

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

```

```

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

```

```

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

```

1.3 Primality Checks

1.3.1 Miller Rabin

```

1  #include "../headers/headers.h"
2
3  ll mulmod(ull a, ull b, ull c)
4  {
5      ull x = 0, y = a % c;
6      while (b)
7      {
8          if (b & 1)
9              x = (x + y) % c;
10             y = (y << 1) % c;
11             b >>= 1;
12         }
13         return x % c;
14     }
15
16     ll fastPow(ll x, ll n, ll MOD)
17     {
18         ll ret = 1;
19         while (n)
20         {
21             if (n & 1)
22                 ret = mulmod(ret, x, MOD);
23             x = mulmod(x, x, MOD);
24             n >>= 1;
25         }
26         return ret;
27     }
28
29     bool isPrime(ll n)
30     {
31         vi a = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
32
33         if (binary_search(a.begin(), a.end(), n))

```

```

34         return true;
35
36         if ((n & 1) == 0)
37             return false;
38
39         int s = 0;
40         for (ll m = n - 1; !(m & 1); ++s, m >>= 1)
41             ;
42
43         int d = (n - 1) / (1 << s);
44
45         for (int i = 0; i < 7; i++)
46         {
47             ll fp = fastPow(a[i], d, n);
48             bool comp = (fp != 1);
49             if (comp)
50                 for (int j = 0; j < s; j++)
51                 {
52                     if (fp == n - 1)
53                     {
54                         comp = false;
55                         break;
56                     }
57
58                     fp = mulmod(fp, fp, n);
59                 }
60             if (comp)
61                 return false;
62         }
63         return true;
64     }

```

1.3.2 Sieve of Eratosthenes

```

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

```

1.3.3 trialDivision

```

1  #include "../headers/headers.h"

```

```

2
3 // O(sqrt(n)/log(sqrt(n))+log(n))
4 vi trialDivision(int n, vi &primes)
5 {
6     vi factors;
7     for (auto p : primes)
8     {
9         if (p * p > n)
10             break;
11         while (n % p == 0)
12             {
13                 primes.pb(p);
14                 if ((n /= p) == 1)
15                     return factors;
16             }
17     }
18     if (n > 1)
19         factors.pb(n);
20
21     return factors;
22 }

```

1.4 Others

1.4.1 Polynomials

```

1 #include "../headers/headers.h"
2
3 template <class T>
4 class Pol
5 {
6 private:
7     vector<T> cofs;
8     int n;
9
10 public:
11     Pol(vector<T> cofs) : cofs(cofs)
12     {
13         this->n = cofs.size() - 1;
14     }
15
16     Pol<T> operator+(const Pol<T> &o)
17     {
18         vector<T> n_cofs;
19         if (n > o.n)
20             {
21                 n_cofs = cofs;
22                 rep(i, o.n + 1)
23                 {
24                     n_cofs[i] += o.cofs[i];
25                 }
26             }
27         else
28             {
29                 n_cofs = o.cofs;

```

```

30         rep(i, n + 1)
31         {
32             n_cofs[i] += cofs[i];
33         }
34     }
35     return Pol(n_cofs);
36 }
37
38 Pol<T> operator-(const Pol<T> &o)
39 {
40     vector<T> n_cofs;
41     if (n > o.n)
42     {
43         n_cofs = cofs;
44         rep(i, o.n + 1)
45         {
46             n_cofs[i] -= o.cofs[i];
47         }
48     }
49     else
50     {
51         n_cofs = o.cofs;
52         rep(i, n + 1)
53         {
54             n_cofs[i] *= -1;
55             n_cofs[i] += cofs[i];
56         }
57     }
58     return Pol(n_cofs);
59 }
60
61 Pol<T> operator*(const Pol<T> &o) //Use Fast Fourier Transform when
62     we implement it
63 {
64     vector<T> n_cofs(n + o.n + 1);
65     rep(i, n + 1)
66     {
67         rep(j, o.n + 1)
68         {
69             n_cofs[i + j] += cofs[i] * o.cofs[j];
70         }
71     }
72     return Pol(n_cofs);
73 }
74
75 Pol<T> operator*(const T &o)
76 {
77     vector<T> n_cofs = cofs;
78     for (auto &cof : n_cofs)
79     {
80         cof *= o;
81     }
82     return Pol(n_cofs);
83 }

```

```

84 double operator()(double x)
85 {
86     double ans = 0;
87     double temp = 1;
88     for (auto cof : cofs)
89     {
90         ans += (double)cof * temp;
91         temp *= x;
92     }
93     return ans;
94 }
95
96 Pol<T> integrate()
97 {
98     vector<T> n_cofs(n + 2);
99     repx(i, 1, n_cofs.size())
100     {
101         n_cofs[i] = cofs[i - 1] / T(i);
102     }
103     return Pol<T>(n_cofs);
104 }
105
106 double integrate(T a, T b)
107 {
108     Pol<T> temp = integrate();
109     return temp(b) - temp(a);
110 }
111
112 friend ostream &operator<<(ostream &str, const Pol &a);
113 };
114
115 ostream &operator<<(ostream &strm, const Pol<double> &a)
116 {
117     bool flag = false;
118     rep(i, a.n + 1)
119     {
120         if (a.cofs[i] == 0)
121             continue;
122
123         if (flag)
124             if (a.cofs[i] > 0)
125                 strm << " + ";
126             else
127                 strm << " - ";
128         else
129             flag = true;
130         if (i > 1)
131         {
132             if (abs(a.cofs[i]) != 1)
133                 strm << abs(a.cofs[i]);
134             strm << "x^" << i;
135         }
136         else if (i == 1)
137         {
138             if (abs(a.cofs[i]) != 1)

```

```

139         strm << abs(a.cofs[i]);
140         strm << "x";
141     }
142     else
143     {
144         strm << a.cofs[i];
145     }
146 }
147 return strm;
148 }

```

1.4.2 Factorial Factorization

```

1 #include "../headers/headers.h"
2
3 // O(n)
4 umap<int, int> factorialFactorization(int n, vi &primes)
5 {
6     umap<int, int> p2e;
7     for (auto p : primes)
8     {
9         if (p > n)
10             break;
11         int e = 0;
12         int tmp = n;
13         while ((tmp /= p) > 0)
14             e += tmp;
15         if (e > 0)
16             p2e[p] = e;
17     }
18     return p2e;
19 }

```