Números primos y factorizaciones en primos

Funciones para encontrar la lista de los primeros k primos y para factorizar números. Incluye factores de fermat.

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <iostream>
#include <numeric>
#include <vector>

using ll = long long;

// Represents p^a
struct prime_to_power
{
    prime_to_power(ll prime, ll power) : p(prime), a(power) {}
    ll p;
    ll a;

    explicit operator ll() const { return std::pow(p, a); }
};

class Factorization
{
public:
    explicit operator ll() const
    {
        ll t = 1;
        for (auto& pa : m_prime_factors)
            t *= ll(pa);
        return t;
    }

    // returns the power of prime p
    ll operator[](ll p) const
    {
        auto it = std::partition_point(
          m_prime_factors.begin(),
          m_prime_factors.end(),
          [p](const prime_to_power& p_a) { return p_a.p < p; });
        if (it == m_prime_factors.end() || it->p != p)
            return 0;
        return it->a;
```

```cpp
    }

    ll& operator[](ll p)
    {
        auto it = std::partition_point(
          m_prime_factors.begin(),
          m_prime_factors.end(),
          [p](const prime_to_power& p_a) { return p_a.p < p; });

        if (it == m_prime_factors.end())
        {
            m_prime_factors.emplace_back(p, 0);
            return m_prime_factors.back().a;
        }

        if (it->p != p)
        {
            ++it;
            it = m_prime_factors.insert(it, prime_to_power(p, 0));
            return it->a;
        }

        return it->a;
    }

    void emplace_back(ll p, ll a)
    {
        assert(m_prime_factors.empty() || p > m_prime_factors.back().p);
        m_prime_factors.emplace_back(p, a);
    }

    auto begin() const { return m_prime_factors.begin(); }

    auto end() const { return m_prime_factors.end(); }

    ll size() const { return m_prime_factors.size(); }

private:
    std::vector<prime_to_power> m_prime_factors;
};

inline std::ostream& operator<<(std::ostream& os, const Factorization& F)
{
    ll i = 0;
    for (auto f : F)
```

```
    {
        os << f.p;
        if (f.a != 1)
            os << "^" << f.a;

        if (i + 1 != F.size())
            os << " * ";

        ++i;
    }
    return os;
}

inline ll int_sqrt(ll n)
{
    double dsqrt = sqrt(double(n));

    return ll(dsqrt + 0.000000000000001);
}

inline bool is_square(ll N)
{
    ll t = int_sqrt(N);
    return t * t == N;
}

inline ll FermatFactor(ll N)
{
    assert(N % 2 == 1);
    ll a = ceil(sqrt(double(N)));
    ll b2 = a * a - N;
    while (!is_square(b2))
    {
        ++a;
        b2 = a * a - N;
    }

    return a - int_sqrt(b2);
}

class PrimeFactorizer
{
public:
    explicit PrimeFactorizer(ll num_primes = 100000)
    {
```

```
    eratosthenes_sieve(num_primes);
}

bool is_prime(ll p) const
{
    if (p <= primes.back())
    {
        return std::binary_search(primes.begin(), primes.end(), p);
    }

    // Maybe comment this out? should test
    // this!!
    if (p <= primes.back() * primes.back())
    {
        return bf_is_prime(p);
    }

    ll a = FermatFactor(p);
    return a == 1;
}

const auto& Primes() const { return primes; }

/// Make sure sqrt(n) <
/// primes.back()*primes.back(), otherwise
/// this could spit out a wrong factorization.
Factorization prime_factorization(ll n)
{
    Factorization F;
    if (n <= 1)
        return F;
    for (auto p : primes)
    {
        ll a = 0;
        while (n % p == 0)
        {
            n /= p;
            ++a;
        }
        if (a != 0)
            F.emplace_back(p, a);

        if (p * p > n)
            break;
    }
```

```
        if (n > 1)
            ++F[n];

        return F;
    }

private:
    void eratosthenes_sieve(ll n)
    {
        std::vector<bool> primecharfunc;
        primecharfunc.resize(n + 1, true);
        primes.reserve((1.1 * n) / log(n) + 50);
        for (ll p = 3; p * p <= n; p += 2)
        {
            if (primecharfunc[p] == true)
            {
                for (ll i = p * 2; i <= n; i += p)
                    primecharfunc[i] = false;
            }
        }

        for (ll p = 11; p < n; p += 2)
            if (primecharfunc[p])
                primes.emplace_back(p);
    }

    // private because n has to be odd, and maybe
    // is already a factor in something.
    void fermat_factorization(ll n, Factorization& F)
    {
        assert(n % 2 == 1);
        assert(n > 5);
        ll a = FermatFactor(n);

        ll b = n / a;

        assert(a * b == n);
        if (a == 1)
        {
            ++F[b];
        }
        else
        {
            fermat_factorization(a, F);
```

```
            fermat_factorization(b, F);
        }
    }

private:
    std::vector<ll> primes = {2, 3, 5, 7};

    bool bf_is_prime(ll n) const
    {
        for (auto p : primes)
        {
            if (p * p > n)
                break;
            if (n % p == 0)
                return false;
        }
        return true;
    }
};

int main()
{
    PrimeFactorizer P;

    std::cout << "Primes: ";
    for (auto it = P.Primes().begin(); it != P.Primes().begin() + 100; ++it)
        std::cout << *it << ' ';
    std::cout << std::endl;

    for (ll n = 2; n <= 30; ++n)
    {
        std::cout << n << " = " << P.prime_factorization(n) << std::endl;
    }

    return 0;
}
```

Output:

```
Primes: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 1
2 = 2
3 = 3
4 = 2^2
5 = 5
6 = 2 * 3
7 = 7
```

```
8 = 2^3
9 = 3^2
10 = 2 * 5
11 = 11
12 = 2^2 * 3
13 = 13
14 = 2 * 7
15 = 3 * 5
16 = 2^4
17 = 17
18 = 2 * 3^2
19 = 19
20 = 2^2 * 5
21 = 3 * 7
22 = 2 * 11
23 = 23
24 = 2^3 * 3
25 = 5^2
26 = 2 * 13
27 = 3^3
28 = 2^2 * 7
29 = 29
30 = 2 * 3 * 5
```