Moscow IPT

Ctrl-XD

Kostyleb Gleb, Pervutinskiy Roman, Ragulin Vladimir

Numerical (1)

1.1 Newton's method

To compute $B = \frac{1}{A}$ modulo x^m: define $B_1 = \{inv(A[0])\}$ and $B_{2n} = B_n(2 - A \cdot B_n)$.

To compute $B = \sqrt{A}$ modulo x^m: define $B_1 = \{\sqrt{A[0]}\}$ and $B_{2n} = \frac{B_n}{2} + \frac{A}{2B_n}$.

To compute $B = \log(1 + xA)$ modulo x^m: $B = \int \frac{(1+xA)'}{1+xA}$.

If T is EGF for some objects then $C = -\log(1-T) = \sum_{k=1}^{+\infty} \frac{T^k}{k}$ is EGF for cycles of them.

To compute $B = e^{xA}$ modulo x^m: define $B_1 = \{1\}$ and $B_{2n} = B_n(1 + A - \log B_n)$.

If T is EGF for some objects then $F = e^T = \sum_{k=0}^{\infty} \frac{T^k}{k!}$ is EGF for their unordered combinations.

In general case you have equation $P(B, x) = 0$. (f.e. $\log B - A(x) = 0$).

The transition is $B_{2n-\alpha} = B_n - \frac{P(B,x)}{P'_B(B,x)}$. (if $P'_B(B, x)$ is divisible by x^α).

1.2 Additional modulos for FFT

Modulo	Other form	Roots
998244353	$(119 << 23) + 1$	$3, 62$
167772161	$(5 << 25) + 1$	$3, 62$
469762049	$(7 << 26) + 1$	$3, 62$
1004535809	$(479 << 21) + 1$	$3, 62$
1012924417	$(483 << 21) + 1$	62

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);
    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] = add(f[i + j], MOD - 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);
    }
}
```

Middle product

Description: Calculates middle-product of two arrays using Tellegen's principle.
Time: $\mathcal{O}(n \log n)$

```
vector<int> mulT(const vector<int>& a, const vector<int>& b) {
    int n = sz(a), m = sz(b), k = 1;
    while (k < n) k <<= 1;
    fill_n(fft1, k, 0), fill_n(fft2, k, 0);
    copy(all(a), fft1), copy(all(b), fft2);
    dft(fft1, k, true), dft(fft2, k);
    for (int i = 0; i < k; ++i) fft1[i] = mul(fft1[i], fft2[i]);
    dft(fft1, k);
    return {fft1, fft1 + n - m + 1};
}
```

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: $\mathcal{O}\left(n^2\right)$

```
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;
}
```

1.3 Linear recurrence

Let A be generating function for our recurrence, C be its characteristic polynomial and $k = |C|$.

Let $D = C \cdot A$. Then $D \bmod x^k = D$

$$A = \frac{(A \bmod x^k)C \bmod x^k}{C} = \frac{A_0 C \bmod x^k}{C}$$

$$[x^n]\frac{P(x)}{Q(x)} = [x^n]\frac{P(x)Q(-x)}{Q(x)Q(-x)}$$

Flows (2)

Dinitz

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

```
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;
}
```

Miller-Rabin

Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$
Time: 7 times the complexity of $a^b \mod c$.

```
bool is_prime(ll n) {
    if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
    ll A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
    ll s = __builtin_ctzll(n - 1), d = n >> s;
    for (ll a : A) {
        ll p = binpow(a % n, d, n), i = s;
        while (p != 1 && p != n - 1 && a % n && i--)
            p = mul(p, p, n);
        if (p != n - 1 && i != s) return 0;
    }
    return 1;
}
```

Pollard-Rho

Description: Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order.
Time: $\mathcal{O}\left(n^{1/4}\right)$, less for numbers with small factors.

```
ll pollard(ll n) {
    auto f = [n](ll x) { return mul(x, x, n) + 1; };
    ll x = 0, y = 0, t = 30, prd = 2, i = 1, q;
    while (t++ % 40 || __gcd(prd, n) == 1) {
        if (x == y) x = ++i, y = f(x);
        if ((q = mul(prd, max(x,y) - min(x,y), n))) prd = q;
        x = f(x), y = f(f(y));
    }
    return __gcd(prd, n);
}
```

```
vector<ll> factor(ll n) {
    if (n == 1) return {};
    if (is_prime(n)) return {n};
    ll x = pollard(n);
    auto l = factor(x), r = factor(n / x);
    l.insert(l.end(), all(r));
    return l;
}
```

3.1 Möbius function

$$\mu(n) = \begin{cases} 0, n \text{ is not square free} \\ 1, n \text{ has even number of prime factors} \\ -1, n \text{ has odd number of prime factors} \end{cases}$$

$$\sum_{d|n} \mu(d) = \text{int}(n = 1)$$

Möbius inversion:

$$g(n) = \sum_{d|n} f(d) \iff f(n) = \sum_{d|n} \mu(d) g(n/d) = \sum_{d|n} \mu(n/d) g(d)$$

3.2 Sums of powers

$$\sum_{s=1}^{n} s^k = \sum_{s=1}^{k} s! S(k,s) C_{n+1}^{s+1}$$

$\sum_{s=1}^n s^1 = 1\frac{(n+1)n}{2!}$

$\sum_{s=1}^n s^2 = 1\frac{(n+1)n}{2!} + 2\frac{(n+1)n(n-1)}{3!} = \frac{n(n+1)(2n+1)}{6}$
$\sum_{s=1}^n s^3 = 1\frac{(n+1)n}{2!} + 6\frac{(n+1)n(n-1)}{3!} + 6\frac{(n+1)n(n-1)(n-2)}{4!} = \frac{(n+1)^2 n^2}{4}$

3.3 Floor sum

$$f(a,b,c,n) = \sum_{x=0}^{n} \left\lfloor \frac{ax+b}{c} \right\rfloor$$

Let $m = \left\lfloor \frac{an+b}{c} \right\rfloor$. When $a < c$ and $b < c$, we have:

$$f(a,b,c,n) = mn - f\left(c, c-b-1, a, m-1\right)$$

Otherwise:

$$f(a,b,c,n) = \frac{n(n+1)}{2}\left\lfloor\frac{a}{c}\right\rfloor + (n+1)\left\lfloor\frac{b}{c}\right\rfloor + f(a \bmod c, b \bmod c, c, n)$$

3.4 Square of floor sum

$$g(a,b,c,n) = \sum_{x=0}^{n} x\left\lfloor\frac{ax+b}{c}\right\rfloor$$

$$h(a,b,c,n) = \sum_{x=0}^{n}\left\lfloor\frac{ax+b}{c}\right\rfloor^2$$

Let $m = \left\lfloor\frac{an+b}{c}\right\rfloor$. When $a < c$ and $b < c$, we have:

$$g(a,b,c,n) = \frac{mn(n+1)}{2} - \frac{f(c,c-b-1,a,m-1)}{2} - \frac{h(c,c-b-1,a,m-1)}{2}$$

$$h(a,b,c,n) = mn(m+1) - 2g(c,c-b-1,a,m-1) - 2f(c,c-b-1,a,m-1) \\ -f(a,b,c,n)$$

Otherwise:

$$g(a,b,c,n) = \frac{n(n+1)(2n+1)}{6}\left\lfloor\frac{a}{c}\right\rfloor + \frac{n(n+1)}{2}\left\lfloor\frac{b}{c}\right\rfloor + g(a \bmod c, b \bmod c, c, n)$$

$$h(a,b,c,n) = \frac{n(n+1)(2n+1)}{6}\left\lfloor\frac{a}{c}\right\rfloor^2 + (n+1)\left\lfloor\frac{b}{c}\right\rfloor^2 + n(n+1)\left\lfloor\frac{a}{c}\right\rfloor\left\lfloor\frac{b}{c}\right\rfloor + \\ +h(a \bmod c, b \bmod c, c, n) + 2\left\lfloor\frac{b}{c}\right\rfloor f(a \bmod c, b \bmod c, c, n) + \\ +2\left\lfloor\frac{a}{c}\right\rfloor g(a \bmod c, b \bmod c, c, n)$$

Combinatorics (4)

4.1 Derangements

Number of n-permutations that none of the elements appears in their original position.

$$D_n = (n-1)(D_{n-1} + D_{n-2}) = nD_{n-1} + (-1)^n \approx \frac{n!}{e}$$

4.2 Burnside's lemma

Given a group G of symmetries and a set Ω, the number of elements of Ω up to symmetry equal

$$\frac{1}{|G|}\sum_{g\in G} N(g)$$

where $N(g)$ is the number of elements fixed by g ($g(x) = x$).

4.3 Stirling numbers (first kind)

Unsigned Stirling numbers of the first kind $c_{n,k}$ is the number of permutations of n elements with k cycles as well as the coefficient on x^k in the expansion $x(x+1)(x+2)\ldots(x+(n-1))$.

Signed Stirling numbers of the first kind $s_{n,k}$ is the coefficient on x^k in the expansion $x(x-1)(x-2)\ldots(x-(n-1))$.

$$c_{0,0} = s_{0,0} = 1 \quad c_{k,0} = s_{k,0} = 0 \quad c_{n,k} = s_{n,k} = 0 \text{ for } k > n$$

$$c_{n,k} = c_{n-1,k-1} + (n-1)c_{n-1,k}$$

$$s_{n,k} = s_{n-1,k-1} - (n-1)s_{n-1,k}$$

$$s_{n,k} = (-1)^{n+k}c_{n,k}$$

$$\text{EGF: } \sum_{n=0}^{\infty}\sum_{k=0}^{n} s_{n,k}\frac{x^n}{n!}y^k = (1+x)^y$$

$$\text{EGF: } \sum_{n=k}^{\infty} s_{n,k}\frac{x^n}{n!} = \frac{(\log(1+x))^k}{k!}$$

4.4 Stirling numbers (second kind)

The Stirling numbers of the second kind $S(n,k)$, count the number of ways to partition a set of n labelled objects into k nonempty unlabelled subsets.

$$S(n,n) = 1 \text{ for } n \geq 0 \quad S(0,n) = S(n,0) = 0 \text{ for } n > 0$$

$$S(n+1,k) = k \cdot S(n,k) + S(n,k-1)$$

$$S(n,k) = \sum_{t=0}^{k}\frac{(-1)^{k-t}t^n}{(k-t)!t!}$$

$$\text{EGF: } \sum_{n=0}^{\infty}\sum_{k=0}^{n} S(n,k)\frac{x^n}{n!}y^k = e^{y(e^x-1)}$$

$$\text{EGF: } \sum_{n=k}^{\infty} S(n,k)\frac{x^n}{n!} = \frac{(e^x-1)^k}{k!}$$

4.5 Bell numbers

Bell number B_n is the number of partitions of n labeled elements.

$$B_0 = B_1 = 1$$

$$B_n = \sum_{k=0}^{n-1} C_n^k B_k = \sum_{k=0}^{n} S(n,k)$$

$$\text{EGF: } \sum_{n=1}^{\infty} \frac{B_n}{n!}x^n = e^{e^x-1}$$

4.6 Narayana numbers

Narayana number $N(n,k)$ is the number of correct bracket sequences with length $2n$ and exactly k distinct nestings. Also the number of unlabeled ordered rooted trees with $n+1$ vertices and k leaves.

$$N(n,k) = \frac{1}{n}C_n^k C_n^{k-1}$$

4.7 Labeled unrooted trees

Every tree on n vertices has unique sequence of $n-2$ integers from $\{1 \dots n\}$ assosiated with the tree.

Vertex with degree d appears in sequence $d-1$ times.

On n vertices: n^{n-2}.

With degrees $d_1, d_2, \dots, d_n$: $\frac{(n-2)!}{(d_1-1)!\dots(d_n-1)!}$.

Strings (5)

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;
}
```

Aho-Corasick

Description: Builds Aho-Corasick
Time: $\mathcal{O}(nC)$

```
const int C = 26;
struct node {
    int nx[C], first = -1, suff = -1;//, zsuff = -1;
    vector<int> idx;
    node() {
        fill(nx, nx + C, -1);
    }
};
vector<node> t(1);
void add_word(string& s, int id) {
    int v = 0;
    for (char ch : s) {
        int x = ch - 'a';
        if (t[v].nx[x] == -1) {
            t[v].nx[x] = sz(t);
            t.emplace_back();
        }
        v = t[v].nx[x];
    }
```

```
    t[v].idx.emplace_back(id);
}
void build_aho() {
    vector<pair<int, int>> q;
    for (int x = 0; x < C; ++x) {
        if (t[0].nx[x] == -1) {
            t[0].nx[x] = 0;
        } else {
            q.emplace_back(0, x);
        }
    }
    for (int st = 0; st < sz(q); ++st) {
        auto [par, x] = q[st];
        int a = t[par].nx[x];
        if (t[par].suff == -1) {
            t[a].suff = 0;
        } else {
            t[a].suff = t[t[par].suff].nx[x];
            // t[a].zsuff = t[t[a].suff].idx.empty() ? t[t[a].suff
                ].zsuff : t[a].suff;
        }
        for (int y = 0; y < C; ++y) {
            if (t[a].nx[y] == -1) {
                t[a].nx[y] = t[t[a].suff].nx[y];
            } else {
                q.emplace_back(a, y);
            }
        }
    }
}
```

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;
}
```

Graphs (6)

Directed MST

Description: Finds a minimum spanning tree/arborescence of a directed graph, given a root node. If no MST exists, returns -1.
Time: $\mathcal{O}(E \log V)$

```
struct RollbackUF {
    vi e; vector<pii> st;
    RollbackUF(int n) : e(n, -1) {}
    int size(int x) { return -e[find(x)]; }
    int find(int x) { return e[x] < 0 ? x : find(e[x]); }
    int time() { return sz(st); }
    void rollback(int t) {
        for (int i = time(); i --> t;)
            e[st[i].first] = st[i].second;
        st.resize(t);
    }
    bool join(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (e[a] > e[b]) swap(a, b);
        st.push_back({a, e[a]});
        st.push_back({b, e[b]});
        e[a] += e[b]; e[b] = a;
        return true;
    }
};

struct Edge { int a, b; ll w; };
struct Node {
    Edge key;
    Node *l, *r;
    ll delta;
    void prop() {
        key.w += delta;
        if (l) l->delta += delta;
        if (r) r->delta += delta;
        delta = 0;
    }
    Edge top() { prop(); return key; }
};
Node *merge(Node *a, Node *b) {
    if (!a || !b) return a ?: b;
    a->prop(), b->prop();
    if (a->key.w > b->key.w) swap(a, b);
    swap(a->l, (a->r = merge(b, a->r)));
    return a;
}
void pop(Node*& a) { a->prop(); a = merge(a->l, a->r); }

pair<ll, vi> dmst(int n, int r, vector<Edge>& g) {
    RollbackUF uf(n);
    vector<Node*> heap(n);
    for (Edge e : g) heap[e.b] = merge(heap[e.b], new Node{e});
    ll res = 0;
    vi seen(n, -1), path(n), par(n);
    seen[r] = r;
    vector<Edge> Q(n), in(n, {-1,-1}), comp;
    deque<tuple<int, int, vector<Edge>>> cycs;
    rep(s,0,n) {
        int u = s, qi = 0, w;
        while (seen[u] < 0) {
            if (!heap[u]) return {-1,{}};
            Edge e = heap[u]->top();
            heap[u]->delta -= e.w, pop(heap[u]);
            Q[qi] = e, path[qi++] = u, seen[u] = s;
            res += e.w, u = uf.find(e.a);
            if (seen[u] == s) {
                Node* cyc = 0;
                int end = qi, time = uf.time();
                do cyc = merge(cyc, heap[w = path[--qi]]);
                while (uf.join(u, w));
                u = uf.find(u), heap[u] = cyc, seen[u] = -1;
                cycs.push_front({u, time, {&Q[qi], &Q[end]}});
            }
        }
        rep(i,0,qi) in[uf.find(Q[i].b)] = Q[i];
```

```
    }

    for (auto& [u,t,comp] : cycs) { // restore sol (optional)
        uf.rollback(t);
        Edge inEdge = in[u];
        for (auto& e : comp) in[uf.find(e.b)] = e;
        in[uf.find(inEdge.b)] = inEdge;
    }
    rep(i,0,n) par[i] = in[i].a;
    return {res, par};
}
```

Link-Cut

Description: Represents a forest of unrooted trees. You can add and remove edges (as long as the result is still a forest), and check whether two nodes are in the same tree.
Time: All operations take amortized $\mathcal{O}(\log N)$.

```
struct Node { // Splay tree. Root's pp contains tree's parent.
    Node *p = 0, *pp = 0, *c[2];
    bool flip = 0;
    Node() { c[0] = c[1] = 0; fix(); }
    void fix() {
        if (c[0]) c[0]->p = this;
        if (c[1]) c[1]->p = this;
        // (+ update sum of subtree elements etc. if wanted)
    }
    void pushFlip() {
        if (!flip) return;
        flip = 0; swap(c[0], c[1]);
        if (c[0]) c[0]->flip ^= 1;
        if (c[1]) c[1]->flip ^= 1;
    }
    int up() { return p ? p->c[1] == this : -1; }
    void rot(int i, int b) {
        int h = i ^ b;
        Node *x = c[i], *y = b == 2 ? x : x->c[h], *z = b ? y : x;
        if ((y->p = p)) p->c[up()] = y;
        c[i] = z->c[i ^ 1];
        if (b < 2) {
            x->c[h] = y->c[h ^ 1];
            y->c[h ^ 1] = x;
        }
        z->c[i ^ 1] = this;
        fix(); x->fix(); y->fix();
        if (p) p->fix();
        swap(pp, y->pp);
    }
    void splay() {
        for (pushFlip(); p; ) {
            if (p->p) p->p->pushFlip();
            p->pushFlip(); pushFlip();
            int c1 = up(), c2 = p->up();
            if (c2 == -1) p->rot(c1, 2);
            else p->p->rot(c2, c1 != c2);
        }
    }
    Node* first() {
        pushFlip();
        return c[0] ? c[0]->first() : (splay(), this);
    }
};

struct LinkCut {
    vector<Node> node;
    LinkCut(int N) : node(N) {}

    void link(int u, int v) { // add an edge (u, v)
        assert(!connected(u, v));
        makeRoot(&node[u]);
        node[u].pp = &node[v];
    }
    void cut(int u, int v) { // remove an edge (u, v)
        Node *x = &node[u], *top = &node[v];
        makeRoot(top); x->splay();
        assert(top == (x->pp ?: x->c[0]));
        if (x->pp) x->pp = 0;
        else {
            x->c[0] = top->p = 0;
            x->fix();
        }
    }
    bool connected(int u, int v) { // are u, v in the same tree?
        Node* nu = access(&node[u])->first();
        return nu == access(&node[v])->first();
    }
    void makeRoot(Node* u) {
        access(u);
        u->splay();
        if(u->c[0]) {
            u->c[0]->p = 0;
            u->c[0]->flip ^= 1;
            u->c[0]->pp = u;
```

```
            u->c[0] = 0;
            u->fix();
        }
    }
    Node* access(Node* u) {
        u->splay();
        while (Node* pp = u->pp) {
            pp->splay(); u->pp = 0;
            if (pp->c[1]) {
                pp->c[1]->p = 0; pp->c[1]->pp = pp; }
            pp->c[1] = u; pp->fix(); u = pp;
        }
        return u;
    }
};
```

Maximum Clique

Description: Quickly finds a maximum clique of a graph (given as symmetric bitset matrix; self-edges not allowed). Can be used to find a maximum independent set by finding a clique of the complement graph.
Time: Runs in about 1s for n=155 and worst case random graphs (p=.90). Runs faster for sparse graphs.

```
typedef vector<bitset<200>> vb;
struct Maxclique {
    double limit=0.025, pk=0;
    struct Vertex { int i, d=0; };
    typedef vector<Vertex> vv;
    vb e;
    vv V;
    vector<vi> C;
    vi qmax, q, S, old;
    void init(vv& r) {
        for (auto& v : r) v.d = 0;
        for (auto& v : r) for (auto j : r) v.d += e[v.i][j.i];
        sort(all(r), [](auto a, auto b) { return a.d > b.d; });
        int mxD = r[0].d;
        rep(i,0,sz(r)) r[i].d = min(i, mxD) + 1;
    }
    void expand(vv& R, int lev = 1) {
        S[lev] += S[lev - 1] - old[lev];
        old[lev] = S[lev - 1];
        while (sz(R)) {
            if (sz(q) + R.back().d <= sz(qmax)) return;
            q.push_back(R.back().i);
            vv T;
            for(auto v:R) if (e[R.back().i][v.i]) T.push_back({v.i}
                );
            if (sz(T)) {
                if (S[lev]++ / ++pk < limit) init(T);
                int j = 0, mxk = 1, mnk = max(sz(qmax) - sz(q) + 1,
                    1);
                C[1].clear(), C[2].clear();
                for (auto v : T) {
                    int k = 1;
                    auto f = [&](int i) { return e[v.i][i]; };
                    while (any_of(all(C[k]), f)) k++;
                    if (k > mxk) mxk = k, C[mxk + 1].clear();
                    if (k < mnk) T[j++].i = v.i;
                    C[k].push_back(v.i);
                }
                if (j > 0) T[j - 1].d = 0;
                rep(k,mnk,mxk + 1) for (int i : C[k])
                    T[j].i = i, T[j++].d = k;
                expand(T, lev + 1);
            } else if (sz(q) > sz(qmax)) qmax = q;
            q.pop_back(), R.pop_back();
        }
    }
    vi maxClique() { init(V), expand(V); return qmax; }
    Maxclique(vb conn) : e(conn), C(sz(e)+1), S(sz(C)), old(S) {
        rep(i,0,sz(e)) V.push_back({i});
    }
};
```

Miscellaneous (7)

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;
}
```