快速幂算法是一种高效计算大整数幂次的方法，时间复杂度为 $O(\log n)$，远优于直接的幂运算 $O(n)$。其核心思想是将指数的二进制展开，通过平方和乘法快速计算幂次。以下是详细的解释和实现。

核心思想

假设我们要计算 $a^b \mod m$，其基本思想如下：

1. 如果 b 是偶数：
   $a^b = (a^{b/2})^2$
2. 如果 b 是奇数：
   $a^b = a \cdot a^{b-1}$

通过这种递归分治的方法，可以将指数迅速减半，从而提高计算效率。

算法步骤

1. 初始化结果 $result = 1$。
2. 当 $b > 0$ 时，执行以下循环：
   - 如果 b 是奇数，更新结果：$result = (result \cdot a) \mod m$。
   - 更新基数：$a = (a \cdot a) \mod m$。
   - 将 b 右移一位（相当于除以 2）。
3. 返回结果。

代码实现

以下是 C++ 代码实现：

```cpp
#include <iostream>
using namespace std;

// 快速幂算法，计算 (base^exp) % mod
long long modExp(long long base, long long exp, long long mod) {
    long long result = 1;  // 初始化结果
    while (exp > 0) {
        if (exp % 2 == 1) {  // 如果 exp 是奇数
            result = (result * base) % mod;
        }
        base = (base * base) % mod;  // 基数自乘
        exp /= 2;  // 指数右移一位
    }
    return result;
}

int main() {
    long long base = 3, exp = 200, mod = 13;
    cout << base << "^" << exp << " % " << mod << " = " << modExp(base, exp, mod) << endl;
    return 0;
}
```

逐行解释

1. **函数声明和初始化**：

   ```cpp
   long long modExp(long long base, long long exp, long long mod) {
       long long result = 1;
   ```

   - `modExp` 函数接受三个参数：`base`（底数），`exp`（指数），和 `mod`（模数）。
   - `result` 初始化为 1，将用于存储最终结果。

2. **循环**：

   ```cpp
       while (exp > 0) {
   ```

   - 当 `exp` 大于 0 时，继续循环。

3. **处理指数为奇数的情况**：

   ```cpp
           if (exp % 2 == 1) {
               result = (result * base) % mod;
           }
   ```

- 如果 `exp` 是奇数（即 `exp % 2 == 1`），更新 `result` 为 `(result * base) % mod`。

4. **更新基数和指数**：

```
        base = (base * base) % mod;
        exp /= 2;
    }
```

- 将 `base` 更新为 `(base * base) % mod`，这一步相当于将 `base` 平方。
- 将 `exp` 右移一位（相当于除以 2）。

5. **返回结果**：

```
    return result;
}
```

- 当 `exp` 为 0 时，循环结束，返回 `result`。

6. **主函数**：

```
int main() {
    long long base = 3, exp = 200, mod = 13;
    cout << base << "^" << exp << " % " << mod << " = " << modExp(base, exp, mod) <<
    return 0;
}
```

- 定义底数、指数和模数，调用 `modExp` 计算结果并输出。

示例计算

假设我们要计算 $3^{200} \mod 13$，步骤如下：

1. 初始值：`result = 1`，`base = 3`，`exp = 200`，`mod = 13`。
2. 200 是偶数，计算 `base = (3 * 3) % 13 = 9`，`exp = 200 / 2 = 100`。
3. 100 是偶数，计算 `base = (9 * 9) % 13 = 81 % 13 = 3`，`exp = 100 / 2 = 50`。
4. 50 是偶数，计算 `base = (3 * 3) % 13 = 9`，`exp = 50 / 2 = 25`。
5. 25 是奇数，计算
   `result = (1 * 9) % 13 = 9`，`base = (9 * 9) % 13 = 81 % 13 = 3`，`exp = 25 / 2 = 12`。
6. 重复上述步骤，直到 `exp` 为 0。

最终结果为 `3^{200} % 13 = 9`。

通过这种方法，可以高效地计算大整数幂次的模运算。如果有其他问题或需要进一步的解释，随时告诉我！

要使用费马小定理和快速幂来计算 $\frac{a}{b} \mod p$，需要先了解费马小定理和快速幂算法。

费马小定理

费马小定理表明，对于一个质数 p 和任意整数 a，如果 a 不被 p 整除，则：

$$a^{p-1} \equiv 1 \pmod{p}$$

通过这个定理，可以得到 a 的逆元：

$$a^{p-2} \equiv a^{-1} \pmod{p}$$

快速幂算法

快速幂算法用于高效地计算大整数的幂模某个数的值。其时间复杂度是 $O(\log n)$。

实现步骤

1. **使用费马小定理求模逆元**。
2. **使用快速幂算法求出 $b^{p-2} \mod p$ 作为 b 的逆元**。
3. **将分数转换为乘法形式**。

C++ 实现

以下是使用费马小定理和快速幂算法计算 $\frac{a}{b} \mod p$ 的 C++ 实现：

```cpp
#include <iostream>
using namespace std;

// 快速幂算法，计算 (base^exp) % mod
long long modExp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {  // 如果 exp 是奇数
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

// 求 b 在模 p 下的逆元，使用费马小定理
long long modInverse(long long b, long long p) {
    // b^(p-2) ≡ b^(-1) (mod p)
    return modExp(b, p - 2, p);
}

// 计算 (a / b) % p
long long modDivide(long long a, long long b, long long p) {
    long long inv = modInverse(b, p);
    return (a * inv) % p;
}

int main() {
    long long a = 10, b = 3, p = 7;
    try {
        long long result = modDivide(a, b, p);
        cout << "Result of " << a << "/" << b << " mod " << p << " is: " << result << er
    } catch (const invalid_argument &e) {
        cout << e.what() << endl;
    }
    return 0;
}
```

逐行解释

1. **快速幂算法**：

```
long long modExp(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {  // 如果 exp 是奇数
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}
```

- `modExp` 函数实现快速幂算法，通过不断将指数减半，并在指数为奇数时乘以当前基数。
- 循环中，基数平方并取模，指数减半，最终结果为 $(base^{exp}) \mod mod$。

2. **求模逆元**：

```
long long modInverse(long long b, long long p) {
    return modExp(b, p - 2, p);
}
```

- 使用费马小定理计算 b 的逆元，即 $b^{p-2} \mod p$。

3. **计算** $\frac{a}{b} \mod p$：

```
long long modDivide(long long a, long long b, long long p) {
    long long inv = modInverse(b, p);
    return (a * inv) % p;
}
```

- 调用 `modInverse` 求出 b 的模逆元 inv。
- 计算并返回 $(a \times inv) \mod p$。

4. **主函数**：

```
int main() {
    long long a = 10, b = 3, p = 7;
    try {
        long long result = modDivide(a, b, p);
        cout << "Result of " << a << "/" << b << " mod " << p << " is: " << result <
    } catch (const invalid_argument &e) {
        cout << e.what() << endl;
    }
    return 0;
}
```

- 定义 a, b, p 的值。
- 调用 `modDivide` 计算结果并输出。

使用扩展欧几里得算法（Extended Euclidean Algorithm, Exgcd）来求解模逆元也是一种经典且高效的方法，特别是在处理大数时更为优越。接下来，我将详细解释如何使用扩展欧几里得算法来实现分数 $\frac{a}{b} \mod p$ 的计算，并与快速幂进行比较。

扩展欧几里得算法的原理

扩展欧几里得算法不仅可以用于求两个数的最大公约数（GCD），还能求出模逆元。对于两个整数 a 和 b，扩展欧几里得算法可以找到一组整数 x 和 y，使得：

$$ax + by = \gcd(a, b)$$

当 a 和 b 互质时，$\gcd(a, b) = 1$，因此方程变为：

$$ax + by = 1$$

在这种情况下，x 就是 a 模 b 的逆元，即：

$$a^{-1} \equiv x \pmod{b}$$

C++ 实现

下面是使用扩展欧几里得算法计算分数 $\frac{a}{b} \mod p$ 的 C++ 实现代码：

```cpp
#include <iostream>
using namespace std;

// 扩展欧几里得算法
long long extendedGCD(long long a, long long b, long long &x, long long &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long x1, y1;
    long long gcd = extendedGCD(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return gcd;
}

// 求 b 的逆元，前提是 b 和 p 互质
long long modInverse(long long b, long long p) {
    long long x, y;
    long long g = extendedGCD(b, p, x, y);
    if (g != 1) {
        throw invalid_argument("Inverse doesn't exist.");
    } else {
        // 确保结果是正数
        return (x % p + p) % p;
    }
}

// 计算 (a / b) % p
long long modDivide(long long a, long long b, long long p) {
    long long inv = modInverse(b, p);
    return (a * inv) % p;
}

int main() {
    long long a = 10, b = 3, p = 7;
    try {
        long long result = modDivide(a, b, p);
        cout << "Result of " << a << "/" << b << " mod " << p << " is: " << result << er
    } catch (const invalid_argument &e) {
        cout << e.what() << endl;
    }
```

```
    return 0;
}
```

逐行解释

1. **扩展欧几里得算法**：

```
long long extendedGCD(long long a, long long b, long long &x, long long &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    long long x1, y1;
    long long gcd = extendedGCD(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return gcd;
}
```

   - 当 $b = 0$ 时，直接返回 a 作为 gcd，设 $x = 1$ 和 $y = 0$。
   - 递归调用 `extendedGCD(b, a % b, x1, y1)`，计算 gcd 及其对应的系数。
   - 通过回溯，计算当前的 x 和 y。
2. **求模逆元**：

```
long long modInverse(long long b, long long p) {
    long long x, y;
    long long g = extendedGCD(b, p, x, y);
    if (g != 1) {
        throw invalid_argument("Inverse doesn't exist.");
    } else {
        return (x % p + p) % p;
    }
}
```

   - 使用扩展欧几里得算法计算 b 的模逆元 x。
   - 如果 gcd 不为 1，则 b 和 p 不互质，逆元不存在。
   - 否则，返回 x 的正值作为逆元。
3. **计算** $\frac{a}{b} \mod p$：

```
long long modDivide(long long a, long long b, long long p) {
    long long inv = modInverse(b, p);
    return (a * inv) % p;
}
```

- 调用 `modInverse` 求出 b 的逆元 inv。
- 返回 $(a \times inv) \mod p$。

4. **主函数**：

```
int main() {
    long long a = 10, b = 3, p = 7;
    try {
        long long result = modDivide(a, b, p);
        cout << "Result of " << a << "/" << b << " mod " << p << " is: " << result <
    } catch (const invalid_argument &e) {
        cout << e.what() << endl;
    }
    return 0;
}
```

- 定义 a, b, p 的值，计算并输出结果。

为什么使用 Exgcd ？

相比于快速幂算法，Exgcd 在某些场景下更加灵活和高效：

1. **处理更大范围的数**：Exgcd 可以直接处理整数（包括负数），而快速幂需要在计算中额外处理这些情况。
2. **无需质数模**：快速幂算法基于费马小定理，只能在模数为质数的情况下使用。而 Exgcd 不要求模数为质数。
3. **更快的计算速度**：对于某些特定的输入，Exgcd 在常数时间上可能更快。

通过这种方式，你可以在 C++ 中使用扩展欧几里得算法来实现高效的分数模运算。如果你还有其他问题或需要进一步的解释，随时告诉我！