Phase 1 Remastered - Part 2

Progress Tracker

Leaderboard

Quiz Leaderboard

9.0 Fermat's Little Secret: When Overflow is What We Want

Expected Duration to Complete: 1 Week

Table of Contents

Video Class

Phase_1 Remastered Part_2 Class_9

We aleady learned about basic modular arithmetic in previous classes. But we will learn more about it in this class.

Exponentiation

Naive

Statement: Given two integers x and n, find x^n modulo m.

Constraints: $1 \leq x \leq 10^9, 0 \leq n \leq 10^6, 1 \leq m \leq 10^9 + 7$. **(mod is large, but n is small)**

Solution: We can just multiply x n times. So the complexity is $O(n)$.

Collapse With Style Copy code

```
#include<bits/stdc++.h>
using namespace std;

int power(int x, int n, int mod) { // O(n)
  int ans = 1 % mod;
  for (int i = 1; i <= n; i++) {
    ans = 1LL * ans * x % mod;
  }
  return ans;
}
```

```cpp
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int x = 1e8 + 9, n = 1e6, m = 1e9 + 7;
  cout << power(x, n, m) << '\n';
  return 0;
}
```

Circular

Statement: Given two integers x and n, find x^n modulo m.

Constraints: $1 \leq x \leq 10^9, 0 \leq n \leq 10^9, 1 \leq m \leq 10^6$. **(mod is small, but n is large)**

Solution: If we print out the values of $x^0, x^1, x^2, x^3, \ldots$ under modulo m we will see that they are repeating after some point. So we can just find the period of the sequence and then find the answer. Also, the period is always $\leq m$ because there are only m possible values of $x^i \mod m$.

Collapse With Style Copy code

```cpp
#include<bits/stdc++.h>
using namespace std;

int power(int x, int n, int mod) { // O(mod)
  if (mod == 1) return 0;
  vector<int> a;
  vector<bool> vis(mod, false);
  a.push_back(1); // x^0
  vis[1] = true;
  int cur = 1, st = 0;
  while (true) {
    cur = 1LL * cur * x % mod;
    if (vis[cur]) {
      st = find(a.begin(), a.end(), cur) - a.begin();
      break;
    }
    vis[cur] = true;
    a.push_back(cur);
  }
  // for (auto x: a) {
  //   cout << x << ' ';
  // }
  // cout << '\n';
  // cout << "st = " << st << '\n';

  int period = (int)a.size() - st;
```

```
  if (n < st) return a[n];
  n -= st;
  return a[st + n % period];
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int x = 2, n = 9, m = 40;
  cout << power(x, n, m) << '\n';
  return 0;
}
```

Binary Exponentiation

Statement: Given three integers x, n and m, find x^n modulo m.

Constraints: $1 \leq x \leq 10^9, 0 \leq n \leq 10^{18}, 1 \leq m \leq 10^9 + 7$. **(mod is large, and n is also large)**

Tutorial: link, first half.

Recursive

We can use *divide and conquer* to solve this problem. Let's say we want to find x^n. If n is even, then we can find $x^{\frac{n}{2}}$ and then square it. If n is odd, then we can find $x^{\lfloor \frac{n}{2} \rfloor}$ and then square it and then multiply by x. So the recurrence is:

- $x^n = 1$ if $n = 0$.
- $x^n = x^{\frac{n}{2}} * x^{\frac{n}{2}}$ if n is even.
- $x^n = x^{\lfloor \frac{n}{2} \rfloor} * x^{\lfloor \frac{n}{2} \rfloor} * x$ if n is odd.

So each time we are dividing n by 2. So the complexity is $O(\log n)$.

Collapse With Style Copy code

```
#include<bits/stdc++.h>
using namespace std;

int power(int x, long long n, int mod) { // O(log n)
  if (n == 0) return 1 % mod;
  int cur = power(x, n / 2, mod);
  if (n % 2 == 0) {
    return 1LL * cur * cur % mod;
  }
  else {
    return 1LL * cur * cur % mod * x % mod;
  }
```

```
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int x = 1e8 + 9; long long n = 1e18; int m = 1e9 + 7;
  cout << power(x, n, m) << '\n';
  return 0;
}
```

Exercise: Solve this problem.

Iterative V1

To understand and feel the iterative version, let's look at the same solution differently.

Any integer n can be written as a sum of powers of 2. For example, $13 = 2^3 + 2^2 + 2^0$. So $x^{13} = x^{2^3+2^2+2^0} = x^{2^3} * x^{2^2} * x^{2^0}$. So we can just find $x^{2^0}, x^{2^1}, x^{2^2}, \ldots$ and multiply them if the corresponding bit of n is 1. To calculate x^{2^i}, we can just square $x^{2^{i-1}}$. Pretty easy, right?

Collapse With Style Copy code

```
#include<bits/stdc++.h>
using namespace std;

int power(int x, long long n, int mod) { // O(log n)
  int ans = 1 % mod, cur = x % mod;
  for (int i = 0; (1LL << i) <= n; i++) {
    // here, cur = x^{2^i}
    if (n & (1LL << i)) { // if the i'th bit of n is 1
      ans = 1LL * ans * cur % mod;
    }
    cur = 1LL * cur * cur % mod;
  }
  return ans;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int x = 1e8 + 9; long long n = 1e18; int m = 1e9 + 7;
  cout << power(x, n, m) << '\n';
  return 0;
}
```

Iterative V2

We can write same code in a different way. Basically we can get the binary representation of n from LSB to MSB by dividing n by 2 each time. If the current bit is 1, then we multiply the answer by x^{2^i}.

Collapse With Style Copy code

```
#include<bits/stdc++.h>
using namespace std;

int power(int x, long long n, int mod) { // O(log n)
  int ans = 1 % mod;
  while (n > 0) {
    if (n & 1) {
      ans = 1LL * ans * x % mod;
    }
    x = 1LL * x * x % mod;
    n >>= 1;
  }
  return ans;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int x = 1e8 + 9; long long n = 1e18; int m = 1e9 + 7;
  cout << power(x, n, m) << '\n';
  return 0;
}
```

This solution might be the easiest to write, also iterative version is faster than recursive version in most cases.

By the way, binary exponentiation under modulo is also called **Bigmod**.

Mulmod

Statement: Given three integers x, y and m, find $x * y$ modulo m.

Constraints: $1 \leq x, y \leq 10^{18}, 1 \leq m \leq 10^{18} + 7$. **(x, y and mod are very large)**

Solution: We can't just multiply x and y and then take modulo by m because $x * y$ may overflow. If m was small, we could just use the identity $(x * y) \mod m = ((x \mod m) * (y \mod m)) \mod m$. So there would be no overflow. But as m is large, even after using this identity, there will be overflow. So we need to find a way to multiply x and y without overflowing.

But notice that $x * y$ means $x + x + x + \cdots + x$ (y times). So we can just add x to itself y times and then take modulo by m. And we can make this process faster by using binary exponentiation.

Collapse With Style Copy code

```
#include<bits/stdc++.h>
using namespace std;

long long mulmod(long long x, long long y, long long mod) { // O(log y)
  long long ans = 0;
  while (y > 0) {
    if (y & 1) {
      ans = (ans + x) % mod;
    }
    x = (x + x) % mod;
    y >>= 1;
  }
  return ans;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  long long x = (long long)1e15 + 2, y = (long long)1e13 + 4, m = (long long)1e18 + 7;
  cout << mulmod(x, y, m) << '\n';
  return 0;
}
```

__int128

There is also a data type for `c++17` or later versions. And that is `__int128` i.e. `128` bit integer! One problem with this data type is that you can't take input or output in `__int128` directly. But you can take input in `string` and then convert it to `__int128` manually or when you want to print `__int128`, you can convert it to `string` manually and then print it.

Collapse With Style Copy code

```
#include<bits/stdc++.h>
using namespace std;

__int128 read() {
  string s; cin >> s;
  __int128 ans = 0;
  for (int i = 0; i < s.size(); i++) {
    ans = ans * 10 + (s[i] - '0');
  }
  return ans;
```

```
}

string to_string(__int128 x) {
  string s;
  while (x > 0) {
    s += (char)(x % 10 + '0');
    x /= 10;
  }
  reverse(s.begin(), s.end());
  return s;
}

void write(__int128 x) {
  cout << to_string(x) << '\n';
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  __int128 x = read(), y = read();
  __int128 ans = x * y;
  ans += 2;
  write(ans);
  return 0;
}
```

Good thing is you can do almost all the operations that you can do with `int` or `long long` with `__int128` too. This is helpful when you want to multiply two large integers and the result may overflow.

When the modulo is, let's say, $10^{18} + 7$, then while multiplying two integers by this modulo, you can just typecast to `__int128` to avoid overflows. So our `mulmod` function will look like this:

Collapse With Style Copy code

```
long long mulmod(long long x, long long y, long long mod) { // O(1)
  return (long long)((__int128)x * y % mod);
}
```

Keep in mind that `__int128` is 2-3 times slower than `int` or `long long`. So don't use it unless you need it.

When Overflow is What We Want!

Statement: Given two integers x, n, find x^n modulo 2^{64}.

Constraints: $1 \leq x \leq 10^9, 0 \leq n \leq 10^{18}$

Note that $x \mod 2^k = x \& (2^k - 1)$. That is $x \mod 2^k$ is equal to the last k bits of x.

Now let's say we have two `unsigned long long` integers x and y and we want to find $x * y \mod 2^{64}$. The thing with `unsigned long long` is that it will automatically take modulo by 2^{64} when the result overflows. So we can just multiply x and y and that's it!

Collapse With Style Copy code

```
#include<bits/stdc++.h>
using namespace std;

#define ull unsigned long long

ull power(ull x, ull n) { // O(log n)
  ull ans = 1;
  while (n > 0) {
    if (n & 1) {
      ans *= x;
    }
    x *= x;
    n >>= 1;
  }
  return ans;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  ull x = 1e9 + 7, n = 1e18;
  cout << power(x, n) << '\n'; // x^n mod 2^64
  return 0;
}
```

Similarly, if we want to take modulo by 2^{32}, we can just use `unsigned int` instead of `unsigned long long`.

Fermat's Little Theorem

Theorem: If p is a prime and a is an integer not divisible by p, then $a^{p-1} \equiv 1 \mod p$.

For example, try to run the following code:

Collapse With Style Copy code

```
#include<bits/stdc++.h>
using namespace std;

// print x^1, x^2, x^3, ..., x^p-1 mod p
```

```cpp
void print(int p) {
  cout << p << " => ";
  int a = 2, cur = a % p;
  for (int i = 1; i < p; i++) {
    cout << cur << ' ';
    cur = 1LL * cur * a % p;
  }
  cout << '\n';
}
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  for (int p = 3; p <= 15; p++) {
    print(p);
  }
  return 0;
}
```

The result will be:

Collapse With Style Copy code

```
3 => 2 1
4 => 2 0 0
5 => 2 4 3 1
6 => 2 4 2 4 2
7 => 2 4 1 2 4 1
8 => 2 4 0 0 0 0 0
9 => 2 4 8 7 5 1 2 4
10 => 2 4 8 6 2 4 8 6 2
11 => 2 4 8 5 10 9 7 3 6 1
12 => 2 4 8 4 8 4 8 4 8 4 8
13 => 2 4 8 3 6 12 11 9 5 10 7 1
14 => 2 4 8 2 4 8 2 4 8 2 4 8 2
15 => 2 4 8 1 2 4 8 1 2 4 8 1 2 4
```

You will notice that:

- For all primes p, $x^{p-1} \equiv 1 \mod p$.
- For all primes, all x^i over $[1, p-1]$ are distinct modulo p.
- There are some non-primes p for which $x^{p-1} \equiv 1 \mod p$ (if you try with $a = 4$ and $p = 15$ you will find that out)

Proof of the Theorem: must check.

So this also means the following:

- $x^n \equiv x^{n \mod (p-1)} \mod p$ if p is prime.

This is because $x^0, x^1, x^2, x^3, \ldots$ cycles after $p-1$ terms.

Exercise: Solve this problem.

Collapse With Style

Copy code

```
#include<bits/stdc++.h>
using namespace std;

int power(int x, long long n, int mod) { // O(log n)
  int ans = 1 % mod;
  while (n > 0) {
    if (n & 1) {
      ans = 1LL * ans * x % mod;
    }
    x = 1LL * x * x % mod;
    n >>= 1;
  }
  return ans;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int mod = 1e9 + 7;
  int q; cin >> q;
  while (q--) {
    int a, b, c; cin >> a >> b >> c;
    int e = power(b, c, mod - 1);
    int ans = power(a, e, mod);
    cout << ans << '\n';
  }
  return 0;
}
```

Modular Multiplicative Inverse

Naive

A number a has a modular multiplicative inverse modulo m if there exists a number b such that $a * b \equiv 1 \mod m$. If a has a modular multiplicative inverse modulo m, then we can write a^{-1} for the modular multiplicative inverse of a modulo m.

For example, $2^{-1} \mod 5 = 3$ because $2 * 3 \equiv 1 \mod 5$.

Collapse With Style Copy code

```cpp
#include<bits/stdc++.h>
using namespace std;

int inverse(int a, int m) { // O(m)
  for (int i = 1; i < m; i++) {
    if (1LL * a * i % m == 1) {
      return i;
    }
  }
  return -1; // inverse doesn't exist
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cout << inverse(2, 5) << '\n'; // 3
  cout << inverse(3, 6) << '\n'; // -1
  return 0;
}
```

Properties

- Not all numbers have a modular multiplicative inverse.
- a has a modular multiplicative inverse modulo m if and only if a and m are coprime (i.e. $\gcd(a, m) = 1$). Proof will be discussed later.
- If a has a modular multiplicative inverse modulo m, then it is unique modulo m.
  - **Proof:** To elaborate, if b and c are multiplicative inverses of a modulo m, then by definition:

$$\begin{aligned} ab &\equiv 1 \mod m \\ ac &\equiv 1 \mod m \end{aligned}$$

    Subtracting the two equations, we get:

$$\begin{aligned} ab - ac &\equiv 0 \mod m \\ a(b - c) &\equiv 0 \mod m \end{aligned}$$

    Since $\gcd(a, m) = 1$, this means that m must divide $b - c$. However, both b and c are between 0 and $m - 1$ (since they are residues modulo m), so the only way m could divide $b - c$ is if $b - c = 0$, or $b = c$.

    Therefore, the multiplicative inverse is unique modulo m when a and m are coprime.
- If a has a modular multiplicative inverse modulo m, then all $a * i \mod m$ are distinct for $i \in [1, m]$.

- **Proof:** Let's say $a * i \equiv a * j \mod m$ for some $i \neq j$. Then we can multiply both sides by a^{-1} to get $i \equiv j \mod m$. But this is not possible because i and j are between 1 and $m - 1$, so they must be equal. So $a * i \mod m$ are distinct for $i \in [1, m - 1]$.

Faster

From Fermat's Little Theorem, we know that $a^{p-1} \equiv 1 \mod p$ if p is a prime and a is not divisible by p. Now divide both sides by a to get $a^{p-2} \equiv a^{-1} \mod p$. So we can just use binary exponentiation to find $a^{-1} \mod p$.

Collapse With Style Copy code

```
#include<bits/stdc++.h>
using namespace std;

int power(int x, int n, int mod) { // O(log n)
  int ans = 1 % mod;
  while (n > 0) {
    if (n & 1) {
      ans = 1LL * ans * x % mod;
    }
    x = 1LL * x * x % mod;
    n >>= 1;
  }
  return ans;
}

// m is prime
int inverse(int a, int m) { // O(log m)
  return power(a, m - 2, m);
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cout << inverse(2, 5) << '\n'; // 3
  return 0;
}
```

Exercise: Solve this problem.

How to find modular multiplicative inverse modulo m if m is not prime? We will learn about it in Phase 2.

Modular Division

Statement: Given three integers a, b, m, find a/b modulo m.

Constraints: $1 \leq a, b < m, 2 \leq m \leq 10^9 + 7$ and m is prime.

Solution: We can just multiply a by the modular multiplicative inverse of b modulo m. So the complexity is $O(\log m)$.

Exercise: Solve this problem.

Why always $10^9 + 7$ or 998244353?

You will see that in most problems, the mod is $10^9 + 7$ or 998244353.

The main reason is that normally problems has values $\leq 10^9$ to keep things in `int` data type, so the mod should be $> 10^9$ and also a prime (because modular inverse must exist for all numbers). And the smallest number that satisfies this condition is indeed $10^9 + 7$.

Now you know it!

And about 998244353, this is an NTT friendly prime. You may not know what this even means, so don't try to understand it right now. You will get to know about it later in your life, but understand that for some specific problems, this mod is very useful.

One Specific Property of Modulo

Property:

- $x \mod m = x$ if $m > x$.
- $x \mod m < \frac{x}{2}$ when $m \leq x$.
  - **Proof:** if $m \leq \frac{x}{2}$, then $x \mod m < \frac{x}{2}$. Otherwise, $x \mod m = x - m < \frac{x}{2}$ because $m > \frac{x}{2}$.

For example, $20 \mod m$ for $m = 1, 2, 3, \ldots, 25$ is:

Collapse With Style Copy code

```
0 0 2 0 0 2 6 4 2 0 9 8 7 6 5 4 3 2 1 0 20 20 20 20 20
```

Exercise 1: Solve this.

Exercise 2: Solve this.

Collapse With Style Copy code

```cpp
#include<bits/stdc++.h>
using namespace std;

#define ll long long
const int N = 1e5 + 9;
```

```
ll a[N];
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int t, cs = 0; cin >> t;
  while (t--) {
    int n; cin >> n;
    priority_queue<pair<ll, int>> q;
    for (int i = 1; i <= n; i++) {
      cin >> a[i];
      q.push({a[i], i});
    }
    int Q; cin >> Q;
    while (Q--) {
      ll m; cin >> m;
      // for (int i = 1; i <= n; i++) {
      //   a[i] = a[i] % m;
      // }
      while (!q.empty()) {
        auto [x, i] = q.top();
        if (x < m) break;
        q.pop();
        q.push({x % m, i});
      }
    }
    while (!q.empty()) {
      auto [x, i] = q.top();
      q.pop();
      a[i] = x;
    }
    cout << "Case " << ++cs << ":\n";
    for (int i = 1; i <= n; i++) {
      cout << a[i] << (i < n ? ' ' : '\n');
    }
  }
  return 0;
}
```

Permutations and Combinations

We already discussed the definition of permutations and combinations in previous classes. But we will learn how to calculate them efficiently in this class.

Computing Modulo a Prime

Statement: You are given q queries. In each query, you are given two integers n and r, you will have to find nPr and nCr modulo $10^9 + 7$.

Constraints: $1 \leq n, q \leq 10^6, 0 \leq r \leq n$.

Solution: $nPr = \frac{n!}{(n-r)!}$. So we can just calculate $n!$ and $(n-r)!$ and then divide them. For division, we can just multiply $(n-r)!$ by the modular multiplicative inverse of $(n-r)!$ modulo $10^9 + 7$. Similarly, $nCr = \frac{n!}{r!*(n-r)!}$.

Collapse With Style

Copy code

```cpp
#include<bits/stdc++.h>
using namespace std;

const int N = 1e6 + 9, mod = 1e9 + 7;

int fact[N], ifact[N];

int power(int x, int n) { // O(log n)
  int ans = 1;
  while (n > 0) {
    if (n & 1) {
      ans = 1LL * ans * x % mod;
    }
    x = 1LL * x * x % mod;
    n >>= 1;
  }
  return ans;
}

int inverse(int a) { // O(log mod)
  return power(a, mod - 2);
}

void prec() { // O(n)
  fact[0] = 1;
  for (int i = 1; i < N; i++) {
    fact[i] = 1LL * fact[i - 1] * i % mod;
  }
  ifact[N - 1] = inverse(fact[N - 1]);
  for (int i = N - 2; i >= 0; i--) {
    ifact[i] = 1LL * ifact[i + 1] * (i + 1) % mod; // 1 / i! = (1 / (i + 1)!) * (i + 1)
  }
}

int nPr(int n, int r) { // O(1)
  if (n < r) return 0;
```

```
  return 1LL * fact[n] * ifact[n - r] % mod;
}

int nCr(int n, int r) { // O(1)
  if (n < r) return 0;
  return 1LL * fact[n] * ifact[r] % mod * ifact[n - r] % mod;
}

int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  prec();
  int q; cin >> q;
  while (q--) {
    int n, r; cin >> n >> r;
    cout << nPr(n, r) << ' ' << nCr(n, r) << '\n';
  }
  return 0;
}
// O(n + q)
```

Exercise 1: Solve this problem.

Exercise 2: Solve this problem.

Collapse With Style

Copy code

```
#include<bits/stdc++.h>
using namespace std;

const int N = 1e6 + 2, mod = 1e9 + 7;
int power(int x, long long n) { // O(log n)
  int ans = 1 % mod;
  while (n > 0) {
    if (n & 1) {
      ans = 1LL * ans * x % mod;
    }
    x = 1LL * x * x % mod;
    n >>= 1;
  }
  return ans;
}
int inverse(int a) {
  return power(a, mod - 2);
}
int fac[N], ifac[N];
void prec() { // O(n)
```

```
  fac[0] = 1;
  for (int i = 1; i < N; i++) {
    fac[i] = 1LL * fac[i - 1] * i % mod;
  }
  ifac[N - 1] = inverse(fac[N - 1]);
  for (int i = N - 2; i >= 0; i--) {
    ifac[i] = 1LL * ifac[i + 1] * (i + 1) % mod;
  }
}
int nCr(int n, int r) { // O(1)
  if (n < 0 or n < r) return 0;
  return 1LL * fac[n] * ifac[r] % mod * ifac[n - r] % mod;
}
int nPr(int n, int r) {
  if (n < 0 or n < r) return 0;
  return 1LL * fac[n] * ifac[n - r] % mod;
}
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  prec();
  string s; cin >> s;
  int n = s.size();
  int ans = fac[n];
  map<char, int> cnt;
  for (auto x: s) {
    cnt[x]++;
  }
  for (auto [_, c]: cnt) {
    ans = 1LL * ans * ifac[c] % mod;
  }
  cout << ans << '\n';
  return 0;
}
```

Computing using a Recurrence

To sompute nCr, we can use the following recurrence:

$$C(n,r) = C(n-1,r-1) + C(n-1,r)$$

with base cases:

- $C(n,0) = C(n,n) = 1$.

Proof: $C(n,r)$ is the number of ways to choose r objects from n objects. Now let's say we have n objects and we want to choose r objects from them. Now there are two cases:

- We choose the n'th object. Then we have to choose $r-1$ objects from the remaining $n-1$ objects. So the number of ways to do this is $C(n-1, r-1)$.
- We don't choose the n'th object. Then we have to choose r objects from the remaining $n-1$ objects. So the number of ways to do this is $C(n-1, r)$.

So the total number of ways is $C(n-1, r-1) + C(n-1, r)$.

Collapse With Style Copy code

```
const int N = 2005, mod = 1e9 + 7;

int C[N][N], fact[N];
void prec() { // O(n^2)
  for (int i = 0; i < N; i++) {
    C[i][0] = C[i][i] = 1;
    for (int j = 1; j < i; j++) {
      C[i][j] = (C[i - 1][j - 1] + C[i - 1][j]) % mod;
    }
  }
  fact[0] = 1;
  for (int i = 1; i < N; i++) {
    fact[i] = 1LL * fact[i - 1] * i % mod;
  }
}

int nCr(int n, int r) { // O(1)
  if (n < r) return 0;
  return C[n][r];
}

int nPr(int n, int r) { // O(1)
  if (n < r) return 0;
  return 1LL * nCr(n, r) * fact[r] % mod;
}
```

One good thing about this approach is that we can calculate nCr even if the modulo is not a prime. But the bad thing is that it is slower than the previous approach.

We will learn more about combinatorics in Phase 2.

Binary Exponentiation Style Problems

Lots of problems can be solved using the similar recursive approach of binary exponentiation. Let's see some of them.

- Mulmod: We already discussed this problem. Basically when you have to do something k times, then most probably you can use binary exponentiation to solve it faster.
- Another Problem: link.

Collapse With Style

Copy code

```
#include<bits/stdc++.h>
using namespace std;

#define ll long long
int power(int x, long long n, int mod) { // O(log n)
  int ans = 1 % mod;
  while (n > 0) {
    if (n & 1) {
      ans = 1LL * ans * x % mod;
    }
    x = 1LL * x * x % mod;
    n >>= 1;
  }
  return ans;
}
// a^0 + a^1 + ... + a^x (under mod m)
int solve(ll x, int a, int mod) {
  if (x == 0) {
    return 1 % mod;
  }
  if (x & 1) {
    ll p = x / 2;
    int cur = solve(p, a, mod);
    int ans = (cur + 1LL * power(a, p + 1, mod) * cur % mod) % mod;
    return ans;
  }
  else {
    int ans = (solve(x - 1, a, mod) + power(a, x, mod)) % mod;
    return ans;
  }
}
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int a; ll x; int m; cin >> a >> x >> m;
  cout << solve(x - 1, a, m) << '\n';
  return 0;
}
```

Privacy Policy Terms & Conditions