MAXimal

Efficient algorithms for factorization

Cont

Here are the implementation of several factorization algorithms, each of which individually may not work as quickly or very slowly, but together th method.

Descriptions of these methods are given, the more that they are well described on the Internet.

Method Pollard p-1

Probabilistic test quickly gives the answer is not for all properties.

Returns either found divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_p_1 (T n)
{
        // Algorithm parameters significantly affect the performance and the quality of search
        const T b = 13;
        const T q [] = {2, 3, 5, 7, 11, 13};

        // Several attempts algorithm
        T a = 5% n;
        for (int j = 0; j <10; j ++)
        {

                // Looking for is a, which is relatively prime to n
                while (gcd (a, n)! = 1)
                {
                        mulmod (a, a, n);
                        a + = 3;
                        a% = n;
                }

                // Calculate a ^ M
                for (size_t i = 0; i <sizeof q / sizeof q [0]; i ++)
                {
                        T qq = q [i];
                        T e = (T) floor (log ((double) b) / log ((double) qq));
                        T aa = powmod (a, powmod (qq, e, n), n);
                        if (aa == 0)
                                continue;

                        // Check not found the answer
                        T g = gcd (aa-1, n);
                        if (1 <g && g <n)
                                return g;
                }

        }

        // If nothing found
        return 1;

}
```

Pollard's method "Ro"

Probabilistic test quickly gives the answer is not for all properties.

Returns either found divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_rho (T n, unsigned iterations_count = 100000)
{
        T
                b0 = rand ()% n,
```

```
                b1 = b0,
                g;
        mulmod (b1, b1, n);
        if (++ b1 == n)
                b1 = 0;
        g = gcd (abs (b1 - b0), n);
        for (unsigned count = 0; count <iterations_count && (g == 1 || g == n); count ++)
        {
                mulmod (b0, b0, n);
                if (++ b0 == n)
                        b0 = 0;
                mulmod (b1, b1, n);
                ++ B1;
                mulmod (b1, b1, n);
                if (++ b1 == n)
                        b1 = 0;
                g = gcd (abs (b1 - b0), n);
        }
        return g;
}
```

Bent method (modification of Pollard "Ro")

Probabilistic test quickly gives the answer is not for all properties.

Returns either found divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_bent (T n, unsigned iterations_count = 19)
{
        T
                b0 = rand ()% n,
                b1 = (b0 * b0 + 2)% n,
                a = b1;
        for (unsigned iteration = 0, series_len = 1; iteration <iterations_count; iteration ++, s
        {
                T g = gcd (b1-b0, n);
                for (unsigned len = 0; len <series_len && (g == 1 && g == n); len ++)
                {
                        b1 = (b1 * b1 + 2)% n;
                        g = gcd (abs (b1-b0), n);
                }
                b0 = a;
                a = b1;
                if (g! = 1 && g! = n)
                        return g;
        }
        return 1;
}
```

Pollard's method of Monte Carlo

Probabilistic test quickly gives the answer is not for all properties.

Returns either found divider, or 1 if the divisor was not found.

```
template <class T>
T pollard_monte_carlo (T n, unsigned m = 100)
{
        T b = rand ()% (m-2) + 2;

        static std :: vector <T> primes;
        static T m_max;
        if (primes.empty ())
                primes.push_back (3);
        if (m_max <m)
        {
                m_max = m;
                for (T prime = 5; prime <= m; ++++ prime)
                {
                        bool is_prime = true;
                        for (std :: vector <T> :: const_iterator iter = primes.begin (), end = pr
                                iter! = end; ++ Iter)
                        {
                                T div = * iter;
                                if (div * div> prime)
                                        break;
                                if (prime% div == 0)
                                {
                                        is_prime = false;
                                        break;
                                }
                        }
                        if (is_prime)
                                primes.push_back (prime);
```

```
                }
        }

        T g = 1;
        for (size_t i = 0; i <primes.size () && g == 1; i ++)
        {
                T cur = primes [i];
                while (cur <= n)
                        cur * = primes [i];
                cur / = primes [i];
                b = powmod (b, cur, n);
                g = gcd (abs (b-1), n);
                if (g == n)
                        g = 1;
        }

        return g;
}
```

Method Farm

This wide method, but it can be very slow if the number is small divisors.

Therefore, it should run only after all other methods.

```
template <class T, class T2>
T ferma (const T & n, T2 unused)
{
        T2
                x = sq_root (n),
                y = 0,
                r = x * x - y * y - n;
        for (;;)
                if (r == 0)
                        return x! = y? xy: x + y;
                else
                        if (r> 0)
                        {
                                r - = y + y + 1;
                                ++ Y;
                        }
                        else
                        {
                                r + = x + x + 1;
                                ++ X;
                        }
}
```

Trivial division

This basic method is useful to immediately handle numbers with very small divisors.

```
template <class T, class T2>
T2 prime_div_trivial (const T & n, T2 m)
{

        // First check the trivial cases
        if (n == 2 || n == 3)
                return 1;
        if (n <2)
                return 0;
        if (even (n))
                return 2;

        // Generate a simple 3 to m
        T2 pi;
        const vector <T2> & primes = get_primes (m, pi);

        // Divisible by all prime
        for (std :: vector <T2> :: const_iterator iter = primes.begin (), end = primes.end ();
                iter! = end; ++ Iter)
        {
                const T2 & div = * iter;
                if (div * div> n)
                        break;
                else
                        if (n% div == 0)
                                return div;
        }

        if (n <m * m)
                return 1;
        return 0;

}
```

Putting it all together

Combine all the methods in the same function.

Also, the function uses the simplicity of the test, otherwise the factorization algorithms can work for very long. For example, you can select a test on BPSW).

```
template <class T, class T2>
void factorize (const T & n, std :: map <T, unsigned> & result, T2 unused)
{
        if (n == 1)
                ;
        else
                // Check whether the number is not prime
                if (isprime (n))
                        ++ Result [n];
                else
                        // If the number is small enough that it expand the simple search
                        if (n <1000 * 1000)
                        {
                                T div = prime_div_trivial (n, 1000);
                                ++ Result [div];
                                factorize (n / div, result, unused);
                        }
                        else
                        {
                                // Number of large, run it factorization algorithms
                                T div;
                                // First go fast algorithms Pollard
                                div = pollard_monte_carlo (n);
                                if (div == 1)
                                        div = pollard_rho (n);
                                if (div == 1)
                                        div = pollard_p_1 (n);
                                if (div == 1)
                                        div = pollard_bent (n);
                                // Will run 100% algorithm Farm
                                if (div == 1)
                                        div = ferma (n, unused);
                                // Recursively Point Multipliers
                                factorize (div, result, unused);
                                factorize (n / div, result, unused);
                        }
}
```

Appendix

Download [5k] source program that uses all of these methods and test factorization BPSW on simplicity.