentiation

Computes:

$$a^n \bmod m$$

```
long long modpow(long long a, long long n, long long mod) {
    long long result = 1;
    a %= mod;

    while (n > 0) {
        if (n & 1)
            result = (result * a) % mod;

        a = (a * a) % mod;
        n >>= 1;
    }
    return result;
}
```