

1 Mathematics

1.1 Useful Data

n	Primes less than n	Maximal Prime Gap	$\max_{0 < i < n}(d(i))$
1e2	25	8	12
1e3	168	20	32
1e4	1229	36	64
1e5	9592	72	128
1e6	78.498	114	240
1e7	664.579	154	448
1e8	5.761.455	220	768
1e9	50.487.534	282	1344

1.2 Modular Arithmetic

1.2.1 Chinese Remainder Theorem

```

1
2 ll inline mod(ll x, ll m) { return ((x % m) < 0) ? x + m : x; }
3 ll inline mul(ll x, ll y, ll m) { return (x * y) % m; }
4 ll inline add(ll x, ll y, ll m) { return (x + y) % m; }
5
6 // extended euclidean algorithm
7 // finds g, x, y such that
8 // a * x + b * y = g = GCD(a,b)
9 ll gcdext(ll a, ll b, ll &x, ll &y)
10 {
11     ll r2, x2, y2, r1, x1, y1, r0, x0, y0, q;
12     r2 = a, x2 = 1, y2 = 0;
13     r1 = b, x1 = 0, y1 = 1;
14     while (r1)
15     {
16         q = r2 / r1;
17         r0 = r2 % r1;
18         x0 = x2 - q * x1;
19         y0 = y2 - q * y1;
20         r2 = r1, x2 = x1, y2 = y1;
21         r1 = r0, x1 = x0, y1 = y0;
22     }
23     ll g = r2;
24     x = x2, y = y2;
25     if (g < 0)
26         g = -g, x = -x, y = -y; // make sure g > 0
27     // for debugging (in case you think you might have bugs)
28     // assert (g == a * x + b * y);
29     // assert (g == __gcd(abs(a),abs(b)));
30     return g;
31 }
32
33 // =====
34 // CRT for a system of 2 modular linear equations

```

```

35 // =====
36 // We want to find X such that:
37 // 1) x = r1 (mod m1)
38 // 2) x = r2 (mod m2)
39 // The solution is given by:
40 // sol = r1 + m1 * (r2-r1)/g * x' (mod LCM(m1,m2))
41 // where x' comes from
42 // m1 * x' + m2 * y' = g = GCD(m1,m2)
43 // where x' and y' are the values found by extended euclidean
44 // algorithm (gcdext)
45 // Useful references:
46 // https://codeforces.com/blog/entry/61290
47 // https://forthright48.com/chinese-remainder-theorem-part-1-coprime-
48 // moduli
49 // https://forthright48.com/chinese-remainder-theorem-part-2-non-
50 // coprime-moduli
51 // ** Note: this solution works if lcm(m1,m2) fits in a long long (64
52 // bits)
53 pair<ll, ll> CRT(ll r1, ll m1, ll r2, ll m2)
54 {
55     ll g, x, y;
56     g = gcdext(m1, m2, x, y);
57     if ((r1 - r2) % g != 0)
58         return {-1, -1}; // no solution
59     ll z = m2 / g;
60     ll lcm = m1 * z;
61     ll sol = add(mod(r1, lcm), m1 * mul(mod(x, z), mod((r2 - r1) / g, z)
62         , z), lcm);
63     // for debugging (in case you think you might have bugs)
64     // assert (0 <= sol and sol < lcm);
65     // assert (sol % m1 == r1 % m1);
66     // assert (sol % m2 == r2 % m2);
67     return {sol, lcm}; // solution + lcm(m1,m2)
68 }
69
70 // =====
71 // CRT for a system of N modular linear equations
72 // =====
73 // Args:
74 // r = array of remainders
75 // m = array of modules
76 // n = length of both arrays
77 // Output:
78 // a pair {X, lcm} where X is the solution of the sytemm
79 // X = r[i] (mod m[i]) for i = 0 ... n-1
80 // and lcm = LCM(m[0], m[1], ..., m[n-1])
81 // if there is no solution, the output is {-1, -1}
82 // ** Note: this solution works if LCM(m[0],...,m[n-1]) fits in a long
83 // long (64 bits)
84 pair<ll, ll> CRT(ll *r, ll *m, int n)
85 {
86     ll r1 = r[0], m1 = m[0];
87     repx(i, 1, n)
88     {
89         ll r2 = r[i], m2 = m[i];

```

```

84     ll g, x, y;
85     g = gcdext(m1, m2, x, y);
86     if ((r1 - r2) % g != 0)
87         return {-1, -1}; // no solution
88     ll z = m2 / g;
89     ll lcm = m1 * z;
90     ll sol = add(mod(r1, lcm), m1 * mul(mod(x, z), mod((r2 - r1) / g
91         , z), z), lcm);
92     r1 = sol;
93     m1 = lcm;
94 }
95 // for debugging (in case you think you might have bugs)
96 // assert (0 <= r1 and r1 < m1);
97 // rep(i, n) assert (r1 % m[i] == r[i]);
98 return {r1, m1};
99 }

```

1.2.2 Binomial Coefficients mod m

```

1  #include "../CRT/CRT.cpp"
2  #include "../primalityChecks/millerRabin/millerRabin.cpp"
3  #include "../primalityChecks/sieveEratosthenes/sieve.cpp"
4
5  // Modular computation of nCr using lucas theorem, granville theorem and
6  CRT
7
8  ll num; //Set num to the corresponding mod for the
9  nCr calculations
10 umap<ll, int> MOD; //MOD[P]=V_p(mod)
11 umap<ll, vector<ll>> FMOD; //n! mod p if MOD[p]=1 else the product of
12 all i mod P^MOD[P], where 1<=i<=n and (i,p)=1
13 umap<ll, vector<ll>> invFMOD; //the inverse of FMOD[n] in the
14 corresponding MOD
15
16 void preCompute()
17 {
18     // Factor mod->MOD
19     vi primes = sieve(num);
20     ll m = num;
21     for (auto p : primes)
22     {
23         if (p * p > m)
24             break;
25         while (m % p == 0)
26         {
27             MOD[p]++;
28             if ((m /= p) == 1)
29                 goto next;
30         }
31     }
32     if (m > 1)
33         MOD[m] = 1;
34 next:
35     // Compute FMOD and invFMOD
36     for (auto p : MOD)

```

```

33 {
34     int m = pow(p.first, p.second); //p^V_p(n)
35     FMOD[p.first].assign(m, 1);
36     invFMOD[p.first].assign(m, 1);
37     repx(i, 2, FMOD[p.first].size())
38     {
39         if (i % p.first == 0 and p.second > 1)
40             FMOD[p.first][i] = FMOD[p.first][i - 1];
41         else
42             FMOD[p.first][i] = mul(FMOD[p.first][i - 1], i, FMOD[p.
43                 first].size());
44
45         //Compute using Euler's theorem i.e. a^phi(m)=1 mod m with (
46             a,m)=1
47         invFMOD[p.first][i] = fastPow(FMOD[p.first][i], m / p.first
48             * (p.first - 1) - 1, m);
49     }
50 }
51
52 // Compute nCr using Granville's theorem (prime powers)
53 // Auxiliary functions
54
55 // V_p(n!) using Legendre's theorem
56 int V(ll n, int p)
57 {
58     int e = 0;
59     while ((n /= p) > 0)
60         e += n;
61     return e;
62 }
63
64 //
65 ll f(ll n, ll p)
66 {
67     ll m = pow(p, MOD[p]);
68     int e = n / m;
69     return mul(fastPow(FMOD[p][m - 1], e, m), FMOD[p][n % m], m);
70 }
71
72 ll F(ll n, ll p)
73 {
74     ll m = pow(p, MOD[p]);
75     ll ans = 1;
76     do
77     {
78         ans = mul(ans, f(n, p), m);
79     } while ((n /= p) > 0);
80     return ans;
81 }
82
83 // Granville theorem
84 ll granville(ll n, ll r, int p)
85 {
86     int e = V(n, p) - V(n - r, p) - V(r, p);
87     ll m = pow(p, MOD[p]);
88     if (e >= MOD[p])

```

```

85     return 0;
86     ll ans = fastPow(p, e, m);
87     ans = mul(ans, F(n, p), m);
88     ans = mul(ans, fastPow(F(r, p), pow(p, MOD[p] - 1) * (p - 1) - 1, m),
89               m);
89     ans = mul(ans, fastPow(F(n - r, p), pow(p, MOD[p] - 1) * (p - 1) -
90               1, m), m);
90     return ans;
91 }
92
93 // Compute nCr using Lucas theorem (primes)
94 ll lucas(ll n, ll r, int p)
95 {
96     // Trivial cases
97     if (r > n or r < 0)
98         return 0;
99     if (r == 0 or n == r)
100         return 1;
101     if (r == 1 or r == n - 1)
102         return n % p;
103     // Base case
104     if (n < p and r < p)
105     {
106         ll ans = mul(invFMOD[p][r], invFMOD[p][n - r], p); // 1/(r!(n-r)
107         ans = mul(ans, FMOD[p][n], p); // n!/(r!(n-r)
108         return ans;
109     }
110     ll ans = lucas(n / p, r / p, p); //Recursion
111     ans = mul(ans, lucas(n % p, r % p, p), p); //False recursion
112     return ans;
113 }
114
115 // Given the prime decomposition of mod;
116 ll nCr(ll n, ll r)
117 {
118     // Trivial cases
119     if (n < r or r < 0)
120         return 0;
121     if (r == 0 or r == n)
122         return 1;
123     if (r == 1 or r == n - 1)
124         return (n % num);
125     // Non-trivial cases
126     ll ans = 0;
127     ll mod = 1;
128     for (auto p : MOD)
129     {
130         ll temp = pow(p.first, p.second);
131         if (p.second > 1)
132         {
133             ans = CRT(ans, mod, granville(n, r, p.first), temp).first;
134         }
135         else

```

```

136     {
137         ans = CRT(ans, mod, lucas(n, r, p.first), temp).first;
138     }
139     mod *= temp;
140 }
141 return ans;
142 }

```

1.3 Primality Checks

1.3.1 Miller Rabin

```

1
2 ll mulmod(ull a, ull b, ull c)
3 {
4     ull x = 0, y = a % c;
5     while (b)
6     {
7         if (b & 1)
8             x = (x + y) % c;
9         y = (y << 1) % c;
10        b >>= 1;
11    }
12    return x % c;
13 }
14
15 ll fastPow(ll x, ll n, ll MOD)
16 {
17     ll ret = 1;
18     while (n)
19     {
20         if (n & 1)
21             ret = mulmod(ret, x, MOD);
22         x = mulmod(x, x, MOD);
23         n >>= 1;
24     }
25     return ret;
26 }
27
28 bool isPrime(ll n)
29 {
30     vi a = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
31
32     if (binary_search(a.begin(), a.end(), n))
33         return true;
34
35     if ((n & 1) == 0)
36         return false;
37
38     int s = 0;
39     for (ll m = n - 1; !(m & 1); ++s, m >>= 1)
40         ;
41
42     int d = (n - 1) / (1 << s);
43

```

```

44   for (int i = 0; i < 7; i++)
45   {
46       ll fp = fastPow(a[i], d, n);
47       bool comp = (fp != 1);
48       if (comp)
49           for (int j = 0; j < s; j++)
50           {
51               if (fp == n - 1)
52               {
53                   comp = false;
54                   break;
55               }
56
57               fp = mulmod(fp, fp, n);
58           }
59       if (comp)
60           return false;
61   }
62   return true;
63 }

```

1.3.2 Sieve of Eratosthenes

```

1
2 // O(n log log n)
3 vi sieve(int n)
4 {
5     vi primes;
6
7     vector<bool> is_prime(n + 1, true);
8     int limit = (int)floor(sqrt(n));
9     repx(i, 2, limit + 1) if (is_prime[i]) for (int j = i * i; j <= n; j
10         += i)
11         is_prime[j] = false;
12
13     repx(i, 2, n + 1) if (is_prime[i]) primes.eb(i);
14
15     return primes;
16 }

```

1.3.3 trialDivision

```

1
2 // O(sqrt(n)/log(sqrt(n))+log(n))
3 vi trialDivision(int n, vi &primes)
4 {
5     vi factors;
6     for (auto p : primes)
7     {
8         if (p * p > n)
9             break;
10        while (n % p == 0)
11        {
12            primes.pb(p);
13            if ((n /= p) == 1)

```

```

14            return factors;
15        }
16    }
17    if (n > 1)
18        factors.pb(n);
19
20    return factors;
21 }

```

1.4 Others

1.4.1 Polynomials

```

1
2 template <class T>
3 class Pol
4 {
5 private:
6     vector<T> cofs;
7     int n;
8
9 public:
10    Pol(vector<T> cofs) : cofs(cofs)
11    {
12        this->n = cofs.size() - 1;
13    }
14
15    Pol<T> operator+(const Pol<T> &o)
16    {
17        vector<T> n_cofs;
18        if (n > o.n)
19        {
20            n_cofs = cofs;
21            rep(i, o.n + 1)
22            {
23                n_cofs[i] += o.cofs[i];
24            }
25        }
26        else
27        {
28            n_cofs = o.cofs;
29            rep(i, n + 1)
30            {
31                n_cofs[i] += cofs[i];
32            }
33        }
34        return Pol(n_cofs);
35    }
36
37    Pol<T> operator-(const Pol<T> &o)
38    {
39        vector<T> n_cofs;
40        if (n > o.n)
41        {
42            n_cofs = cofs;

```

```

43     rep(i, o.n + 1)
44     {
45         n_cofs[i] -= o.cofs[i];
46     }
47 }
48 else
49 {
50     n_cofs = o.cofs;
51     rep(i, n + 1)
52     {
53         n_cofs[i] *= -1;
54         n_cofs[i] += cofs[i];
55     }
56 }
57 return Pol(n_cofs);
58 }
59
60 Pol<T> operator*(const Pol<T> &o) //Use Fast Fourier Transform when
    we implement it
61 {
62     vector<T> n_cofs(n + o.n + 1);
63     rep(i, n + 1)
64     {
65         rep(j, o.n + 1)
66         {
67             n_cofs[i + j] += cofs[i] * o.cofs[j];
68         }
69     }
70     return Pol(n_cofs);
71 }
72
73 Pol<T> operator*(const T &o)
74 {
75     vector<T> n_cofs = cofs;
76     for (auto &cof : n_cofs)
77     {
78         cof *= o;
79     }
80     return Pol(n_cofs);
81 }
82
83 double operator()(double x)
84 {
85     double ans = 0;
86     double temp = 1;
87     for (auto cof : cofs)
88     {
89         ans += (double)cof * temp;
90         temp *= x;
91     }
92     return ans;
93 }
94
95 Pol<T> integrate()
96 {

```

```

97     vector<T> n_cofs(n + 2);
98     repx(i, 1, n_cofs.size())
99     {
100         n_cofs[i] = cofs[i - 1] / T(i);
101     }
102     return Pol<T>(n_cofs);
103 }
104
105 double integrate(T a, T b)
106 {
107     Pol<T> temp = integrate();
108     return temp(b) - temp(a);
109 }
110
111 friend ostream &operator<<(ostream &str, const Pol &a);
112 };
113
114 ostream &operator<<(ostream &strm, const Pol<double> &a)
115 {
116     bool flag = false;
117     rep(i, a.n + 1)
118     {
119         if (a.cofs[i] == 0)
120             continue;
121
122         if (flag)
123             if (a.cofs[i] > 0)
124                 strm << " + ";
125             else
126                 strm << " - ";
127
128         flag = true;
129         if (i > 1)
130         {
131             if (abs(a.cofs[i]) != 1)
132                 strm << abs(a.cofs[i]);
133             strm << "x^" << i;
134         }
135         else if (i == 1)
136         {
137             if (abs(a.cofs[i]) != 1)
138                 strm << abs(a.cofs[i]);
139             strm << "x";
140         }
141         else
142         {
143             strm << a.cofs[i];
144         }
145     }
146     return strm;
147 }

```

1.4.2 Factorial Factorization

```
2 // O(n)
3 umap<ll, int> factorialFactorization(int n, vi &primes)
4 {
5     umap<ll, int> p2e;
6     for (auto p : primes)
7     {
8         if (p > n)
9             break;
10        int e = 0;
11        ll tmp = n;
12        while ((tmp /= p) > 0)
13            e += tmp;
14        if (e > 0)
15            p2e[p] = e;
16    }
17    return p2e;
18 }
```