NANJING UNIVERSITY

ACM-ICPC Codebook 2

Number Theory
Linear Algebra
Combinatorics

August 2, 2018

Contents

1 Number Theory

1.1 Modulo operations

1.1.1 Modular exponentiation (fast power-mod)

Calculate $b^e \bmod m$.

Time complexity: $O(\log e)$

```
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.

⚠ 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 }
14
15 LL mulmod(LL a, LL b) {
16     LL tmp = (a * b - (LL)((long double)a/p*b + 1e-8)*p);
17     return tmp < 0 ? tmp + p : tmp;
18 }
```

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

```
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

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

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.

```
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 ($a^{p-1} \equiv 1 \mod p$), when $m = p$ is a prime, the multiplicative inverse can also be written as $a^{-1} = \left(a^{p-2} \mod p\right)$.

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,   when $n < 2^{64}$.
9780504, 1795265022}

⚠ 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 ith 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:

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: $O(n \log n)$.

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

```
18              }
19              return r;
20          }
21
22      NTT(int k){
23              K = k; N = 1 << k;
24              g = powmod(G, (P-1)/N);
25              g_inv = powmod(g, N-1);
26              omega[0] = oinv[0] = 1;
27              rep (i, N){
28                  rev[i] = (rev[i>>1]>>1) | ((i&1)<<(K-1));
29                  if (i){
30                      omega[i] = omega[i-1] * g % P;
31                      oinv[i] = oinv[i-1] * g_inv % P;
32                  }
33              }
34          }
35
36      void _ntt(LL* a, LL* w){
37              rep (i, N) if (i < rev[i]) swap(a[i], a[rev[i]]);
38              for (int l = 2; l <= N; l *= 2){
39                  int m = l/2;
40                  for (LL* p = a; p != a + N; p += l)
41                      rep (k, m){
42                          LL t = w[N/l*k] * p[k+m] % P;
43                          p[k+m] = (p[k] - t + P) % P;
44                          p[k] = (p[k] + t) % P;
45                      }
46              }
47          }
48
49      void ntt(LL* a){_ntt(a, omega);}
50      void intt(LL* a){
51              LL inv = powmod(N, P-2);
52              _ntt(a, oinv);
53              rep (i, N) a[i] = a[i] * inv % P;
54          }
55
56      void conv(LL* a, LL* b){
57              ntt(a); ntt(b);
58              rep (i, N) a[i] = a[i] * b[i] % P;
59              intt(a);
60          }
61  };
```

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: $O(n \log n)$.

```
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:

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

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

```
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, 0, sizeof(r.m));
```

```
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   }
```

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.

```
1   const int MAXD = 30;
2   struct linearbasis {
3       ULL b[MAXD] = {};
4
5       bool insert(ll v) {
6           for (int j = MAXD - 1; j >= 0; j--) {
7               if (!(v & (1ll << j))) continue;
8               if (b[j]) v ^= b[j];
9               else {
10                  for (int k = 0; k < j; k++)
11                      if (v & (1ll << k)) v ^= b[k];
12                  for (int k = j + 1; k < MAXD; k++)
13                      if (b[k] & (1ll << j)) b[k] ^= v;
14                  b[j] = v;
15                  return true;
16              }
17          }
18          return false;
19      }
20  };
```

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:

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: $O(n)$

```
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(sz);
7       rep (i, sz) res[i] = rhs[lhs[i]];
8       return res;
9   }
10
11  perm operator ~ (const perm lhs){
12      int sz = lhs.size();
13      perm res(sz);
14      rep (i, sz) res[lhs[i]] = i;
15      return res;
16  }
17
18  struct permutation{
19      int size;
20      vector<vector<int>> orbits;
21
22      permutation(perm p){
```

```
23          size = p.size();
24          vector<bool> visited(size);
25          rep (i, size) {
26              if (visited[i]) continue;
27              int cur = i;
28              vector<int> orbit;
29              while (!visited[cur]){
30                  visited[cur] = true;
31                  orbit.push_back(cur);
32                  cur = p[cur];
33              }
34              orbits.push_back(move(orbit));
35          }
36      }
37  };
```

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.

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