Nanjing University

ACM-ICPC Codebook 2

Number Theory
Linear Algebra
Combinatorics

September 12, 2018

Contents

1 Number Theory

1.1 Modulo operations

1.1.1 Modular exponentiation (fast power-mod)

Calculate $b^e \bmod m$.

Time complexity: $O(\log e)$

```
LL powmod(LL b, unsigned long long e, LL m){
    LL r = 1;
    while (e){
        if (e & 1) r = r * b % m;
        b = b * b % m;
        e >>= 1;
    }
    return r;
}
```

1.1.2 Mathematical modulo operation

The result has the same sign as divisor.

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

1.1.3 Modular multiplication on long long

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

⚠ a, b, m must be non-negative.

Time complexity: $O(\log b)$

```
LL mulmod(LL a, LL b, LL m){
    LL r = 0;
    a %= m; b %= m;
    while(b) {
        if(b & 1) r += a, r %= m;
        b >>= 1;
        if(a < m - a)
```

```
            a <<= 1;
        else
            a -= (m - a);
    }
    return r;
}

LL mulmod(LL a, LL b) {
  LL tmp = (a * b - (LL)((long double)a/p*b + 1e-8)*p);
  return tmp < 0 ? tmp + p : tmp;
}
```

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.

⚠ 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: $O(\log \min\{a, b\})$

```
void exgcd(int a, int b, int &g, int &x, int &y){
    if (!b) g = a, x = 1, y = 0;
    else {
        exgcd(b, a % b, g, y, x),
        y -= x * (a / b);
    }
}
```

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

$$ax \equiv 1 \mod m.$$

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.

```cpp
inline LL minv(LL a, LL m){
    LL g, x, y;
    exgcd(a, m, g, x, y);
    return (x % m + m) % m;
}
```

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

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

```cpp
LL inv[100005];
LL mod;

void init(){
    inv[1] = 1;
    for (int i = 2; i < n; i++)
        inv[i] = (mod - mod / i) * inv[mod % i] % mod;
}
```

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: $O(\log n)$

```cpp
bool test(LL n){
    if (n < 3) return n==2;
    // ! The array a[] should be modified if the range of x changes.
    const LL a[] = {2LL, 7LL, 61LL, LLONG_MAX};
    LL r = 0, d = n-1, x;
    while (~d & 1) d >>= 1, r++;
    for (int i=0; a[i] < n; i++){
        x = powmod(a[i], d, n);
        if (x == 1 || x == n-1) goto next;
        rep (i, r) {
            x = mulmod(x, x, n);
            if (x == n-1) goto next;
        }
        return false;
next:;
    }
    return true;
}
```

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.

```
const int MAXX = 1e7+5;
bool p[MAXX];

void sieve(){
    p[0] = p[1] = 1;
    for (int i = 2; i*i < MAXX; i++) if (!p[i])
        for (int j = i*i; j < MAXX; j+=i) p[j] = true;
}
```

1.4.2 of Euler

Usage:

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

Time complexity: Linear.

```
const int MAXX = 1e7+5;
bool p[MAXX];
```

```cpp
int prime[MAXX], sz;

void sieve(){
    p[0] = p[1] = 1;
    for (int i = 2; i < MAXX; i++){
        if (!p[i]) prime[sz++] = i;
        for (int j = 0; j < sz && i*prime[j] < MAXX; j++){
            p[i*prime[j]] = 1;
            if (i % prime[j] == 0) break;
        }
    }
}
```

This technique can also be used to compute multiplicative functions.

```cpp
namespace sieve {
  constexpr int MAXN = 10000007;
  bool p[MAXN];
  int prime[MAXN], sz;
  int pval[MAXN], pcnt[MAXN];
  int f[MAXN];

  void exec(int N = MAXN) {
    p[0] = p[1] = 1;

    pval[1] = 1;
    pcnt[1] = 0;
    f[1] = 1;

    for (int i = 2; i < N; i++) {
      if (!p[i]) {
        prime[sz++] = i;
        for (LL j = i; j < N; j *= i) {
          int b = j / i;
          pval[j] = i * pval[b];
          pcnt[j] = pcnt[b] + 1;
          f[j] = ______; // f[j] = f(i^pcnt[j])
        }
      }
      for (int j = 0; i * prime[j] < N; j++) {
        int x = i * prime[j]; p[x] = 1;
        if (i % prime[j] == 0) {
          pval[x] = pval[i] * prime[j];
          pcnt[x] = pcnt[i] + 1;
        } else {
          pval[x] = prime[j];
          pcnt[x] = 1;
        }
        if (x != pval[x]) {
          f[x] = f[x / pval[x]] * f[pval[x]]
        }
        if (i % prime[j] == 0) break;
      }
    }
  }
}
```

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.

⚠ Please use Miller-Rabin to test primality of the input; for prime input, the algorithm may trap into infinite loop.

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

```
ULL gcd(ULL a, ULL b) {return b ? gcd(b, a % b) : a;}

ULL PollardRho(ULL n){
    ULL c, x, y, d = n;
    if (~n&1) return 2;
    while (d == n){
        x = y = 2;
        d = 1;
        c = rand() % (n - 1) + 1;
        while (d == 1){
            x = (mulmod(x, x, n) + c) % n;
            y = (mulmod(y, y, n) + c) % n;
            y = (mulmod(y, y, n) + c) % n;
            d = gcd(x>y ? x-y : y-x, n);
        }
    }
    return d;
}
```

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:

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

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

```
const int NMAX = 1<<21;

// 998244353 = 7*17*2^23+1, G = 3
const int P = 1004535809, G = 3; // = 479*2^21+1

struct NTT{
    int rev[NMAX];
    LL omega[NMAX], oinv[NMAX];
    int g, g_inv; // g: g_n = G^((P-1)/n)
    int K, N;

    LL powmod(LL b, LL e){
        LL r = 1;
        while (e){
            if (e&1) r = r * b % P;
            b = b * b % P;
            e >>= 1;
        }
        return r;
    }

    NTT(int k){
        K = k; N = 1 << k;
        g = powmod(G, (P-1)/N);
        g_inv = powmod(g, N-1);
        omega[0] = oinv[0] = 1;
        rep (i, N){
            rev[i] = (rev[i>>1]>>1) | ((i&1)<<(K-1));
            if (i){
                omega[i] = omega[i-1] * g % P;
                oinv[i] = oinv[i-1] * g_inv % P;
            }
        }
    }

    void _ntt(LL* a, LL* w){
        rep (i, N) if (i < rev[i]) swap(a[i], a[rev[i]]);
        for (int l = 2; l <= N; l *= 2){
            int m = l/2;
            for (LL* p = a; p != a + N; p += l)
```

```
                rep (k, m){
                    LL t = w[N/l*k] * p[k+m] % P;
                    p[k+m] = (p[k] - t + P) % P;
                    p[k] = (p[k] + t) % P;
                }
        }
    }

    void ntt(LL* a){_ntt(a, omega);}
    void intt(LL* a){
        LL inv = powmod(N, P-2);
        _ntt(a, oinv);
        rep (i, N) a[i] = a[i] * inv % P;
    }

    void conv(LL* a, LL* b){
        ntt(a); ntt(b);
        rep (i, N) a[i] = a[i] * b[i] % P;
        intt(a);
    }
};
```

1.7 Fast Walsh-Hadamard transform

This is to compute

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

where $\oplus$ is a binary bitwise operation.

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

```
void fwt(int* a, int n){
    for (int d = 1; d < n; d <<= 1)
        for (int i = 0; i < n; i += d << 1)
            rep (j, d){
                int x = a[i+j], y = a[i+j+d];
                // a[i+j] = x+y, a[i+j+d] = x-y;    // xor
                // a[i+j] = x+y;                    // and
                // a[i+j+d] = x+y;                  // or
            }
}

void ifwt(int* a, int n){
    for (int d = 1; d < n; d <<= 1)
        for (int i = 0; i < n; i += d << 1)
```

```
            rep (j, d){
                int x = a[i+j], y = a[i+j+d];
                // a[i+j] = (x+y)/2, a[i+j+d] = (x-y)/2;    // xor
                // a[i+j] = x-y;                            // and
                // a[i+j+d] = y-x;                          // or
            }
}

void conv(int* a, int* b, int n){
    fwt(a, n);
    fwt(b, n);
    rep(i, n) a[i] *= b[i];
    ifwt(a, n);
}
```

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:

$$\begin{pmatrix} x_k \\ y_k \end{pmatrix} = \begin{pmatrix} x_0 & ny_0 \\ y_0 & x_0 \end{pmatrix}^k \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

Some smallest solutions to Pell's equation:

n	2	3	5	6	7	8	10	11	12	13	14	15	17	18	19	20
x	3	2	9	5	8	3	19	10	7	649	15	4	33	17	170	9
y	2	1	4	2	3	1	6	3	2	180	4	1	8	4	39	2

2 Linear Algebra

2.1 Modular exponentiation of matrices

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

Usage:

`n`	Order of matrices.
`modular`	The divisor in modulo operations.
`m_powmod(b, e)`	Calculate $b^e \bmod modular$. The result is stored in `r`.

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

```cpp
const int MAXN = 105;
const LL modular = 1000000007;
int n; // order of matrices

struct matrix{
    LL m[MAXN][MAXN];

    void operator *=(matrix& a){
        static LL t[MAXN][MAXN];
        Rep (i, n){
            Rep (j, n){
                t[i][j] = 0;
                Rep (k, n){
                    t[i][j] += (m[i][k] * a.m[k][j]) % modular;
                    t[i][j] %= modular;
                }
            }
        }
        memcpy(m, t, sizeof(t));
    }
};

matrix r;
void m_powmod(matrix& b, LL e){
    memset(r.m, 0, sizeof(r.m));
    Rep(i, n)
        r.m[i][i] = 1;
    while (e){
        if (e & 1) r *= b;
        b *= b;
        e >>= 1;
    }
}
```

2.2 Linear basis

Compute the basis over $\mathbb{F}_2$ field.

Usage:

`insert(v)`	Insert the vector. Return whether the vector is independent of the existing vectors.

Time complexity: $O(d)$ per operation.

```cpp
const int MAXD = 30;
```

```
struct linearbasis {
    ULL b[MAXD] = {};

    bool insert(ll v) {
        for (int j = MAXD - 1; j >= 0; j--) {
            if (!(v & (1ll << j))) continue;
            if (b[j]) v ^= b[j]
            else {
                for (int k = 0; k < j; k++)
                    if (v & (1ll << k)) v ^= b[k];
                for (int k = j + 1; k < MAXD; k++)
                    if (b[k] & (1ll << j)) b[k] ^= v;
                b[j] = v;
                return true;
            }
        }
        return false;
    }
};
```

2.3 Berlekamp-Massey algorithm

Compute the minimal polynomial of a linearly recurrent sequence over some finite field $\mathbb{F}_p$.

Usage:

`solve(v)`	Compute the minimum polynomial.

Time complexity: $O(n^2)$.

```
const LL MOD = 1000000007;

LL inverse(LL b) {
  LL e = MOD - 2, r = 1;
  while (e) {
    if (e & 1) r = r * b % MOD;
    b = b * b % MOD;
    e >>= 1;
  }
  return r;
}

struct Poly {
  vector<int> a;

  Poly() { a.clear(); }

  Poly(vector<int> &a) : a(a) {}

  int length() const { return a.size(); }

  Poly move(int d) {
    vector<int> na(d, 0);
    na.insert(na.end(), a.begin(), a.end());
    return Poly(na);
  }

  int calc(vector<int> &d, int pos) {
    int ret = 0;
    for (int i = 0; i < (int)a.size(); ++i) {
      if ((ret += (long long)d[pos - i] * a[i] % MOD) >= MOD) {
        ret -= MOD;
      }
    }
    return ret;
  }

  Poly operator - (const Poly &b) {
    vector<int> na(max(this->length(), b.length()));
    for (int i = 0; i < (int)na.size(); ++i) {
      int aa = i < this->length() ? this->a[i] : 0,
          bb = i < b.length() ? b.a[i] : 0;
      na[i] = (aa + MOD - bb) % MOD;
    }
    return Poly(na);
  }
};

Poly operator * (const int &c, const Poly &p) {
  vector<int> na(p.length());
  for (int i = 0; i < (int)na.size(); ++i) {
    na[i] = (long long)c * p.a[i] % MOD;
  }
  return na;
}

vector<int> solve(vector<int> a) {
  int n = a.size();
  Poly s, b;
  s.a.push_back(1), b.a.push_back(1);
  for (int i = 1, j = 0, ld = a[0]; i < n; ++i) {
    int d = s.calc(a, i);
    if (d) {
      if ((s.length() - 1) * 2 <= i) {
        Poly ob = b;
        b = s;
        s = s - (long long)d * inverse(ld) % MOD * ob.move(i - j);
        j = i;
        ld = d;
      } else {
        s = s - (long long)d * inverse(ld) % MOD * b.move(i - j);
      }
    }
  }
  // Caution: s.a might be shorter than expected
  return s.a;
}
```

3 Combinatorics

3.1 Twelvefold Way

$A(n)$	$B(m)$	f	number of f
dist.	dist.	-	m^n
dist.	dist.	inj.	$m^{\underline{n}}$
dist.	dist.	surj.	$m!S(n,m)$
dist.	id.	-	$\sum_{i=1}^{m} S(n,i)$
dist.	id.	inj.	$[n \le m]$
dist.	id.	surj.	$S(n,m)$
id.	dist.	-	$\binom{n+m-1}{n}$
id.	dist.	inj.	$\binom{m}{n}$
id.	dist.	surj.	$\binom{n-1}{m-1}$
id.	id.	-	$\sum_{i=1}^{m} p_i(n)$
id.	id.	inj.	$[n \le m]$
id.	id.	surj.	$p_m(n)$

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

3.3 Permutations

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

Usage:

`a*b`	Compute the composition of permutations a and b.
`~a`	Compute the inverse permutation of a.
`permutation(a)`	Factorize the permutation to disjoint cycles.

Time complexity: $O(n)$

```
typedef vector<int> perm;

```

```
perm operator * (const perm lhs, const perm rhs){
    int sz;
    assert((sz = lhs.size()) == rhs.size());
    perm res(sz);
    rep (i, sz) res[i] = rhs[lhs[i]];
    return res;
}

perm operator ~ (const perm lhs){
    int sz = lhs.size();
    perm res(sz);
    rep (i, sz) res[lhs[i]] = i;
    return res;
}

struct permutation{
    int size;
    vector<vector<int>> orbits;

    permutation(perm p){
        size = p.size();
        vector<bool> visited(size);
        rep (i, size) {
            if (visited[i]) continue;
            int cur = i;
            vector<int> orbit;
            while (!visited[cur]){
                visited[cur] = true;
                orbit.push_back(cur);
                cur = p[cur];
            }
            orbits.push_back(move(orbit));
        }
    }
};
```

3.4 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.

4 Appendix

4.1 Prime table

4.1.1 First primes

p	$g(p)$	p	$g(p)$	p	$g(p)$	p	$g(p)$	p	$g(p)$
2	1	3	2	5	2	7	3	11	2
13	2	17	3	19	2	23	5	29	2
31	3	37	2	41	6	43	3	47	5
53	2	59	2	61	2	67	2	71	7
73	5	79	3	83	2	89	3	97	5
101	2	103	5	107	2	109	6	113	3
127	3	131	2	137	3	139	2	149	2
151	6	157	5	163	2	167	5	173	2
179	2	181	2	191	19	193	5	197	2
199	3	211	2	223	3	227	2	229	6

4.1.2 Arbitrary length primes

$\lg p$	p	$g(p)$	p	$g(p)$
3	967	5	1031	14
4	9859	2	10273	10
5	96331	10	102931	3
6	958543	6	1031137	5
7	9594539	2	10169651	2
8	96243449	3	103211039	7
9	980483981	2	1042484357	2
10	9858935453	2	10261276009	7
11	95748666809	3	101759940101	2
12	950781833849	3	1012797784423	5
13	9739822952371	7	10037217092377	7
14	96181051140397	5	104974966380359	11
15	981030138360889	13	1029038416465403	2
16	9655206098080843	3	10116299875820773	2
17	97687777921994419	3	101506415998163437	2

4.1.3 $\sim 1 \times 10^9$

p	$g(p)$	p	$g(p)$	p	$g(p)$
954854573	3	967607731	2	973215833	3
975831713	3	978949117	2	980766497	3
983879921	3	985918807	3	986608921	29
991136977	5	991752599	13	997137961	11
1003911991	3	1009775293	2	1012423549	6
1021000537	5	1023976897	7	1024153643	2
1037027287	3	1038812881	11	1044754639	3
1045125617	3	1047411427	3	1047753349	6

4.1.4 $\sim 1 \times 10^{18}$

p	$g(p)$	p	$g(p)$
951970612352230049	3	963284339889659609	3
967495386904694119	3	969751761517096213	2
983238274281901499	2	984647442475101409	23
989286107138674069	11	1002507954383424641	3
1006658951440146419	2	1020152326159075903	3
1034876265966119449	7	1042753851435034019	2
1043609016597371563	2	1045571042176595707	2
1048364250160580293	2	1049495624119026949	2