



NANJING UNIVERSITY

ACM-ICPC Codebook 2

Number Theory
Linear Algebra
Combinatorics

January 27, 2018

Contents

1	Number Theory	4
1.1	Modulo operations	4
1.1.1	Modular exponentiation (fast power-mod)	4
1.1.2	Mathematical modulo operation	4
1.1.3	Modular multiplication on long long	4
1.2	Extended Euclidian algorithm	5
1.2.1	Modular multiplicative inverse	5
1.3	Primality test (Miller-Rabin)	6
1.4	Sieve	7
1.4.1	of Eratosthenes	7
1.4.2	of Euler	7
1.5	Integer factorization (Pollard's rho algorithm)	8
1.6	Number theoretic transform	9
1.7	Fast Walsh-Hadamard transform	10
1.8	Pell's equation	11
2	Linear Algebra	12
2.1	Modular exponentiation of matrices	12
3	Combinatorics	13
3.1	Möbius inversion	13
3.2	Permutations	13
3.3	Pólya enumeration theorem	14

1 Number Theory

1.1 Modulo operations

1.1.1 Modular exponentiation (fast power-mod)

Calculate $b^e \bmod m$.

Time complexity: $O(\log e)$

```

1 LL powmod(LL b, unsigned long long e, LL m){
2     LL r = 1;
3     while (e){
4         if (e & 1) r = r * b % m;
5         b = b * b % m;
6         e >>= 1;
7     }
8     return r;
9 }
```

1.1.2 Mathematical modulo operation

The result has the same sign as divisor.

```

1 inline LL mathmod(LL a, LL b){
2     return (a % b + b) % b;
3 }
```

1.1.3 Modular multiplication on long long

Calculate $ab \bmod m$, where a, b, m are long long integers.

\triangle a, b, m must be non-negative.

Time complexity: $O(\log b)$

```

1 LL mulmod(LL a, LL b, LL m){
2     LL r = 0;
3     a %= m; b %= m;
4     while(b) {
5         if(b & 1) r += a, r %= m;
6         b >>= 1;
7         if(a < m - a)
```

```

8         a <= 1;
9     else
10         a -= (m - a);
11     }
12     return r;
13 }

```

1.2 Extended Euclidian algorithm

Solve $ax + by = g = \gcd(a, b)$ w.r.t. x, y .

If (x_0, y_0) is an integer solution of $ax + by = g = \gcd(x, y)$, then every integer solution of it can be written as $(x_0 + kb', y_0 - ka')$, where $a' = a/g$, $b' = b/g$, and k is arbitrary integer.

\triangle x and y must be positive.

Usage:

`exgcd(a, b, g, x, y)` Find a special solution to $ax + by = g = \gcd(a, b)$.

Time complexity: $O(\log \min\{a, b\})$

```

1 void exgcd(int a, int b, int &g, int &x, int &y){
2     if (!b) g = a, x = 1, y = 0;
3     else {
4         exgcd(b, a % b, g, y, x),
5         y -= x * (a / b);
6     }
7 }

```

1.2.1 Modular multiplicative inverse

An integer a has modular multiplicative inverse w.r.t. the modulus m , iff $\gcd(a, m) = 1$. Assume the inverse is x , then

$$ax \equiv 1 \pmod{m}.$$

Call `exgcd(a, m, g, x, y)`, if $g = 1$, $x + km$ is the modular multiplicative inverse of a w.r.t. the modulus m .

```

1 inline LL minv(LL a, LL m){
2     LL g, x, y;
3     exgcd(a, m, g, x, y);
4     return (x % m + m) % m;
5 }

```

Or, by Fermat's little theorem ($a^{p-1} \equiv 1 \pmod{p}$), when $m = p$ is a prime, the multiplicative inverse can also be written as $a^{-1} = (a^{p-2} \pmod{p})$.

Also, the inverses of first n numbers can be precalculated in $O(n)$ time.

```

1 LL inv[100005];
2 LL mod;
3
4 void init(){
5     inv[1] = 1;
6     for (int i = 2; i < n; i++)
7         inv[i] = (mod - mod / i) * inv[mod % i] % mod;
8 }

```

1.3 Primality test (Miller-Rabin)

Test whether n is a prime.

The array $a[]$ (excluding sentinel, e.g. `LLONG_MAX`) should be

{2}	when $n < 2,047$.
{2, 7, 61}	when $n < 4,759,123,141$ (2^{32}).
{2, 3, 5, 7, 11}	when $n < 2.1 \times 10^{12}$.
{2, 325, 9375, 28178, 450775, 9780504, 1795265022}	when $n < 2^{64}$.

△ When n exceeds the range of `int`, the mul-mod and pow-mod operations should be rewritten.

Requirement:

1.1.1 Modular exponentiation (fast power-mod)

Time complexity: $O(\log n)$

```

1 bool test(LL n){
2     if (n < 3) return n==2;
3     // ! The array a[] should be modified if the range of x changes.
4     const LL a[] = {2LL, 7LL, 61LL, LLONG_MAX};
5     LL r = 0, d = n-1, x;
6     while (~d & 1) d >>= 1, r++;
7     for (int i=0; a[i] < n; i++){
8         x = powmod(a[i], d, n);
9         if (x == 1 || x == n-1) goto next;
10        rep (i, r) {
11            x = (x * x) % n;
12            if (x == n-1) goto next;
13        }
14        return false;

```

```

15 next;;
16     }
17     return true;
18 }

```

1.4 Sieve

1.4.1 of Eratosthenes

Usage:

sieve() Generate the table.
 p[i] True if i is **not** a prime; otherwise false.

Time complexity: Approximately linear.

```

1 const int MAXX = 1e7+5;
2 bool p[MAXX];
3
4 void sieve(){
5     p[0] = p[1] = 1;
6     for (int i = 2; i*i < MAXX; i++) if (!p[i])
7         for (int j = i*i; j < MAXX; j+=i) p[j] = true;
8 }

```

1.4.2 of Euler

Usage:

sieve() Generate the table.
 p[i] True if i is **not** a prime; otherwise false.
 prime[i] The i th prime number.

Time complexity: Linear.

```

1 const int MAXX = 1e7+5;
2 bool p[MAXX];
3 int prime[MAXX], sz;
4
5 void sieve(){
6     p[0] = p[1] = 1;
7     for (int i = 2; i < MAXX; i++){
8         if (!p[i]) prime[sz++] = i;
9         for (int j = 0; j < sz && i*prime[j] < MAXX; j++){
10             p[i*prime[j]] = 1;

```

```

11         if (i % prime[j] == 0) break;
12     }
13 }
14 }

```

This technique can also be used to compute multiplicative functions. For example, the following program computes the Euler's totient function.

```

1  const int MAXX = 1e7+5;
2  int phi[MAXX];
3  int prime[MAXX], sz;
4
5  void sieve(){
6      phi[1] = 1;
7      for (int i = 2; i < MAXX; i++){
8          if (!phi[i]) phi[i] = i-1, prime[sz++] = i;
9          for (int j = 0; j < sz && i*prime[j] < MAXX; j++){
10             if (i % prime[j] == 0) {
11                 phi[i*prime[j]] = phi[i]*prime[j];
12                 break;
13             }
14             phi[i*prime[j]] = phi[i]*(prime[j] - 1);
15         }
16     }
17 }

```

1.5 Integer factorization (Pollard's rho algorithm)

Find a nontrivial factor of a composite integer. One can recursively call this procedure to complete the factorization, by divide and conquer.

Time complexity: Believed to be $O(n^{1/4})$ in expectation.

```

1  ULL gcd(ULL a, ULL b) {return b ? gcd(b, a % b) : a;}
2
3  ULL PollardRho(ULL n){
4      ULL c, x, y, d = n;
5      if (~n&1) return 2;
6      while (d == n){
7          x = y = 2;
8          d = 1;
9          c = rand() % (n - 1) + 1;
10         while (d == 1){
11             x = (mulmod(x, x, n) + c) % n;
12             y = (mulmod(y, y, n) + c) % n;
13             y = (mulmod(y, y, n) + c) % n;

```



```

14         d = gcd(x-y>0 ? x-y : y-x, n);
15     }
16 }
17 return d;
18 }

```

1.6 Number theoretic transform

△ The size of the sequence must be some power of 2.

△ When performing convolution, the size of the sequence should be doubled. To compute k , one may call `32-__builtin_clz(a+b-1)`, where a and b are the lengths of two sequences.

Usage:

<code>NTT(k)</code>	Initialize the structure with maximum sequence length 2^k .
<code>ntt(a)</code>	Perform number theoretic transform on sequence a .
<code>intt(a)</code>	Perform inverse number theoretic transform on sequence a .
<code>conv(a, b)</code>	Convolve sequence a with b .

Time complexity: $O(n \log n)$.

```

1  const int NMAX = 1<<21;
2  // 998244353 = 7*17*2^23+1, G = 3
3  const int P = 1004535809, G = 3; // = 479*2^21+1
4
5  struct NTT{
6      int rev[NMAX];
7      LL omega[NMAX], oinv[NMAX];
8      int g, g_inv; // g: g_n = G^((P-1)/n)
9      int K, N;
10
11     LL powmod(LL b, LL e){
12         LL r = 1;
13         while (e){
14             if (e&1) r = r * b % P;
15             b = b * b % P;
16             e >>= 1;
17         }
18         return r;
19     }
20
21     NTT(int k){
22         K = k; N = 1 << k;
23         g = powmod(G, (P-1)/N);
24         g_inv = powmod(g, N-1);
25         omega[0] = oinv[0] = 1;

```

```

26     rep (i, N){
27         rev[i] = (rev[i>>1]>>1) | ((i&1)<<(K-1));
28         if (i){
29             omega[i] = omega[i-1] * g % P;
30             oinv[i] = oinv[i-1] * g_inv % P;
31         }
32     }
33 }
34
35 void _ntt(LL* a, LL* w){
36     rep (i, N) if (i < rev[i]) swap(a[i], a[rev[i]]);
37     for (int l = 2; l <= N; l *= 2){
38         int m = l/2;
39         for (LL* p = a; p != a + N; p += l)
40             rep (k, m){
41                 LL t = w[N/l*k] * p[k+m] % P;
42                 p[k+m] = (p[k] - t + P) % P;
43                 p[k] = (p[k] + t) % P;
44             }
45     }
46 }
47
48 void ntt(LL* a){_ntt(a, omega);}
49 void intt(LL* a){
50     LL inv = powmod(N, P-2);
51     _ntt(a, oinv);
52     rep (i, N) a[i] = a[i] * inv % P;
53 }
54
55 void conv(LL* a, LL* b){
56     ntt(a); ntt(b);
57     rep (i, N) a[i] = a[i] * b[i] % P;
58     intt(a);
59 }
60 };

```

1.7 Fast Walsh-Hadamard transform

This is to compute

$$C[i] = \sum_{j \oplus k = i} A[j] \cdot B[k],$$

where \oplus is a binary bitwise operation.

Time complexity: $O(n \log n)$.

```

1 void fwt(int* a, int n){
2     for (int d = 1; d < n; d <= 1)
3         for (int i = 0; i < n; i += d << 1)
4             rep (j, d){
5                 int x = a[i+j], y = a[i+j+d];
6                 // a[i+j] = x+y, a[i+j+d] = x-y;    // xor
7                 // a[i+j] = x+y;                    // and
8                 // a[i+j+d] = x+y;                    // or
9             }
10 }
11
12 void ifwt(int* a, int n){
13     for (int d = 1; d < n; d <= 1)
14         for (int i = 0; i < n; i += d << 1)
15             rep (j, d){
16                 int x = a[i+j], y = a[i+j+d];
17                 // a[i+j] = (x+y)/2, a[i+j+d] = (x-y)/2;    // xor
18                 // a[i+j] = x-y;                               // and
19                 // a[i+j+d] = y-x;                               // or
20             }
21 }
22
23 void conv(int* a, int* b, int n){
24     fwt(a, n);
25     fwt(b, n);
26     rep(i, n) a[i] *= b[i];
27     ifwt(a, n);
28 }

```

1.8 Pell's equation

$x^2 - ny^2 = 1$, where n is a positive nonsquare integer.

Let (x_0, y_0) be the smallest positive solution of the equation, then the k -th solution is:

$$\begin{pmatrix} x_k \\ y_k \end{pmatrix} = \begin{pmatrix} x_1 & ny_1 \\ y_1 & x_1 \end{pmatrix}^k \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

2 Linear Algebra

2.1 Modular exponentiation of matrices

Calculate $b^e \bmod \text{modular}$, where b is a matrix. The modulus is element-wise.

Usage:

<code>n</code>	Order of matrices.
<code>modular</code>	The divisor in modulo operations.
<code>m_powmod(b, e)</code>	Calculate $b^e \bmod \text{modular}$. The result is stored in <code>r</code> .

Time complexity: $O(n^3 \log e)$

```

1 const int MAXN = 105;
2 const LL modular = 1000000007;
3 int n; // order of matrices
4
5 struct matrix{
6     LL m[MAXN][MAXN];
7
8     void operator *=(matrix& a){
9         static LL t[MAXN][MAXN];
10        Rep (i, n){
11            Rep (j, n){
12                t[i][j] = 0;
13                Rep (k, n){
14                    t[i][j] += (m[i][k] * a.m[k][j]) % modular;
15                    t[i][j] %= modular;
16                }
17            }
18        }
19        memcpy(m, t, sizeof(t));
20    }
21 };
22
23 matrix r;
24 void m_powmod(matrix& b, LL e){
25     memset(r.m, sizeof(r.m), 0);
26     Rep(i, n)
27         r.m[i][i] = 1;
28     while (e){
29         if (e & 1) r *= b;
30         b *= b;
31         e >>= 1;
32     }
33 }
```

3 Combinatorics

3.1 Möbius inversion

Möbius function:

$$\mu(n) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{if } p_i^{a_i} \mid n \text{ where } a_i > 0 \\ (-1)^r & \text{if } n \text{ is the product of } r \text{ distinct primes} \end{cases}$$

If $S_f(n) = \sum_{d|n} f(d)$, then $f(n) = \sum_{d|n} \mu(d) S_f(n/d)$.

3.2 Permutations

This provides operations of permutations of 0 to $n - 1$.

Usage:

<code>a*b</code>	Compute the composition of permutations a and b .
<code>~a</code>	Compute the inverse permutation of a .
<code>permutation(a)</code>	Factorize the permutation to disjoint cycles.

Time complexity: $O(n)$

```

1 typedef vector<int> perm;
2
3 perm operator * (const perm lhs, const perm rhs){
4     int sz;
5     assert((sz = lhs.size()) == rhs.size());
6     perm res; res.resize(sz);
7     for (int i=0; i<sz; i++){
8         res[i] = rhs[lhs[i]];
9     }
10    return res;
11 }
12
13 perm operator ~ (const perm lhs){
14     int sz = lhs.size();
15     perm res; res.resize(sz);
16     for (int i=0; i<sz; i++){
17         res[lhs[i]] = i;
18     }
19     return res;
20 }
21
22 struct permutation{

```

```

23  int size;
24  vector<vector<int>> orbits;
25
26  permutation(permutation p){
27      size = p.size();
28      vector<bool> visited(size);
29      for (int i=0; i<size; i++){
30          if (visited[i]) continue;
31          int cur = i;
32          vector<int> orbit;
33          while (!visited[cur]){
34              visited[cur] = true;
35              orbit.push_back(cur);
36              cur = p[cur];
37          }
38          orbits.push_back(move(orbit));
39      }
40  }
41 };

```

3.3 Pólya enumeration theorem

The Burnside's lemma says that

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

where G is a group acting on X , X^g is the set of elements in X that are fixed by g , i.e. $X^g = \{x \in X : gx = x\}$.

The unweighted version of Pólya enumeration theorem says that

$$|Y^X/G| = \frac{1}{|G|} \sum_{g \in G} m^{c_g}$$

where $m = |X|$ is the number of colors, c_g is the number of the cycles of permutation g .