Survey of Classical Integer Factorization Techniques

Yongwhan Lim

March 22, 2012

Contents

1 Introduction **2**

2 Deterministic Algorithms **3**

2.1 Trial Division 3

2.1.1 Basic Version − $O(n)$ 4

2.1.2 Refined Version I − $O(\sqrt{n})$ 5

2.1.3 Refined Version II − $O\left(\frac{\sqrt{n}}{\ln n}\right)$ 6

2.2 Fermat's 7

2.3 Pollard's ρ 9

2.4 Pollard's $p-1$ 10

2.5 Williams' $p+1$ 12

2.6 Dixon's 13

2.7 Quadratic Sieve (QS) 16

2.7.1 Basic Version 16

2.7.2 Refined Version − The Block Lanczos 19

3 Probabilistic Algorithms **21**

3.1 Fermat 21

3.2 Miller-Rabin 22

3.3 Solovay-Strassen 23

4 Experiments **24**

5 Conclusion **41**

6 Appendix **42**
6.1 Checking Validity of the Input . . . 42
6.2 Time Related Helper Functions . . . 42
6.2.1 Measuring Time . . . 42
6.2.2 Formatting Time . . . 43
6.3 Infinite Precision Techniques . . . 43
6.4 Tables . . . 46
6.4.1 Primes up to 1000 . . . 46
6.4.2 $\pi(n)$ for $n = 10^k$ where $k = 1, \cdots, 10$ . . . 46
6.5 Starting few lines... . . . 47
6.6 Using the code... . . . 48

7 Acknowledgement **48**

8 Reference **48**

1 Introduction

One of the main problems that computational number theory is concerned with is the integer factorization. In this paper, the basic results in integer factorizations/primality tests, both deterministic and probabilistic, are described − we provide the algorithm and corresponding implementation in C++; for some of the algorithms, we will address in what ways the code can be improved and, if such improvement can be achieved, the refined version(s) are also provided. Moreover, we provide the space complexity, the heuristic time-complexity, and the explicit run-time for several sample sample experiments; the author hopes that the reader will get a good grasp for how each algorithm does in practice after reading this paper.

Among the deterministic algorithms, we are going to look at the following algorithms in detail:

1. Trial Division
2. Fermat's
3. Pollard's ρ
4. Pollard's $p-1$
5. Williams' $p+1$
6. Dixon's
7. Quadratic Sieve

Once we treat the deterministic algorithms, we will move on to some of the well-known probabilistic approach for primality tests, which outputs either the given number is 'composite' or 'probably prime':

1. Fermat
2. Miller-Rabin
3. Solovay-Strassen

The exposition is as self-contained as possible − for each of the above algorithms, if some background knowledge is required, we will provide it within that section. We will, however, assume that the reader has basic understanding of the classical number theory and basic familiarity with the C++ *Standard Template Library* (STL).

Before exploring the algorithms, we begin by showing the following easy observation, which forms the basis of our paper:

Lemma 1. *it suffices to find a prime factor for an odd positive integer.*

Proof. Say that we are given some arbitrary integer n. Obviously, we can assume it is positive integer, for negative integer can be written as $-1 \cdot (-n)$ where $-n$ is positive integer and 0 has a trivial factorization. Also, since n can be written as $2^k \cdot m$ where k is the largest exponent of 2 such that 2^k divides n and m is, thus, an odd number, it suffices to consider how to factor an odd number into primes. Finally, it suffices to give an algorithm for finding just one prime factor for an odd number n since once we have $n = p \cdot m$ where p is a prime, we can repeatedly use the algorithm for m (alternatively, we can do this in distributed manner by finding some odd factor a for $n = ab$ and 'divide-and-conquer' for a and b separately and combine the result after; in fact, combining this idea with caching is the best way to treat a large positive integer). □

Using this idea, in what follows, we provide an algorithm for finding a prime factor p for an odd positive integer n.

2 Deterministic Algorithms

There are many well-known deterministic integer factorization algorithms − of these, we are going to treat trial division, Fermat's, Pollard's, Williams', Dixon's, and Quadratic Sieve. We are going to omit elliptic curve, continued fraction, general number field sieve, and Shor's; see [1] for the reference to these algorithms.

2.1 Trial Division

Of all the factorization algorithm, the trial division is the simplest of all − the idea is to divide by all the numbers less than the number.

2.1.1 Basic Version – $O(n)$

In the simplest version, we divide n by all the numbers from 2 to n and, if the number is divisible by any of the number m, we return m and apply the same algorithm for $n \leftarrow n/m$. Of course, this is a linear time algorithm and quite inefficient. Here is the code that implements this basic idea:

```
void factor_trial_naive_linear(ll n) {
        npfact=0;
        bool ok=false, toolarge=false;
        for (ll d=2; d<=n; d++) {
                if(d>MAX_INT) { toolarge=true; break; }
                while(1) {
                        if(n%d) break;
                        if(ok) cout << "x";
                        ok=true;
                        cout << d; n/=d;
                        npfact++;
                }
        }
        if(!toolarge) {
                if(n>1) {
                        if(ok) cout << "x";
                        cout << n;
                        npfact++; n=1;
                }
        }
        if(n==1) {
                if (npfact==1) cout << " (PRIME)";
                else cout << " (COMPOSITE)";
        } else cout << " (TLE)";
        cout << endl;
}
```

Here, `ll` is `long long` in short – it can treat numbers at most $9,223,372,036,854,775,807$; to use this abbreviation, we need the following line at the very top of the code:

```
typedef long long ll;
```

To avoid the algorithm from running forever, we define `MAX_INT` to be 10^9 typically so that it returns TLE (time limit exceeded) if the factorization cannot be done fast enough, because n is larger than `MAX_INT`. This algorithm has time complexity $O(n)$ to find one factor in the worst case (which precisely occurs when n is a prime) and space complexity $O(1)$.

2.1.2 Refined Version I – $O(\sqrt{n})$

Now, this linear algorithm can trivially be improved to square-root. To see this we write $n = ab$ where a and b are non-unit positive integers. Note that we can impose an ordering on a and b – say $1 < a < b$. Then, we see that $a^2 \leq ab = n$, or $a \leq \sqrt{n}$. This means that when m is a non-trivial factor of n then there always exist a factor l of n such that $l \leq \sqrt{n}$. So, to check whether there is a non-trivial factor for n, it suffices to check whether there is a factor at most $\sqrt{n}$. With this idea in mind, we have a following refined version:

```
void factor_trial_naive_sqrt(ll n) {
        npfact=0;
        bool ok=false, toolarge=false;
        for (ll d=2; d*d<=n; d++) {
                if(d>MAX_INT) { toolarge=true; break; }
                while(1) {
                        if(n%d) break;
                        if(ok) cout << "x";
                        ok=true;
                        cout << d; n/=d;
                        npfact++;
                }
        }
        if(!toolarge) {
                if(n>1) {
                        if(ok) cout << "x";
                        cout << n;
                        npfact++; n=1;
                }
        }
        if(n==1) {
                if (npfact==1) cout << " (PRIME)";
                else cout << " (COMPOSITE)";
        } else cout << " (TLE)";
        cout << endl;
}
```

Nothing really changed except the bound for the factor d; instead of $d \leq n$, we now have $d \cdot d \leq n$; we do this instead of $d \leq \sqrt{n}$ to avoid floating-point error in C++. This is a standard trick.

2.1.3 Refined Version II – $O\left(\frac{\sqrt{n}}{\ln n}\right)$

Now, note that this algorithm can be optimized even more! From the *Fundamental Theorem of Arithmetic*, n can be written as $\prod p_i^{e_i}$ where p_i's are primes and e_i's are positive integers. To find a factor for n, therefore, it suffices to find a *prime* that divides n. So, instead of dividing by all the integers from 2 to $\sqrt{n}$, we only consider only the prime numbers in the range. Say that $\pi(x)$ be the number of prime numbers up to x. Then, by the *Prime Number Theorem*, we have

$$\pi(x) \approx \frac{x}{\ln x}.$$

Hence, roughly, we only need to check $O\left(\frac{\sqrt{n}}{\ln n}\right)$ numbers, instead of $O(\sqrt{n})$; this is a factor of $\ln n$ improvement. To make this to work, we only need to provide an algorithm that computes prime numbers within a range efficiently; then, we can pre-compute the prime numbers and cache it, to be used later in this algorithm.

There are few ways to treat computing prime numbers. Of those, *sieve of Eratosthenes* is quite a popular choice, due to its simplicity. The most basic version proceeds as follows. We initially have a table containing 2 to n that are all initially mark on. We mark 2 to be prime and skip. We mark off all the multiples of 2. Then, we move to the next marked entry in the table. We mark that number as a prime, skip, and repeat until we reach $\sqrt{n}$. All the marked numbers after $\sqrt{n}$ are primes. All the marked numbers from 2 to n are primes and the rest are composite. The proof of the correctness is trivial by construction. The implementation of this algorithm follows:

```
void sieve_era() {
        for (ll i=0; i<MAX_INT; i++) is_prime[i]=true;
        is_prime[0]=is_prime[1]=false;
        for (ll i=2; i*i<MAX_INT; i++)
                if(is_prime[i])
                for (ll j=i; j*i<MAX_INT; j++)
                        is_prime[j*i]=false;
        for (ll i=0; i<MAX_INT; i++)
                if(is_prime[i])
                        p.push_back(i);
        nprime=p.size();
}
```

Here, `is_prime` is the table of markers (represented as booleans) and p will contain all the prime numbers from 1 to `MAX_INT`. Since prime harmonic series asymtotically approaches $\ln \ln n$, the run-time of this algorithm can be shown to be $O(\sqrt{n} \cdot \frac{\ln \ln n}{\ln n})$.

Using these two pre-computed entries, `is_prime` and p, we can implement the refined version of the algorithm as follows:

```
void factor_trial_sieve(ll n) {
        if(n<MAX_INT && is_prime[n]) cout << n << " (PRIME)" << endl;
        else {
                npfact=0;
                bool ok=false;
                for (int i=0; i<nprime; i++) {
                        ll d=p[i];
                        if(d*d>n) break;
                        while(1) {
                                if(n%d) break;
                                if(ok) cout << "x";
                                ok=true;
                                cout << d; n/=d;
                                npfact++;
                        }
                }
                if(n>1) {
                        if(ok) cout << "x";
                        cout << n;
                        npfact++;
                }
                if (npfact==1) cout << " (PRIME)";
                else cout << " (COMPOSITE)";
                cout << endl;
        }
}
```

So, when n is a prime within the range `MAX_INT`, we return that it is prime right away, in $O(1)$, using the pre-computed table `is_prime`. Otherwise, we try dividing by all the primes at most $\sqrt{n}$. This takes $O\left(\frac{\sqrt{n}}{\ln n}\right)$ time, since there are about that many prime numbers at most $\sqrt{n}$. Without too much work, this would be just about the best we can do when we are trying to optimize the trivial division as much as possible.

2.2 Fermat's

The idea behind Fermat's algorithm is simple − that an odd integer n can be written as a difference of two sqaures. To see this, we write n as $a^2 - b^2 = (a-b)(a+b)$. Now, say n can be factored as pq where $p > q$. Then, we can set

$$a = \frac{1}{2}(p+q) \text{ and } b = \frac{1}{2}(p-q).$$

Note that each of these are integers as p and q must both be odd, which follows as $n = pq$ is odd.

So, our goal is to find a such that $b^2 = a^2 - n$ is a square. If such a exists, we can factor n; otherwise, the algorithm fails. We start from a smallest possible a that can yield a non-negative number; that is, $a = \lceil\sqrt{n}\rceil$. The following is the implementation of this idea:

```
void factor_fermat(ll n) {
        npfact=0;
        bool ok=false;
        while(1) {
                if(n%2) break;
                if(ok) cout << "x";
                ok=true;
                cout << 2; n/=2;
                npfact++;
        }
        if(n>=MAX_FERMAT*MAX_FERMAT) { cout << " (OUT OF RANGE)" << endl; return; }
        if (n>1) {
                queue<ll> q; q.push(n);
                vector<ll> factor;
                while(!q.empty()) {
                        ll cur=q.front(), iter=0, a=ll(ceil(sqrt(cur))), b, b2=a*a-cur;
                        while(1) {
                                if(iter>MAX_FERMAT) { cout << " (TLE)" << endl; return;
                                b=ll(sqrt(b2)+EPS);
                                if(b*b==b2) break;
                                a++; b2=a*a-cur; iter++;
                        }
                        ll d=(a-b), e=(a+b);
                        if(d!=1) { q.push(d); q.push(e); }
                        else factor.push_back(e);
                }
                sort(factor.begin(), factor.end());
                int ct=factor.size();
                for (int i=0; i<ct; i++) {
                        if(ok) cout << "x";
                        ok=true;
                        cout << factor[i];
                }
                npfact+=ct;
        }

        if (npfact==1) cout << " (PRIME)";
        else cout << " (COMPOSITE)";
        cout << endl;
}
```

Note that EPS is chosen to be a small real number, typically 10e-6, so as to avoid a floating point error. MAX_FERMAT is there as a standard trick. Basically, we repeat, using the idea we had in the introduction − factor has all the factors of n that are found to completely factorize n; note that some of the factors may not be prime, because the algorithm may fail to find a prime factor for a composite number. In the output, we sort the factors in ascending order.

2.3 Pollard's ρ

This algorithm is inspired by the *Floyd's Cycle-Finding Algorithm.* We have two numbers x and y initialized to 2 and divisor d. The idea is to apply a pseudo-random function f to x and y, 1 and 2 per iteration; that is $x \leftarrow f(x)$ and $y \leftarrow f(f(y))$. The popular choice for f is just quadratic $x^2 + c$ where $c \neq 0, -2$. Then, we set $d \leftarrow \gcd(|x-y|, n)$. If d is ever n then we return fail; otherwise, we return d. The iteration stops when $d \neq 1$. We can visualize this as a graph of (x, y) where node is connected if and only if the operation above send it there. Then, in finite number of steps, the algorithm terminate because there are only n^2 points in the lattice. The greatest common divisor is obtained from the Euclidean algorithm. We have the following simple implementation:

```
void factor_pollard_rho(ll n) {
        vector<ll> factor;
        queue<ll> q; q.push(n);
        while(!q.empty()) {
                ll cur=q.front(); q.pop();
                bool ok=false;
                ll x,y,d;
                for (int i=0; i<MAX_ITER; i++) {
                        ll c=rand()%(cur-1)+1;
                        x=2, y=2, d=1;
                        while(d==1) {
                                if(x>=MAX_INT || y>=MAX_INT || c>=MAX_INT) break;
                                x=f(x,cur,c);
                                y=f(f(y,cur,c),cur,c);
                                d=gcd(abs(x-y),cur);
                        }
                        if(1<d && d<cur) { ok=true; break;}
                }
                if(ok) { q.push(d); q.push(cur/d); }
                else factor.push_back(cur);
        }
        sort(factor.begin(), factor.end());
        int sz=factor.size();
        for (int i=0; i<sz; i++) {
                if(i) cout << "x";
                cout << factor[i];
        }
        cout << endl;
}
```

where `gcd` is obtained from Euclidean algorithm:

```
ll gcd(ll a, ll b) {
        if(!b) return a;
        else return gcd(b, a%b);
}
```

`mulmod` gets $ab \pmod{c}$:

```
ll mulmod(ll a, ll b, ll c) {
        ll x=0, y=a%c;
        while(b>0) {
                if(b%2) x=(x+y)%c;
                y=(y*2)%c;
                b/=2;
        }
        return (x%c);
}
```

`f` is just the pseudo-random function:

```
ll f(ll x, ll n, ll c) {
        return (mulmod(x,x,n)+c)%n;
}
```

Since some c's may fail to produce any non-trivial factor, we iterate the algorithm for different choices of c; `factor` is the same as before.

2.4 Pollard's $p-1$

Let n be a composite integer with prime factor p. Then, applying *Fermat's Little Theorem*, we have, where a is coprime to p, $a^{m(p-1)} \equiv 1 \pmod{p}$ where m is a positive integer. The idea is to make the exponent a large multiple of $p-1$ by making it a number with many prime factors. To be more precise, we choose a smooth bound B and let $M = \prod_{p \le B} p^{\lfloor \log_q B \rfloor}$. Fix a coprime to n and let g be $\gcd(a^M - 1, n)$. Now, return g if $1 < g < n$ but iterate otherwise. The below is the implementation of this idea:

```
void factor_pminus(ll n) {
        vector<ll> factor;
        queue<ll> q; q.push(n);
        while(!q.empty()) {
                ll val=q.front(); q.pop();
                ll B=MIN_B, iter=0, g;
                bool ok=false;
                while(iter<MAX_ITER_SMALL) {
                        B=rand()%(MAX_B-B)+B; g=val;
                        ll a = rand()%(val-1)+1, res=1;
                        for (int i=0; i<nprime; i++) {
                                ll cur=1, q=p[i];
                                if(q>B) break;
                                while(1) {
                                        if(cur*q>B) break;
                                        cur*=q;
                                }
                                res=(res*mod(a,cur,val))%val;
                        }
                        res=(res+val-1)%val;
                        g=gcd(res,val);
                        if(1<g && g<val) { ok=true; break; }
                        iter++;
                }
                if(ok) { q.push(g); q.push(val/g); }
                else factor.push_back(val);
        }
        sort(factor.begin(), factor.end());
        int sz=factor.size();
        for (int i=0; i<sz; i++) {
                if(i) cout << "x";
                cout << factor[i];
        }
        cout << endl;
}
```

where `mod(a,b,c)` computes $a^b \pmod{c}$:

```
ll mod(ll a, ll b, ll c) {
        ll r=1;
        while(b) {
                if(b%2) r=(r*a)%c;
                a=(a*a)%c;
                b/=2;
        }
        return r;
}
```

2.5 Williams' $p + 1$

This algorithm works by picking the Lucas sequence $x_0 \equiv 2 \pmod{n}$, $x_1 \equiv b \pmod{n}$, $x_k \equiv bx_{k-1} - x_{k-2} \pmod{n}$ for all $k \geq 2$. Then for each k, we calculate $g = \gcd(n, x_k - 2)$. If $1 < g < n$ then we found non-trivial factor of n but repeat otherwise. Now, to make this algorithm run faster, we can replace each i by $i!$. The algorithm below is the efficient implementation of the factorial version. Note that this algorithm works fairly well but most of the time, not too fast:

```
void factor_pplus(ll n) {
        vector<ll> factor;
        queue<ll> q; q.push(n);
        while(!q.empty()) {
                ll cur=q.front(); q.pop();
                bool ok=false;
                ll iter=0, g=cur;
                while(iter<MAX_ITER_SMALL) {
                        ll b=rand()%(MAX_B-MIN_B)+MIN_B, x, y;
                        for (ll m=1; m<MAX_ITER; m++) {
                                x=b;
                                y=(mulmod(b,b,cur)+cur-2)%cur;
                                ll val=1;
                                for (ll j=0; j<62; j++) {
                                        if(val*2LL>m) break;
                                        if(val&m) {
                                                x=(mulmod(x,y,cur)+(cur-b))%cur;
                                                y=(mulmod(y,y,cur)+(cur-2))%cur;
                                        } else {
                                                y=(mulmod(x,y,cur)+(cur-b))%cur;
                                                x=(mulmod(x,x,cur)+(cur-2))%cur;
                                        }
                                        val<<=(1LL);
                                }
                                b=x;
                                g=gcd((b+(cur-2))%cur, cur);
                                if(1<g&&g<cur) { ok=true; break; }
                        }
                        if(ok) break;
                        iter++;
                }
                if(ok) { q.push(g); q.push(cur/g); }
                else factor.push_back(cur);
        }
        sort(factor.begin(), factor.end());
        int sz=factor.size();
        for (int i=0; i<sz; i++) {
                if(i) cout << "x";
                cout << factor[i];
        }
        if(sz==1) cout << " (PRIME)";
        else cout << " (COMPOSITE)";
        cout << endl;
}
```

2.6 Dixon's

The Dixon's algorithm is closely related to the quadratic sieve; it is, in fact, a precursor. It works as follows.

1. Choose a smoothness bound B.
2. Find $\pi(B)+1$ numbers a_i such that $b_i \equiv a_i^2 \pmod{n}$ is B-smooth.
3. Factor b_i and generate exponent vectors for each one in $\mathbb{F}_2$.
4. Use linear algebra to find a subset that adds to zero vector.
5. We form $a^2 \equiv b^2 \pmod{n}$ where a and b are formed by multiplying elements from the subset.
6. We have $(a+b)(a-b) \equiv 0 \pmod{n}$. So, check whether $\gcd(a+b,n)$ or $\gcd(a-b,n)$ are non-trivial factors of n. If it is, return it but otherwise repeat with different subset, higher smoothness bound, or more a_i's.

Here is the implementation of Dixon's method:

```
void factor_dixon(ll n) {
        vector<ll> factor;
        queue<ll> q; q.push(n);
        ll x,y;
        while(!q.empty()) {
                ll cur=q.front(); q.pop();
                if(factorable_dixon(cur,x,y)) { q.push(x); q.push(y); }
                else factor.push_back(cur);
        }
        int sz=factor.size();
        sort(factor.begin(), factor.end());
        for (int i=0; i<sz; i++) {
                if(i) cout << "x";
                cout << factor[i];
        }
        if(sz==1) cout << " (PRIME)";
        else cout << " (COMPOSITE)";
        cout << endl;
}
```

where `factorable_dixon` is:

```
bool factorable_dixon(ll n, ll &xx, ll &yy) {
        ll B = min(n, max(MIN_B_QS, get_B(n)));
        vector<ll> v;
        for (int i=0; i<=B; i++)
                if(is_prime[i]) v.push_back(i);
        int sz=v.size(), nrow=ll(sz+EPS_MAT), ncol=sz;
        vector<ll> zo, zsq;
        ll z=ll(sqrt(n))+1;
        int idx=0, iter=0;
        while(1) {
                if(idx>=nrow) break;
                if(iter>=MAX_ITER_QS) { nrow=idx; break; }
                ll cur=(z*z)%n;
                if(isSmooth(cur,v)) { procSmooth(cur,v,idx); zo.push_back(z); zsq.push_b
                z++; iter++;
        }

        for (int i=0; i<nrow; i++)
                for (int j=0; j<nrow; j++) {
                        if(i==j) id_mod2[i][j]=1;
                        else id_mod2[i][j]=0;
                }

        gauss_elim_mod2(grid_mod2,nrow,ncol);

        for (int i=0; i<nrow; i++) {
                bool ok=true;
                for (int j=0; j<ncol; j++) if(grid_mod2[i][j]) { ok=false; break; }
                if(ok) {
                        ll x=1,y=1;
                        for (int j=0; j<ncol; j++) pexp[j]=0;
                        for (int j=0; j<nrow; j++)
                                if(id_mod2[i][j]) {
                                        x=mulmod(x,zo[j],n);
                                        for (int k=0; k<ncol; k++) pexp[k]+=grid[j][k];
                                }
                        for (int j=0; j<ncol; j++) { pexp[j]/=2; y*=mod(v[j],pexp[j],n);
                        ll g=gcd((x+y)%n, n);
                        if(1<g && g<n) { xx=g; yy=n/g; return true; }
                }
        }
        return false;
}
```

where the auxliary functions are:

```
ll get_B(ll n) { return ll(MUL_FACTOR * sqrt( exp( sqrt( log(n) * log( log(n) ) ) ) ) )

bool isSmooth(ll n, vector<ll> & pp) {
        int sz=pp.size();
        for (int i=0; i<sz; i++) {
                ll d=pp[i]; if(d>n) break;
                while(1) {
                        if(n%d) break;
                        n/=d;
                }
        }
        return (n==1);
}

void procSmooth(ll n, vector<ll> & pp, int idx) {
        int sz=pp.size();
        for (int i=0; i<sz; i++) {
                ll d=pp[i];
                int cur=0;
                while(1) {
                        if(n%2) break;
                        n/=d; cur++;
                }
                grid[idx][i]=cur;
                grid_mod2[idx][i]=cur%2;
        }
}

void gauss_elim_mod2(int grid_mod2[SIZE_FACTOR_BASE][SIZE_FACTOR_BASE], int nrow, int nc
        for (int i=0; i<nrow; i++) {
                int i_max=i;
                for (int ii=i; ii<nrow; ii++)
                        if(grid_mod2[ii][i]==1) { i_max=ii; break; }

                if(i_max!=i) {
                        for (int j=0; j<ncol; j++) swap(grid_mod2[i][j],grid_mod2[i_max]
                        for (int j=0; j<nrow; j++) swap(id_mod2[i][j],id_mod2[i_max][j])
                }

                for (int ii=i+1; ii<nrow; ii++)
                        if(grid_mod2[ii][i]) {
                                for (int jj=i; jj<ncol; jj++) grid_mod2[ii][jj]^=grid_mo
                                for (int jj=0; jj<nrow; jj++) id_mod2[ii][jj]^=id_mod2[i
                        }
        }
}
```

2.7 Quadratic Sieve (QS)

Here, we are going to look at the quadratic sieve. We will first look at the basic version and replace the Gaussian elimination step by the block Lanczos method to get a refined version.

2.7.1 Basic Version

In the basic QS, we replace step 2 of Dixon's method by sieving.

1. Choose a smoothness bound B.
2. Use sieve to find $\pi(B)+1$ numbers a_i such that $b_i \equiv a_i^2 \pmod{n}$ is B-smooth.
3. Factor b_i and generate exponent vectors for each one in $\mathbb{F}_2$.
4. Use linear algebra to find a subset that adds to zero vector.
5. We form $a^2 \equiv b^2 \pmod{n}$ where a and b are formed by multiplying elements from the subset.
6. We have $(a+b)(a-b) \equiv 0 \pmod{n}$. So, check whether $\gcd(a+b, n)$ or $\gcd(a-b, n)$ are non-trivial factors of n. If it is, return it but otherwise repeat with different subset, higher smoothness bound, or more a_i's.

Here is the implementation of the above idea – this strictly follows *Section 6.1.2. Basic QS: A Summary* from [1, page 266]:

```
void factor_qsieve_naive(ll n) {
        vector<ll> factor;
        queue<ll> q; q.push(n);
        ll x,y;
        while(!q.empty()) {
                ll cur=q.front(); q.pop();
                if(factorable_qsieve_naive(cur,x,y)) { q.push(x); q.push(y); }
                else factor.push_back(cur);
        }
        int sz=factor.size();
        sort(factor.begin(), factor.end());
        for (int i=0; i<sz; i++) {
                if(i) cout << "x";
                cout << factor[i];
        }
        if(sz==1) cout << " (PRIME)";
        else cout << " (COMPOSITE)";
        cout << endl;
}
```

where `factorable_qsieve_naive` is just:

```
bool factorable_qsieve_naive(ll n, ll &xxx, ll &yyy) {
        // 1. Initalization
        ll B = min(n, max(MIN_B_QS, get_B(n)));
        vector<ll> v,w;
        v.push_back(2);
        for (int i=0; i<=B; i++)
                if(is_prime[i] && jac(i,n)==1) v.push_back(i);
        for (int i=0; i<=B; i++)
                if(is_prime[i]) w.push_back(i);

        // 2. Sieving
        int sz=v.size(), nrow=ll(sz+EPS_MAT), ncol=sz;
        vector<ll> xo, xf;
        ll x=ll(sqrt(n))+1;
        int idx=0, iter=0;
        while(1) {
                if(idx>=nrow) break;
                if(iter>=MAX_ITER_QS) { nrow=idx; break; }
                ll cur=f_qs(x,n);
                if(isSmooth(cur,w)) { iter=0; procSmooth(cur,v,idx); xo.push_back(x); xf
                x++; iter++;
        }

        // 3. Linear Algebra -- by Gaussian elimination
        for (int i=0; i<nrow; i++)
                for (int j=0; j<nrow; j++) {
                        if(i==j) id_mod2[i][j]=1;
                        else id_mod2[i][j]=0;
                }
        gauss_elim_mod2(grid_mod2,nrow,ncol);

        // 4. Factorization
        for (int i=0; i<nrow; i++) {
                bool ok=true;
                for (int j=0; j<ncol; j++) if(grid_mod2[i][j]) { ok=false; break; }
                if(ok) {
                        ll xx=1, yy=1;
                        for (int j=0; j<ncol; j++) pexp[j]=0;
                        for (int j=0; j<nrow; j++)
                                if(id_mod2[i][j]) {
                                        xx=mulmod(xx,xo[j],n);
                                        for (int k=0; k<ncol; k++) pexp[k]+=grid[j][k];
                                }
                        for (int j=0; j<ncol; j++) { pexp[j]/=2; yy*=mod(v[j],pexp[j],n)
                        ll g=gcd((xx+yy)%n, n);
                        if(1<g && g<n) { xxx=g; yyy=n/g; return true; }
                }
        }
        return false;
}
```

where `jac`, and `f_qs` are (the others are given in the Section 2.6):

```
int jac(ll a, ll n) {
        if(!a) return 0;
        int ans=1;
        ll t;
        if(a<0) { a=-a; if(n%4==3) ans=-ans; }
        if(ans==1) return ans;
        while(a) {
                if(a<0) { a=-a; if(n%4==3) ans=-ans; }
                while(a%2==0) { a/=2; if(n%8==3 || n%8==5) ans=-ans; }
                swap(a,n);
                if(a%4==3 && n%4==3) ans=-ans;
                a%=n;
                if(a>n/2) a-=n;
        }
        if(n==1) return ans;
        return 0;
}

ll f_qs(ll x, ll n) { return (x*x-n); }
```

Note that the Gaussian elimination can be switched to a faster algorithm, if we make use of the sparsity of the matrix. There are three well-known ways to optimize the Gaussian elimination: the block Lanczos, the conjugate gradient descent, and the coordinate recurrence method. Most of the state-of-the-art system uses the block Lanczos because it is designed for the efficient implementation. We describe how it works and sketch the idea behind the implementation; only the most important portions of the code is provided in this paper, due to its length.

2.7.2 Refined Version – The Block Lanczos

The implementation of the block Lanczos follows the algorithm provided in [2], step by step. Here, I give a brief overview of the algorithm and discuss the implementation details. Say B is $m \times n$ matrix over $\mathbb{F}_2$. Then, it suffices to solve for $Bx = 0$ in the Gaussian elimination step. Since the Lanczos require that B to be symmetric, we compute for $A = B^T B$, which has dimension $n \times n$. Now, let N be the word size of computer, typically 32 or 64. Select random $n \times N$ matrix Y over $\mathbb{F}_2$ and compute AY. It suffices to find $n \times N$ matrix X such that $AX = AY$. Then, the column vectors of $X - Y$ will be the random vectors in the null space of A; so, columns of $X - Y$ can be combined to find vectors in the null space of B, which we can do by the Gaussian elimination (as we did earlier). So, effectively, we need to provide efficient algorithm to compute for X; see [2] for the proof of correctness:

0. Choose random Y.
1. Initialize $V_0 = AY$.
2. Iterate $V_{i+1} = AV_i S_i S_i^T + V_i D_{i+1} + V_{i-1} E_{i+1} + V_{i-2} F_{i+1}$ until $V_i^T A V_i = 0$ for some $i = m$.
3. Convert the result to vectors in null space of A using the Gaussian elimination.
4. Check whether the result is OK; if not, go to step 0 and iterate until maximum number of iteration is reached.

In step 2, to get V_{i+1}, we need to do the following sub-steps:

a1. Pre-compute $V_i^T A V_i$ and $V_i^T A^2 V_i$.

a2. Compute S_i and W_i^{inv} using the algorithm in [2, Section 8].

b1. Compute $D_{i+1} = I_N - W_i^{inv}(V_i^T A^2 V_i S_i S_i^T + V_i^T A V_i)$ [2, Equation 19].

b2. Compute $E_{i+1} = -W_{i-1}^{inv} V_i^T A V_i S_i S_i^T$ [2, Equation 19].

b3. Compute $F_{i+1} = -W_{i-2}^{inv}(I_N - V_{i-1}^T A V_{i-1} W_{i-1}^{inv})(V_{i-1}^T A^2 V_{i-1} S_{i-1} S_{i-1}^T + V_{i-1}^T A V_{i-1}) S_i S_i^T$ [2, Equation 18].

c. Compute $V_{i+1} = A V_i S_i S_i^T + V_i D_{i+1} + V_{i-1} E_{i+1} + V_{i-2} F_{i+1}$ [2, Equation 19].

d. Update $X \leftarrow X + V_i W_i^{inv} V_i^T V_0$ [2, Equation 20].

To get all this done, we don't need more than three layers; so we do 3-layer dynamic programming. Here is the code for the step 2. Note that step 0, 1, 3, and 4 are trivial (step 3 is treated in Section 2.6.1). Like before, we use bit operation whenever possible to speed-up the computation:

```
while(1) {
// step a1
        mnns(size,ncols,B,v[0],g);
        tmnns(ncols,B,g,vi);
        snns(v[0],vi,g,vtav[0],n);
        snns(vi,vi,g,vta2v[0],n);
        for (i=0; i<SIZE; i++) if (vtav[0][i]) break;
        if (i==SIZE) break;
// step a2
        d0=find(vtav[0],s[0],s[1],d1,winv[0]);
        if (!d0) break;
        m0 = 0;
        for (i=0; i<d0; i++) m0|=mask[s[0][i]];
// step b1
        for (i=0; i<SIZE; i++) d[i]=(vta2v[0][i]&m0)^vtav[0][i];
        mul(winv[0],d,d);
        for (i=0; i<SIZE; i++) d[i]^=mask[i];
// step b2
        mul(winv[1],vtav[0],e);
        for (i=0; i<SIZE; i++) e[i]&=m0;
// step b3
        mul(vtav[1],winv[1],f);
        for (i=0; i<SIZE; i++) f[i]^=bitmask[i];
        mul(winv[2],f,f);
        for (i=0; i<SIZE; i++) f2[i]=((vta2v[1][i]&m1)^vtav[1][i])&m0;
        mul(f,f2,f);
// step c
        for (i=0; i<n; i++) vi[i]&=m0;
        nsss(v[0],d,g,vi,n);
        nsss(v[1],e,g,vi,n);
        nsss(v[2],f,g,vi,n);
// step d
        snns(v[0],v0,g,d,n);
        mul(winv[0], d, d);
        nsss(v[0],d,g,x,n);
        swap(v[2],v[1]); swap(v[1],v[0]); swap(v[0],vnxt);
        swap(winv[2],winv[1]); swap(winv[1],winv[0]);
        swap(vtav[1],vtav[0]);
        swap(vta2v[1], vta2v[0]);
        m1=m0; d1=d0;
        memcpy(s[1],s[0],SIZE*sizeof(long));
}
```

where `SIZE` is typically 64. Note that `mnns`, `tmnns`, `snns`, `mul`, and `nsss` are nothing more than multiplication of $a \times b$ matrix by $b \times c$ matrix; each of those are implemented using bit operations. `find` is algorithm for computing for S and W^{inv}.

3 Probabilistic Algorithms

There are three different probabilistic algorithms that we would like to look at − Fermat, Miller-Rabin, and Solovay-Strassen. The term 'probabilistic' means that the algorithm is randomized. Hence, it does not guarantee that the number n is prime, when it outputs 'probably prime'; rather, it just means that the algorithm failed to find that the number is composite.

3.1 Fermat

The *Fermat's Little Theorem* states that if p is prime and $1 \leq a < p$ then $a^{p-1} \equiv 1 \pmod{p}$; if the statement does not hold then p must be composite. So, all we need to do to show that n is composite is, pick some numbers in range 1 to $n-1$ and show $a^{n-1} \equiv 1 \pmod{n}$ does not hold. If the statement holds for some number of iterations, we output n is probably prime, because we cannot find a witness for the compositeness of n. Here is the simple implementation:

```
void isprime_fermat(ll n) {
        if(n==1) { cout << "COMPOSITE" << endl; return; }
        for (int i=0; i<MAX_ITER; i++) {
                ll a=rand()%(n-1)+1;
                if(mod(a,n-1,n)!=1) { cout << "COMPOSITE" << endl; return; }
        }
        cout << "PROBABLY PRIME" << endl; return;
}
```

Here, `mod(a,n-1,n)` gets $a^{n-1} \pmod n$. Also, `rand()` function gets a random generator, a default function from C++. Now, to implement `mod`, note that we can compute a^{n-1} avoiding overflow by representing $n-1$ in base 2 and multiply a^{2^i} that are needed to form a^{n-1}, all of these in mod n; this is faster than the linear algorithm of multiplying a by itself $n-1$ times. The implementation is:

```
ll mod(ll a, ll b, ll c) {
        ll r=1;
        while(b) {
                if(b%2) r=(r*a)%c;
                a=(a*a)%c;
                b/=2;
        }
        return r;
}
```

3.2 Miller-Rabin

We are using the *Fermat's Little Thereom* again. Also, note that if p is prime and $x^2 \equiv 1 \pmod p$ then $x \equiv 1 \pmod p$ or $x \equiv -1 \pmod p$. So, if we write $n-1$ as $2^k \cdot m$. where m is odd and $k \geq 0$ then, as long as n is a prime, $a^m \equiv 1 \pmod n$ or $a^{m \cdot 2^t} \equiv -1 \pmod p$ for some $0 \leq t < d$; otherwise, a^{n-1} cannot be 1 for any prime number a.

So, it suffices to do the following. We write $n-1$ as $2^k \cdot m$ and choose $1 \leq a < n$. For each iteration, fix a and check $a^m \equiv 1 \pmod p$ or $a^{m \cdot 2^t} \equiv -1 \pmod p$ for some t such that $0 \leq t < k$. If both fail then n is definitely a composite. Otherwise, n could be a prime. The implementation is:

```
void isprime_miller_rabin(ll n) {
        if(n<2) { cout << "COMPOSITE" << endl; return; }
        if(n!=2 && n%2==0) { cout << "COMPOSITE" << endl; return; }
        ll s=n-1;
        while(s%2==0) s/=2;
        for (int i=0; i<MAX_ITER; i++) {
                ll a=rand()%(n-1)+1, t=s, m=mod(a,t,n);
                while(t!=n-1 && m!=1 && m!=n-1) { m=mulmod(m,m,n); t*=2; }
                if(m!=n-1 && t%2==0) { cout << "COMPOSITE" << endl; return; }
        }
        cout << "PROBABLY PRIME" << endl; return;
}
```

where `mulmod` is, using the similar logic as `mod`, just:

```
ll mulmod(ll a, ll b, ll c) {
        ll x=0, y=a%c;
        while(b>0) {
                if(b%2) x=(x+y)%c;
                y=(y*2)%c;
                b/=2;
        }
        return (x%c);
}
```

3.3 Solovay-Strassen

The idea here is to use the identity proved by Euler: for an odd prime p and any integer a, we have

$$a^{(p-1)/2} \equiv \left(\frac{a}{p}\right) \pmod{p}.$$

where $(\cdot)$ is the Jacobi symbol.

So, for each iteration, we pick an integer a in range from 1 to $n-1$ and check whether the identity holds. If the identity does not hold, n must be composite. Otherwise, n could be a prime. We have the following trivial implementation:

```
void isprime_solovay_strassen(ll n) {
        if(n<2) { cout << "COMPOSITE" << endl; return; }
        if(n!=2 && n%2==0) { cout << "COMPOSITE" << endl; return; }
        for (int i=0; i<MAX_ITER; i++) {
                ll a=rand()%(n-1)+1, j=(n+jac(a,n))%n, m=mod(a,(n-1)/2,n);
                if(!j || m!=j) { cout << "COMPOSITE" << endl; return; }
        }
        cout << "PROBABLY PRIME" << endl; return;
}
```

where `jac(a,n)` is the jacobi symbol. Using the property of the notation, we have the following implementation of the function:

```
int jac(ll a, ll n) {
        if(!a) return 0;
        int ans=1;
        ll t;
        if(a<0) { a=-a; if(n%4==3) ans=-ans; }
        if(ans==1) return ans;
        while(a) {
                if(a<0) { a=-a; if(n%4==3) ans=-ans; }
                while(a%2==0) { a/=2; if(n%8==3 || n%8==5) ans=-ans; }
                swap(a,n);
                if(a%4==3 && n%4==3) ans=-ans;
                a%=n;
                if(a>n/2) a-=n;
        }
        if(n==1) return ans;
        return 0;
}
```

4 Experiments

When the code starts, it starts by pre-computing the primes up to the limit pre-defined in the beginning as `MAX_INT`. We chose it to be 100000001 for the demonstration. Here are few outputs of the result; note that in the range of long long, it is hard to see the effect of the quadratic sieve; all the traditional algorithms starts to fail around 10^{20}; so, for the purpose of this demonstration, we limit ourselves to relatively small range, up to 10^{16}.

When I type invalid input like r0923859023860 the output is

```
r0923859023860
INVALID INPUT
#######################################
```

When the input is 1, the output is:

```
(1)
TRIVIAL
#######################################
```

Here I provide the result of ≈ 20 runs of valid inputs; the numbers are randomly chosen by hand from the range of long long. Also, see Section 6.6 of *Appendix* to see more details on how the code is executed; $P-1$ is omitted because it is too slow.

```
PRE-PROCESS TIME = 3sec(s) 440ms 498Ms
5761455 primes up to 100000001
```

```
#######################################

(15)
FACTORIZATION
-------------------------
3x5 (COMPOSITE)
FERMAT'S = 21Ms
3x5 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 3Ms
3x5 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 3Ms
3x5 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 3Ms
3x5
POLLARD RHO = 4ms 278Ms
3x5 (COMPOSITE)
P+1 = 1sec(s) 789ms 641Ms
3x5 (COMPOSITE)
DIXON'S = 989Ms
3x5 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 13ms 746Ms
3x5 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 12ms 276Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 3Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(1001)
FACTORIZATION
-------------------------
7x11x13 (COMPOSITE)
FERMAT'S = 9Ms
7x11x13 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 3Ms
7x11x13 (COMPOSITE)
```

```
TRIAL DIVISION -- NAIVE (SQRT) = 3Ms
7x11x13 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 3Ms
7x11x13
POLLARD RHO = 14ms 6Ms
7x11x13 (COMPOSITE)
P+1 = 2sec(s) 614ms 181Ms
7x11x13 (COMPOSITE)
DIXON'S = 3ms 991Ms
7x11x13 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 22ms 392Ms
7x11x13 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 21ms 617Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 3Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(15347)
FACTORIZATION
-------------------------
103x149 (COMPOSITE)
FERMAT'S = 9Ms
103x149 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 5Ms
103x149 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 4Ms
103x149 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 4Ms
103x149
POLLARD RHO = 61ms 105Ms
103x149 (COMPOSITE)
P+1 = 1sec(s) 711ms 403Ms
103x149 (COMPOSITE)
DIXON'S = 188ms 535Ms
103x149 (COMPOSITE)
```

```
QUADRATIC SIEVE -- NAIVE = 334ms 849Ms
103x149 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 333ms 650Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 3Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(84923)
FACTORIZATION
-------------------------
163x521 (COMPOSITE)
FERMAT'S = 13Ms
163x521 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 10Ms
163x521 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 5Ms
163x521 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 4Ms
163x521
POLLARD RHO = 126ms 737Ms
163x521 (COMPOSITE)
P+1 = 1sec(s) 718ms 726Ms
163x521 (COMPOSITE)
DIXON'S = 287ms 130Ms
163x521 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 553ms 737Ms
163x521 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 556ms 591Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 4Ms
```

```
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(10101017)
FACTORIZATION
-------------------------
83x131x929 (COMPOSITE)
FERMAT'S = 55Ms
83x131x929 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 15Ms
83x131x929 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 5Ms
83x131x929 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 4Ms
83x131x929
POLLARD RHO = 187ms 77Ms
83x131x929 (COMPOSITE)
P+1 = 2sec(s) 573ms 755Ms
83x131x929 (COMPOSITE)
DIXON'S = 711ms 336Ms
83x131x929 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 39Ms
83x131x929 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 914ms 49Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 4Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 4Ms
#######################################

(2905821)
FACTORIZATION
-------------------------
3x3x3x281x383 (COMPOSITE)
FERMAT'S = 21Ms
3x3x3x281x383 (COMPOSITE)
```

```
TRIAL DIVISION -- NAIVE (LINEAR) = 8Ms
3x3x3x281x383 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 7Ms
3x3x3x281x383 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 6Ms
3x3x3x281x383
POLLARD RHO = 132ms 906Ms
3x3x3x281x383 (COMPOSITE)
P+1 = 4sec(s) 442ms 173Ms
3x3x3x281x383 (COMPOSITE)
DIXON'S = 1sec(s) 268ms 127Ms
3x3x3x281x383 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 708ms 672Ms
3x3x3x281x383 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 705ms 933Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 5Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(209385025)
FACTORIZATION
-------------------------
5x5x8375401 (COMPOSITE)
FERMAT'S = 115ms 759Ms
5x5x8375401 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 109ms 692Ms
5x5x8375401 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 42Ms
5x5x8375401 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 17Ms
5x5x8375401
POLLARD RHO = 27sec(s) 58ms 892Ms
5x5x8375401 (COMPOSITE)
P+1 = 5sec(s) 339ms 169Ms
5x5x8375401 (COMPOSITE)
```

```
DIXON'S = 1sec(s) 141ms 583Ms
5x5x8375401 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 195ms 828Ms
5x5x8375401 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 189ms 320Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 7Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 2Ms
########################################

(1904847)
FACTORIZATION
-------------------------
3x7x61x1487 (COMPOSITE)
FERMAT'S = 19Ms
3x7x61x1487 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 22Ms
3x7x61x1487 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 3Ms
3x7x61x1487 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 4Ms
3x7x61x1487
POLLARD RHO = 198ms 38Ms
3x7x61x1487 (COMPOSITE)
P+1 = 3sec(s) 653ms 344Ms
3x7x61x1487 (COMPOSITE)
DIXON'S = 1sec(s) 5ms 490Ms
3x7x61x1487 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 635ms 855Ms
3x7x61x1487 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 602ms 356Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
```

```
COMPOSITE
MILLER RABIN VERDICT = 3Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 2Ms
#######################################

(50285201)
FACTORIZATION
-------------------------
17x431x6863 (COMPOSITE)
FERMAT'S = 61Ms
17x431x6863 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 91Ms
17x431x6863 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 9Ms
17x431x6863 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 6Ms
17x431x6863
POLLARD RHO = 531ms 248Ms
17x431x6863 (COMPOSITE)
P+1 = 2sec(s) 602ms 371Ms
17x431x6863 (COMPOSITE)
DIXON'S = 959ms 701Ms
17x431x6863 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 229ms 255Ms
17x431x6863 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 224ms 718Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 5Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(20958227)
FACTORIZATION
-------------------------
73x287099 (COMPOSITE)
```

```
FERMAT'S = 3ms 907Ms
73x287099 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 3ms 728Ms
73x287099 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 9Ms
73x287099 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 7Ms
73x287099
POLLARD RHO = 3sec(s) 998ms 730Ms
73x287099 (COMPOSITE)
P+1 = 3sec(s) 537ms 654Ms
73x287099 (COMPOSITE)
DIXON'S = 969ms 639Ms
73x287099 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 251ms 967Ms
73x287099 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 259ms 475Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 3Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(20968677)
FACTORIZATION
-------------------------
3x3x37x62969 (COMPOSITE)
FERMAT'S = 818Ms
3x3x37x62969 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 819Ms
3x3x37x62969 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 9Ms
3x3x37x62969 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 5Ms
3x3x37x62969
POLLARD RHO = 1sec(s) 684ms 469Ms
3x3x37x62969 (COMPOSITE)
```

```
P+1 = 4sec(s) 723ms 970Ms
3x3x37x62969 (COMPOSITE)
DIXON'S = 1sec(s) 704ms 781Ms
3x3x37x62969 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 765ms 879Ms
3x3x37x62969 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 738ms 599Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 4Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 4Ms
#######################################

(27620777)
FACTORIZATION
-------------------------
2029x13613 (COMPOSITE)
FERMAT'S = 153Ms
2029x13613 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 179Ms
2029x13613 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 30Ms
2029x13613 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 14Ms
2029x13613
POLLARD RHO = 877ms 625Ms
2029x13613 (COMPOSITE)
P+1 = 2sec(s) 557ms 195Ms
2029x13613 (COMPOSITE)
DIXON'S = 734ms 509Ms
2029x13613 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 32ms 734Ms
2029x13613 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 32ms 443Ms

PRIMALITY TEST
-------------------------
```

```
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 4Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(77777777)
FACTORIZATION
-------------------------
7x11x73x101x137 (COMPOSITE)
FERMAT'S = 15Ms
7x11x73x101x137 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 5Ms
7x11x73x101x137 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 5Ms
7x11x73x101x137 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 5Ms
7x11x73x101x137
POLLARD RHO = 95ms 527Ms
7x11x73x101x137 (COMPOSITE)
P+1 = 4sec(s) 326ms 590Ms
7x11x73x101x137 (COMPOSITE)
DIXON'S = 1sec(s) 4ms 131Ms
7x11x73x101x137 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 293ms 103Ms
7x11x73x101x137 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 269ms 361Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 5Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 2Ms
#######################################

(7777777)
FACTORIZATION
```

```
-------------------------
7x239x4649 (COMPOSITE)
FERMAT'S = 48Ms
7x239x4649 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 62Ms
7x239x4649 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 5Ms
7x239x4649 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 5Ms
7x239x4649
POLLARD RHO = 396ms 934Ms
7x239x4649 (COMPOSITE)
P+1 = 2sec(s) 618ms 695Ms
7x239x4649 (COMPOSITE)
DIXON'S = 820ms 100Ms
7x239x4649 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 931ms 380Ms
7x239x4649 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 943ms 97Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 4Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(208626111)
FACTORIZATION
-------------------------
3x3x3x3x179x14389 (COMPOSITE)
FERMAT'S = 112Ms
3x3x3x3x179x14389 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 190Ms
3x3x3x3x179x14389 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 5Ms
3x3x3x3x179x14389 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 5Ms
3x3x3x3x179x14389
```

```
POLLARD RHO = 744ms 856Ms
3x3x3x3x179x14389 (COMPOSITE)
P+1 = 5sec(s) 923ms 965Ms
3x3x3x3x179x14389 (COMPOSITE)
DIXON'S = 1sec(s) 623ms 502Ms
3x3x3x3x179x14389 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 981ms 426Ms
3x3x3x3x179x14389 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 986ms 297Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 3Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(290384205)
FACTORIZATION
-------------------------
3x5x139x139273 (COMPOSITE)
FERMAT'S = 1ms 711Ms
3x5x139x139273 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 1ms 811Ms
3x5x139x139273 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 8Ms
3x5x139x139273 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 6Ms
3x5x139x139273
POLLARD RHO = 2sec(s) 651ms 440Ms
3x5x139x139273 (COMPOSITE)
P+1 = 4sec(s) 4ms 883Ms
3x5x139x139273 (COMPOSITE)
DIXON'S = 2sec(s) 142ms 238Ms
3x5x139x139273 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 2sec(s) 517ms 497Ms
3x5x139x139273 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 2sec(s) 505ms 943Ms
```

```
PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 5Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 4Ms
#######################################

(190581011)
FACTORIZATION
-------------------------
29x6571759 (COMPOSITE)
FERMAT'S = 91ms 39Ms
29x6571759 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 85ms 553Ms
29x6571759 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 37Ms
29x6571759 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 15Ms
29x6571759
POLLARD RHO = 23sec(s) 752ms 680Ms
29x6571759 (COMPOSITE)
P+1 = 3sec(s) 651ms 589Ms
29x6571759 (COMPOSITE)
DIXON'S = 1sec(s) 317ms 64Ms
29x6571759 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 1sec(s) 413ms 485Ms
29x6571759 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 1sec(s) 408ms 379Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 4Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################
```

```
(2905801811)
FACTORIZATION
-------------------------
11x11x139x197x877 (COMPOSITE)
FERMAT'S = 200Ms
11x11x139x197x877 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 15Ms
11x11x139x197x877 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 5Ms
11x11x139x197x877 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 5Ms
11x11x139x197x877
POLLARD RHO = 206ms 413Ms
11x11x139x197x877 (COMPOSITE)
P+1 = 4sec(s) 375ms 989Ms
11x11x139x197x877 (COMPOSITE)
DIXON'S = 1sec(s) 985ms 883Ms
11x11x139x197x877 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 2sec(s) 183ms 146Ms
11x11x139x197x877 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 2sec(s) 157ms 252Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 5Ms
COMPOSITE
MILLER RABIN VERDICT = 4Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 2Ms
########################################

(29068206117)
FACTORIZATION
-------------------------
3x41x683x346013 (COMPOSITE)
FERMAT'S = 3ms 18Ms
3x41x683x346013 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 4ms 492Ms
3x41x683x346013 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 12Ms
3x41x683x346013 (COMPOSITE)
```

```
TRIAL DIVISION -- SIEVE = 7Ms
3x41x683x346013
POLLARD RHO = 4sec(s) 527ms 519Ms
3x41x683x346013 (COMPOSITE)
P+1 = 4sec(s) 643ms 211Ms
3x41x683x346013 (COMPOSITE)
DIXON'S = 2sec(s) 845ms 324Ms
3x41x683x346013 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 3sec(s) 118ms 220Ms
3x41x683x346013 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 3sec(s) 87ms 218Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 3Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(109687106781)
FACTORIZATION
-------------------------
3x3x7x83x2801x7489 (COMPOSITE)
FERMAT'S = 429Ms
3x3x7x83x2801x7489 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 100Ms
3x3x7x83x2801x7489 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 41Ms
3x3x7x83x2801x7489 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 17Ms
3x3x7x83x2801x7489
POLLARD RHO = 745ms 195Ms
3x3x7x83x2801x7489 (COMPOSITE)
P+1 = 5sec(s) 338ms 531Ms
3x3x7x83x20976689 (COMPOSITE)
DIXON'S = 4sec(s) 299ms 378Ms
3x3x7x83x2801x7489 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 4sec(s) 612ms 361Ms
3x3x7x83x2801x7489 (COMPOSITE)
```

```
QUADRATIC SIEVE -- OPTIMIZED = 4sec(s) 486ms 346Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 3Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################

(1908510861011)
FACTORIZATION
-------------------------
 (TLE)
FERMAT'S = 695ms 730Ms
53 (TLE)
TRIAL DIVISION -- NAIVE (LINEAR) = 1sec(s) 307ms 751Ms
53x36009638887 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 2ms 611Ms
53x36009638887 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 562Ms
53x36009638887
POLLARD RHO = 16ms 474Ms
53x36009638887 (COMPOSITE)
P+1 = 9sec(s) 438ms 508Ms
53x36009638887 (COMPOSITE)
DIXON'S = 2sec(s) 712ms 509Ms
53x36009638887 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 2sec(s) 849ms 327Ms
53x36009638887 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 2sec(s) 707ms 269Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 4Ms
COMPOSITE
MILLER RABIN VERDICT = 4Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
```

```
#######################################

(1111111111111111)
FACTORIZATION
-------------------------
11x17x73x101x137x5882353 (COMPOSITE)
FERMAT'S = 188ms 157Ms
11x17x73x101x137x5882353 (COMPOSITE)
TRIAL DIVISION -- NAIVE (LINEAR) = 78ms 831Ms
11x17x73x101x137x5882353 (COMPOSITE)
TRIAL DIVISION -- NAIVE (SQRT) = 36Ms
11x17x73x101x137x5882353 (COMPOSITE)
TRIAL DIVISION -- SIEVE = 17Ms
11x17x73x101x137x5882353
POLLARD RHO = 22sec(s) 558ms 973Ms
11x17x73x101x137x5882353 (COMPOSITE)
P+1 = 5sec(s) 943ms 654Ms
11x17x73x101x137x5882353 (COMPOSITE)
DIXON'S = 6sec(s) 540ms 816Ms
11x17x73x101x137x5882353 (COMPOSITE)
QUADRATIC SIEVE -- NAIVE = 9sec(s) 760ms 115Ms
11x17x73x101x137x5882353 (COMPOSITE)
QUADRATIC SIEVE -- OPTIMIZED = 6sec(s) 645ms 172Ms

PRIMALITY TEST
-------------------------
COMPOSITE
FERMAT VERDICT = 3Ms
COMPOSITE
MILLER RABIN VERDICT = 4Ms
COMPOSITE
SOLOVAY-STRASSEN VERDICT = 3Ms
#######################################
```

5 Conclusion

In this paper, number of different classical integer factorization algorithms are implemented, including qudaratic sieve. Together with number field sieve, the refined version of the quadratic sieve is among the popular picks in factorizing large numbers. Small numbers are usually treated by the optimized version of the trial division because the code is very simple to write − in the coding competition setting, the trial division with sieve of Eratosthenes is the most exclusive pick because the large input is in the order of 10^9, where the simple algorithm runs in breeze. There are algorithms that we have not treated in this paper: elliptic curve, continued fraction, general number field sieve, and Shor's; this is because the quadratic sieve is superior to all but possibly to the number field one when the input is extremely large − to see the effect of the number field sieve, however, the number should be typically in the order of 10^{200}. The interested reader should see [1].

Rather than providing only the theory behind the algorithm, the actual implementation of the algorithms were provided in the paper, to shed a light on how god the algorithms are in practice. The implementation is tested on some input data, whose result is given in the experiment section. We will complete by providing some auxiliary functions that we used in the implementation: checking the input, measuring time, and treating the infinite precision.

This project is by far one of the best journey I have yet taken to understand a single domain of knowledge in detail − here, the integer factorization techniques. Now I am confident about how what seemed to be 'black box' in a lecture (in a sense that I do not know its sensitivity/behavior in an input data) actually works.

6 Appendix

6.1 Checking Validity of the Input

The validity of the input can be checked by simply checking whether all the character is a digit. Then, we truncate the leading zeros before processing the input; note that we do not consider the negative numbers, which is in accordance with **Lemma 1**. The following is the implementation:

```
bool isValid(string & s) {
        int sz=s.size();
        for (int i=0; i<sz; i++) {
                char ch=s[i];
                if(ch<'0' || ch>'9') return false;
        }
        while(s[0]=='0') s=s.substr(1);
        if(s.size()==0) s='0';
        return true;
}
```

6.2 Time Related Helper Functions

There are two helper functions that I have written to (1) measure time and (2) format time to user-friendly format.

6.2.1 Measuring Time

The time can be measured using timeval struct provided in the standard `sys/time.h` library.

```
ll get_time(struct timeval & a, struct timeval &b) {
        ll ans;
        ans = ll(b.tv_sec-a.tv_sec) * 1000000LL + ll(b.tv_usec - a.tv_usec);
        return ans;
}
```

6.2.2 Formatting Time

The following code switches the time in micro-seconds to user-friendly format.

```
void format_time(string s, ll t) {
        int div[6]={1000, 1000, 60, 60, 24, 365};
        string suf[7]={"Ms", "ms", "sec(s)", "min(s)", "hr(s)", "day(s)", "year(s)"};

        vector<ll> v(7);
        for (int i=0; i<7; i++) { v[i]=0; ll mod=div[i], cur=t%mod; t/=mod; v[i]=cur; }

        cout << s;
        bool ok=false;
        for (int i=0; i<7; i++) {
                ll cur=v[6-i];
                if(cur) { ok=true; cout << cur << suf[6-i] << " "; }
        }

        if(!ok) cout << "0s";
        cout << endl;
}
```

6.3 Infinite Precision Techniques

The code provided here deals with numbers up to 10^{18} fast – the range of `long long`. To treat the numbers that require more precisions than the standard `long long` provided by C++, we need to use an infinite precision library. There is a version that I have written to treat this; it treats sum, difference, product, division by an integer, and exponentiation by integer. The following is the code:

```
bool cmp(string s, string t) {
        if(s==t) return 0;
        int n=s.size(), m=t.size();
        if(n<m) return -1;
        if(n>m) return 1;
        for (int i=0; i<n; i++) {
                char a=s[i], b=t[i];
                if(a>b) return 1;
                if(a<b) return -1;
        }
}

vector<int> tov(string s) {
        int sz=s.size();
        vector<int> ans(sz,0);
        for (int i=0; i<sz; i++)
                ans[i]=s[sz-1-i]-'0';
        return ans;
}

string tos(vector<int> & v) {
        int sz=v.size();
        string ans;
        for (int i=0; i<sz; i++)
                ans=(char)(v[i]+'0')+ans;
        while(ans[0]=='0') ans=ans.substr(1);
        if(!ans.size()) return "0";
        return ans;
}

string prod(string s, string t) {
        int n=s.size(), m=t.size(), l=n+m+1;
        vector<int> v=tov(s), w=tov(t), ans(l,0);
        for (int i=0; i<n; i++)
                for (int j=0; j<m; j++)
                        ans[i+j]+=v[i]*w[j];
        for (int i=0; i<l-1; i++) {
                ans[i+1]+=ans[i]/10;
                ans[i]%=10;
        }
        return tos(ans);
}

string sum(string s, string t) {
        int n=s.size(), m=t.size(), l=max(n,m)+1;
        vector<int> v=tov(s), w=tov(t), ans(l,0);
        for (int i=0; i<n; i++)
                ans[i]+=v[i];
        for (int i=0; i<m; i++)
```

```
                ans[i]+=w[i];
        for (int i=0; i<l-1; i++) {
                ans[i+1]+=ans[i]/10;
                ans[i]%=10;
        }
        return tos(ans);
}

string diff(string s, string t) {
        char sgn='+';
        if(cmp(s,t)<0) { sgn='-'; swap(s,t); }
        int n=s.size(), m=t.size(), l=max(n,m);
        vector<int> v=tov(s), w=tov(t), ans(l,0);
        for (int i=0; i<n; i++)
                ans[i]+=v[i];
        for (int i=0; i<m; i++)
                ans[i]-=w[i];
        for (int i=0; i<l-1; i++) {
                if(ans[i]<0) {
                        ans[i+1]--;
                        ans[i]+=10;
                }
        }
        string cur=tos(ans);
        if(sgn=='-') cur=sgn+cur;
        return cur;
}

string div(string s, int n) {
        int sz=s.size(), cur=0;
        string ans;
        for (int i=0; i<sz; i++) {
                cur=cur*10+(s[i]-'0');
                ans+=('0'+cur/n);
                cur%=n;
        }
        while(ans[0]=='0') ans=ans.substr(1);
        return ans;
}

string exp(string s, int n) {
        string ans="1";
        while(n--) ans=prod(ans,s);
        return ans;
}
```

The basic idea behind this is to represent the number in string format, so that an arbitary number of digits can be represented, only bounded by the memory limit; for instance, the numbers that have million digits can be dealt without a problem using string. Then, we perform corresponding arithmetic on each character of the string. This version, however, is a bit too naive and not practical enough for integer factorization, in terms of speed. The more practical source is either the *Number Theory Library* (NTL) written by Victor Shoup in C++ for treating computational number theory problem or the standard *GNU Multiple Precision Arithmetic Library* (GMP) in C for general audience.

I used *NTL* to extend the quadratic sieve to treat the infinite precision case because the library is just the perfect one for the integer factorization. After downloading the library, only very minor change was required to go from the finite version to the infinite version – the type of variable used had to be replaced from `ll` to `ZZ`. The code turns out impractical for all but quadratic sieve for very large input; so, the use of the infinite precision in my implementation was to mainly test the robustness of the quadratic sieve code for a larger input.

6.4 Tables

I provide here the tables that are explicitly computed from the sieve of Eratosthenes code, as a sanity check for the code:

6.4.1 Primes up to 1000

Here is a list of primes up to 1000, from the sieve of Eratosthenes code: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997

6.4.2 $\pi(n)$ for $n = 10^k$ where $k = 1, \cdots, 10$

Here is the list of $\pi(n)$ for some small powers of 10, computed from the sieve of Eratosthenes code: 4 25 168 1229 9592 78498 664579 5716455 50847534 455052511

6.5 Starting few lines...

Just for completeness, here is first few lines of the code, that includes and defines appropriate variables:

```
#include<map>
#include<queue>
#include<algorithm>
#include<math.h>
#include<sstream>
#include<iostream>
#include<stdio.h>
#include<set>
#include<string>
#include<vector>
#include<stdlib.h>
#include<time.h>
#include<sys/time.h>
using namespace std;

#define EPS 10e-6
#define EPS_MAT 100
#define MAX_LL 10000000000000000LL
#define MAX_INT 100000001
#define MAX_FERMAT 50000000LL
#define MAX_ITER 5000
#define MAX_ITER_SMALL 100
#define MAX_ITER_QS 20000LL
#define MAX_B 1000000000LL
#define MIN_B 1000LL
#define MIN_B_QS 20000LL
#define SIZE_FACTOR_BASE 5000
#define MUL_FACTOR 1

typedef long long ll;
typedef unsigned long long ull;

vector<ll> p;
bool is_prime[MAX_INT];
int npfact, nprime;
int grid[SIZE_FACTOR_BASE][SIZE_FACTOR_BASE];
int grid_mod2[SIZE_FACTOR_BASE][SIZE_FACTOR_BASE];
int id_mod2[SIZE_FACTOR_BASE][SIZE_FACTOR_BASE];
int pexp[SIZE_FACTOR_BASE];
```

6.6 Using the code...

The code is written in a single file named `factor.cpp`. To run it in Linux, we do:

```
g++ factor.cpp && ./a.out
```

We can type in whatever input we desire in the command line. Alternatively, we can feed in some text document, named, for example, `in.txt`, where each line is a single string of digits. To run this, we can do:

```
g++ factor.cpp && ./a.out < in.txt
```

Finally, to save the output in a file, named, for example, `out.txt`, we can do:

```
g++ factor.cpp && ./a.out < in.txt > out.txt
```

The result in Section 4 is an outcome of feeding in an input file consisting ≈ 20 lines, which are randomly chosen by hand, before executing the code; a care is taken to get sufficiently random numbers – uniformly generating random numbers were also tried, but it was strictly worse than this choice of inputs.

7 Acknowledgement

This paper is prepared to fulfill the final project for **MATH 249B** – An Introduction to Computational Number Theory (WINTER 2011-2012). I thank Professor Akshay Venkatesh for introducing me to this wonderful subject and guiding me to finish this write-up.

8 Reference

1. Crandall, Richard; Pomerance, Carl (2005), Prime Numbers: A Computational Perspective (2nd ed.), Berlin, New York: Springer-Verlag, ISBN 978-0-387-25282-7

2. P. L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In Proc. Eurocrypt 1995, volume 921 of Lect. Notes Comput. Sci., pages 106120, Heidelberg, Germany, 1995. Springer Verlag.