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:
    using value_type = prime_to_power;
    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& PF = m_prime_factors;
        auto it = std::partition_point(
          PF.begin(), PF.end(), [p](const prime_to_power& p_a) {
              return p_a.p < p;
```

```
          });

        if (it == PF.end() || it->p != p)
            return 0;

        return it->a;
    }

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

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

        // if it exists, everything is fine
        if (it->p == p)
            return it->a;

        // Has to insert into correct position
        ++it;
        it = PF.insert(it, prime_to_power(p, 0));
        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;
```

```
};

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;
}

// returns the biggest integer t such that t*t <= n
ll integral_sqrt(ll n)
{
    ll t = std::round(std::sqrt(n));
    if (t * t > n)
        return t - 1;

    return t;
}

bool is_square(ll N)
{
    ll t = std::round(std::sqrt(N));
    return t * t == N;
}

ll FermatFactor(ll N)
{
    assert(N % 2 == 1);
    ll a = std::ceil(std::sqrt(N));
    ll b2 = a * a - N;

    while (!is_square(b2))
```

```
    {
        ++a;
        b2 = a * a - N;
    }

    return a - integral_sqrt(b2);
}

class PrimeFactorizer
{
public:
    explicit PrimeFactorizer(ll primes_up_to = 1000000) : m_upto(primes_up_to)
    {
        eratosthenes_sieve(m_upto);
    }

    // Number of primes
    ll size() const { return primes.size(); }

    // Calculated all primes up to
    ll up_to() const { return m_upto; }

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

        ll largest = primes.back();
        if (p <= largest*largest)
            return bf_is_prime(p);

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

    auto begin() const { return primes.begin(); }
    auto end() const { return primes.end(); }
    ll operator[](ll index) const { return primes[index]; }

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

    /// Make sure sqrt(n) <
    /// primes.back()^2, otherwise
    /// this could spit out a wrong factorization.
    Factorization prime_factorization(ll n) const
```

```
    {
        Factorization F;

        if (n <= 1)
            return F;

        for (auto p : primes)
        {
            ll a = 0;
            auto qr = std::div(n, p);

            while (qr.rem == 0)
            {
                n = qr.quot;
                qr = std::div(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)
    {
        // primecharfunc[a] == true means 2*a+1 is prime
        std::vector<bool> primecharfunc = {false};
        primecharfunc.resize(n/2 + 1, true);

        primes.reserve((1.1 * n) / std::log(n) + 10); // can remove this line

        ll i = 1;
        ll p = 3; // p = 2*i + 1
        for ( ; p*p <= n; ++i, p += 2)
        {
            if (primecharfunc[i])
```

```
            {
                primes.emplace_back(p);
                for (ll j = i+p; j < primecharfunc.size(); j += p)
                    primecharfunc[j] = false;
            }
        }

        for ( ; p < n; p += 2,++i)
        {
            if (primecharfunc[i])
                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:
    ll m_upto;
    std::vector<ll> primes = {2};

    bool bf_is_prime(ll n) const
    {
        for (auto p : primes)
        {
```

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

            if (n % p == 0)
                return false;
        }

        return true;
    }
}; // end class PrimeFactorizer

class EulerPhi
{
public:
    EulerPhi(const PrimeFactorizer& P) : m_phi(P.up_to())
    {
        m_phi[0] = 0;
        m_phi[1] = 1;
        dfs_helper(P, 1, 0);
    }

    // TODO(mraggi): only works if already calculated. Do something else if not.
    ll operator()(ll k)
    {
        if (k < size())
            return m_phi[k];
    }

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

private:
    void dfs_helper(const PrimeFactorizer& P, ll a, ll i)
    {
        ll n = m_phi.size();
        for (; i < P.size() && P[i] * a < n; ++i)
        {
            ll p = P[i];
            ll multiplier = p - 1;
            if (a % p == 0)
                multiplier = p;
            m_phi[p * a] = multiplier * m_phi[a];
            dfs_helper(P, p * a, i);
        }
    }
```

```
    std::vector<ll> m_phi;
};

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
```