Numerical (1)

FFT

Description: Applies the discrete Fourier transform to a sequence of numbers modulo MOD.
Time: $\mathcal{O}(n \log n)$

```
int rev[N], root[N];

void init(int n) {
    static int last_init = -1;
    if (n == last_init) return;
    last_init = n;
    for (int i = 1; i < n; ++i) {
        rev[i] = (rev[i >> 1] >> 1) | (i & 1) * (n >> 1);
    }
    const int root_n = binpow(ROOT, (MOD - 1) / n);
    int cur = 1;
    for (int i = 0, cur = 1; i < n / 2; ++i) {
        root[i + n / 2] = cur;
        cur = mul(cur, root_n);
    }
    for (int i = n / 2 - 1; i >= 0; --i) {
        root[i] = root[i << 1];
    }
}

void dft(int* f, int n, bool inverse = false) {
    init(n);
    for (int i = 0; i < n; ++i) {
        if (i < rev[i]) swap(f[i], f[rev[i]]);
    }
    for (int k = 1; k < n; k <<= 1)
        for (int i = 0; i < n; i += (k << 1))
            for (int j = 0; j < k; ++j) {
                int z = mul(f[i + j + k], root[j + k]);
                f[i + j + k] = sub(f[i + j], z);
                f[i + j] = add(f[i + j], z);
            }
    if (inverse) {
        reverse(f + 1, f + n);
        const int inv_n = inv(n);
        for (int i = 0; i < n; ++i) f[i] = mul(f[i], inv_n);
    }
}
```

Berlekamp-Massey

Description: Returns the polynomial of a recurrent sequence of order n from the first $2n$ terms.
Usage: `berlekamp_massey({0, 1, 1, 3, 5, 11}) // {1, -1, -2}`
Time: $\mathcal{O}(n^2)$

```
vector<int> berlekamp_massey(vector<int> s) {
    int n = sz(s), L = 0, m = 0;
    vector<int> c(n), b(n), t;
    c[0] = b[0] = 1;
    int eval = 1;
    for (int i = 0; i < n; ++i) {
        m++;
        int delta = 0;
        for (int j = 0; j <= L; ++j) {
            delta = add(delta, mul(c[j], s[i - j]));
        }
        if (delta == 0) continue;
        t = c;
        int coef = mul(delta, inv(eval));
        for (int j = m; j < n; ++j) {
            c[j] = sub(c[j], mul(coef, b[j - m]));
        }
        if (2 * L > i) continue;
        L = i + 1 - L, m = 0, b = t, eval = delta;
    }
    c.resize(L + 1);
    return c;
}
```

Flows (2)

Dinitz

Description: Finds maximum flow using Dinitz algorithm.
Time: $\mathcal{O}(n^2 m)$

```
struct Edge {
    int to, cap, flow;
};
vector<Edge> E;
vector<int> gr[N];

int n;
int d[N], ptr[N];
```

```
bool bfs(int v0 = 0, int cc = 1) {
    fill(d, d + n, -1);
    d[v0] = 0;
    vector<int> q{v0};
    for (int st = 0; st < sz(q); ++st) {
        int v = q[st];
        for (int id : gr[v]) {
            auto [to, cp, fl] = E[id];
            if (d[to] != -1 || cp - fl < cc) continue;
            d[to] = d[v] + 1;
            q.emplace_back(to);
        }
    }
    return d[n - 1] != -1;
}

int dfs(int v, int flow, int cc = 1) {
    if (v == n - 1 || !flow) return flow;
    for (; ptr[v] < sz(gr[v]); ++ptr[v]) {
        auto [to, cp, fl] = E[gr[v][ptr[v]]];
        if (d[to] != d[v] + 1 || cp - fl < cc) continue;
        int pushed = dfs(to, min(flow, cp - fl), cc);
        if (pushed) {
            int id = gr[v][ptr[v]];
            E[id].flow += pushed;
            E[id ^ 1].flow -= pushed;
            return pushed;
        }
    }
    return 0;
}

ll dinitz() {
    ll flow = 0;
    for (int c = INF; c > 0; c >>= 1) {
        while (bfs(0, c)) {
            fill(ptr, ptr + n, 0);
            while (int pushed = dfs(0, INF, c))
                flow += pushed;
        }
    }
    return flow;
}
```

MCMF

Description: Finds Minimal Cost Maximal Flow.

```
struct Edge {
    ll to, f, c, w;
};

vector<Edge> E;
vector<int> gr[N];

void add_edge(int u, int v, ll c, ll w) {
    gr[u].push_back(sz(E));
    E.emplace_back(v, 0, c, w);
    gr[v].push_back(sz(E));
    E.emplace_back(u, 0, 0, -w);
}

pair<ll, ll> mcmf(int n) {
    vector<ll> dist(n);
    vector<ll> pr(n);
    vector<ll> phi(n);
    auto dijkstra = [&] {
        fill(all(dist), INF);
        dist[0] = 0;
        priority_queue<pair<ll, int>, vector<pair<ll, int>>,
            greater<>> pq;
        pq.emplace(0, 0);
        while (!pq.empty()) {
            auto [d, v] = pq.top();
            pq.pop();
            if (d != dist[v]) continue;
            for (int idx : gr[v]) {
                if (E[idx].c == E[idx].f) continue;
                int to = E[idx].to;
                ll w = E[idx].w + phi[v] - phi[to];
                if (dist[to] > d + w) {
                    dist[to] = d + w;
                    pr[to] = idx;
                    pq.emplace(d + w, to);
                }
            }
        }
    };

    ll total_cost = 0, total_flow = 0;
    while (true) {
        dijkstra();
        if (dist[n - 1] == INF) break;
        ll min_cap = INF;
        int cur = n - 1;
        while (cur != 0) {
            min_cap = min(min_cap, E[pr[cur]].c - E[pr[cur]].f);
            cur = E[pr[cur] ^ 1].to;
        }
        cur = n - 1;
        while (cur != 0) {
            E[pr[cur]].f += min_cap;
            E[pr[cur] ^ 1].f -= min_cap;
            total_cost += min_cap * E[pr[cur]].w;
            cur = E[pr[cur] ^ 1].to;
        }
        total_flow += min_cap;
        for (int i = 0; i < n; ++i) {
            phi[i] += dist[i];
        }
    }

    return {total_flow, total_cost};
}
```

Number Theory (3)

Extended GCD

Description: Finds two integers x and y, such that $ax + by = gcd(a, b)$.

```
ll exgcd(ll a, ll b, ll &x, ll &y) {
    if (!b) return x = 1, y = 0, a;
    ll d = euclid(b, a % b, y, x);
    return y -= a/b * x, d;
}
```

CRT

Description: Chinese Remainder Theorem. `crt(a, m, b, n)` computes x s.t. $x \equiv a \mod m, x \equiv b \mod n$.
Time: $\mathcal{O}(\log n)$

```
ll crt(ll a, ll m, ll b, ll n) {
    if (n > m) swap(a, b), swap(m, n);
    ll x, y, g = exgcd(m, n, x, y);
    assert((a - b) % g == 0); // else no solution
    x = (b - a) % n * x % n / g * m + a;
    return x < 0 ? x + m * n / g : x;
}
```

Strings (4)

KMP

Description: Calculates prefix function and Z-function of the given string.
Time: $\mathcal{O}(n)$

```
vector<int> pi(const string& s) {
  vector<int> p(sz(s));
  for (int i = 1; i < sz(s); ++i) {
    int g = p[i - 1];
    while (g && s[i] != s[g]) g = p[g - 1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

vector<int> zf(const string& s) {
  vector<int> z(sz(s));
  int l = -1, r = -1;
  for (int i = 1; i < sz(s); ++i) {
    z[i] = i >= r ? 0 : min(r - i, z[i - l]);
    while (i + z[i] < sz(s) && s[i + z[i]] == s[z[i]]) ++z[i];
    if (i + z[i] > r) l = i, r = i + z[i];
  }
  return z;
}
```

Suffix array

Description: Calculates suffix array, inverse suffix array and LCP array of the given string.
Time: $\mathcal{O}(n \log n)$

```
const int M = 1e5 + 10;
vector<int> sa, pos, lcp;

void suffix_array(string& s) {
    int n = sz(s);
    vector<int> c(n), cur(n);
    sa.resize(n), pos.resize(n), lcp.resize(n);
    for (int i = 0; i < n; ++i) {
        sa[i] = i, c[i] = s[i];
    }
    sort(all(sa), [&](int i, int j) { return c[i] < c[j]; });
    vector<int> cnt(M);
    for (int k = 1; k < n; k <<= 1) {
        fill(all(cnt), 0);
        for (int x : c) cnt[x]++;
        for (int i = 1; i < M; ++i) cnt[i] += cnt[i - 1];
        for (int i : sa) {
            int c2 = c[(i - k + n) % n] - 1;
            cur[cnt[c2]++] = (i - k + n) % n;
        }
        swap(cur, sa);
        int x = -1, y = -1, p = 0;
        for (int i : sa) {
            if (c[i] != x || c[(i + k) % n] != y) {
                x = c[i], y = c[(i + k) % n], p++;
            }
            cur[i] = p;
        }
        swap(cur, c);
    }
    for (int i = 0; i < n; ++i) pos[sa[i]] = i;
    int l = 0;
    for (int i = 0; i < n; ++i) {
        if (pos[i] == n - 1) {
            l = 0;
        } else {
            while (s[(i + l) % n] == s[(sa[pos[i] + 1] + l) % n])
                ++l;
            lcp[pos[i]] = l;
            l = max(0, l - 1);
        }
    }
}
```

Minimal rotation

Description: Rotates the given string until it is lexicographically minimal, returns shift.
Time: $\mathcal{O}(n)$

```
int min_rotation(string& s, int len) {
    s += s;
    int i = 0, ans = 0;
    while (i < len) {
        ans = i;
        int j = i + 1, k = i;
        while (j < len * 2 && s[k] <= s[j]) {
            if (s[k] < s[j]) {
                k = i;
            } else {
                k += 1;
            }
            j += 1;
        }
        while (i <= k) {
            i += j - k;
        }
    }
    s = s.substr(ans, len);
    return ans;
}
```

Miscellaneous (5)

Integrate

Description: Function integration over an interval using Simpson's rule. The error is proportional to h^4.

```
double integrate(double a, double b, auto&& f, int n = 1000) {
    double h = (b - a) / 2 / n, rs = f(a) + f(b);
    for (int i = 1; i < n * 2; ++i) {
        rs += f(a + i * h) * (i & 1 ? 4 : 2);
    }
    return rs * h / 3;
}
```

Fractional binary search

Description: Finds the smallest fraction $p/q \in [0,1]$ s.t. $f(p/q)$ is true and $p, q \leq N$.
Time: $\mathcal{O}(\log N)$

```
struct frac { ll p, q; };

frac fracBS(auto&& f, ll N) {
    bool dir = true, A = true, B = true;
    frac lo{0, 1}, hi{1, 1}; // Set hi to 1/0 to search (0, N]
    if (f(lo)) return lo;
    assert(f(hi));
    while (A || B) {
        ll adv = 0, step = 1;
        for (int si = 0; step; (step *= 2) >>= si) {
            adv += step;
            frac mid{lo.p * adv + hi.p, lo.q * adv + hi.q};
            if (abs(mid.p) > N || mid.q > N || dir == !f(mid)) {
                adv -= step; si = 2;
            }
        }
        hi.p += lo.p * adv;
        hi.q += lo.q * adv;
        dir = !dir;
        swap(lo, hi);
        A = B; B = !!adv;
    }
    return dir ? hi : lo;
}
```