

# Contents

<b>1 Basic</b>	<b>1</b>
1.1 vimrc	1
1.2 Fast Integer Input	1
1.3 Big Integer	1
<b>2 Flow</b>	<b>3</b>
2.1 Dinic's Algorithm	3
2.2 Min cost Max flow	3
2.3 Maximum Bipartite Matching	3
2.4 Maximum Weighted Bipartite Matching	4
<b>3 Math</b>	<b>4</b>
3.1 FFT	4
3.2 NTT	4
3.3 Miller-Rabin	5
3.4 Pollard's Rho	5
3.5 $[\frac{n}{7}]$ Enumeration	5
3.6 $\mu$ function	5
3.7 Extended GCD	5
<b>4 Graph</b>	<b>5</b>
4.1 Strongly Connected Components	5
4.2 Heavy-Light Decomposition	6
4.3 Centroid Decomposition	6
4.4 Maximum Clique	6
4.5 Biconnected Component	7
4.6 2-Satisfiability	7
<b>5 Data Structures</b>	<b>7</b>
5.1 Dark Magic	7
5.2 Treap	7
5.3 Persistent Disjoint Set	8
5.4 Leftlist Tree	8
<b>6 Geometry</b>	<b>8</b>
6.1 Points	8
6.2 Segment Intersection	8
6.3 Line Intersection	9
6.4 Circle Intersection	9
6.5 Convex Hull	9
6.6 Rotating Caliper	9
6.7 Closest Pair	9
6.8 Circle	9
6.9 Minimum Enclosing Circle	9
6.10 Triangle Center	10
<b>7 String</b>	<b>10</b>
7.1 KMP	10
7.2 Suffix Array	10
7.3 Z Algorithm	10
7.4 Manacher's Algorithm	10
7.5 Smallest Rotation	11
7.6 Rolling Hash Primes	11
<b>8 Dynamic Programming</b>	<b>11</b>
8.1 Concept	11
8.2 Linear Convex Hull Optimization	11
8.3 Convex Hull Optimization	11
8.4 Convex Montone Optimization	11
8.5 Divide and Conquer Optimization	12

## 1 Basic

### 1.1 vimrc

```
syn on
se ai nu ru mouse=a
se cin et ts=4 sw=4 sts=4
so $VIMRUNTIME/mswin.vim
colo desert
se gfn=Monospace\ 15
execute pathogen#infect()
```

### 1.2 Fast Integer Input

```
#define getchar gtx

inline int gtx() {
    const int N = 1048576;
    static char buffer[N];
    static char *p = buffer, *end = buffer;
    if (p == end) {
        if ((end = buffer + fread(buffer, 1, N, stdin)) ==
            buffer) return EOF;
        p = buffer;
    }
    return *p++;
}

template <typename T>
inline bool rit(T& x) {
    char __c = 0; bool flag = false;
    while (__c = getchar(), (__c < '0' && __c != '-') ||
        __c > '9') if (__c == -1) return false;
    __c == '-' ? (flag = true, x = 0) : (x = __c - '0');
    while (__c = getchar(), __c >= '0' && __c <= '9') x =
        x * 10 + __c - '0';
    if (flag) x = -x;
    return true;
}

template <typename T, typename ...Args>
inline bool rit(T& x, Args& ...args) { return rit(x) &&
    rit(args...); }
```

### 1.3 Big Integer

```
#include <bits/stdc++.h>

struct Int {
    static const int inf = 1e9;
    std::vector<int> dig;
    bool sgn;
    Int() {
        dig.push_back(0);
        sgn = true;
    }
    Int(int n) {
        sgn = n >= 0;
        while (n) {
            dig.push_back(n % 10);
            n /= 10;
        }
        if (dig.size() == 0) dig.push_back(0);
    }
    Int(std::string s) {
        int i = 0; sgn = true;
        if (s[i] == '-') sgn = false, ++i;
        for (; i < s.length(); ++i) dig.push_back(s[i] - '0');
        reverse(dig.begin(), dig.end());
        if (dig.size() == 1 && dig[0] == '0') sgn = true;
    }
    Int(const std::vector<int>& d, const bool& s = true) {
        dig = std::vector<int>(d.begin(), d.end());
        sgn = s;
    }
};
```

```

}
Int(const Int& n) {
    sgn = n.sgn;
    dig = n.dig;
}
bool operator<(const Int& rhs) const {
    if (sgn && !rhs.sgn) return true;
    if (!sgn && rhs.sgn) return false;
    if (!sgn && !rhs.sgn) return Int(dig) > Int(rhs.dig);
    if (dig.size() < rhs.dig.size()) return true;
    if (dig.size() > rhs.dig.size()) return false;
    for (int i = dig.size() - 1; i >= 0; --i) {
        if (dig[i] != rhs.dig[i]) return dig[i] < rhs.dig[i];
    }
    return false;
}
bool operator==(const Int& rhs) const {
    if (sgn != rhs.sgn) return false;
    return dig == rhs.dig;
}
bool operator>(const Int& rhs) const {
    return !(*this < rhs) && !(*this == rhs);
}
bool operator<(const int& n) const {
    return *this < Int(n);
}
bool operator>(const int& n) const {
    return *this > Int(n);
}
bool operator==(const int& n) const {
    return *this == Int(n);
}
Int operator-() const {
    return Int(dig, !sgn);
}
Int operator+(const Int& rhs) const {
    bool res = true;
    if (!sgn && !rhs.sgn) res = false;
    else if (!sgn && rhs.sgn) return rhs - (*this);
    else if (sgn && !rhs.sgn) return *this - -rhs;
    std::vector<int> v1 = dig, v2 = rhs.dig;
    if (v2.size() > v1.size()) swap(v1, v2);
    int car = 0;
    std::vector<int> nvec;
    for (int i = 0; i < v2.size(); ++i) {
        int k = v1[i] + v2[i] + car;
        nvec.push_back(k % 10);
        car = k / 10;
    }
    for (int i = v2.size(); i < v1.size(); ++i) {
        int k = v1[i] + car;
        nvec.push_back(k % 10);
        car = k / 10;
    }
    return Int(nvec, res);
}
Int operator-(const Int& rhs) const {
    if (*this < rhs) {
        std::vector<int> nvec = (rhs - *this).dig;
        return Int(nvec, false);
    }
    if (*this == rhs) return Int(0);
    std::vector<int> v1 = dig, v2 = rhs.dig;
    std::vector<int> nvec;
    for (int i = 0; i < v2.size(); ++i) {
        int k = v1[i] - v2[i];
        if (k < 0) {
            for (int j = i + 1; j < v1.size(); ++j) if (v1[j] > 0) {
                --v1[j]; k += 10;
                break;
            }
        }
        nvec.push_back(k);
    }
    int rind = v1.size() - 1;
    while (rind >= v2.size() && v1[rind] == 0) --rind;
    for (int i = v2.size(); i <= rind; ++i) {
        nvec.push_back(v1[i]);
    }
}

```

```

return Int(nvec);
}
Int operator*(const Int& rhs) const {
    if (sgn && !rhs.sgn || !sgn && rhs.sgn) return -(Int(dig, true) * Int(rhs.dig, true));
    if (*this == 0) return Int();
    if (rhs == 0) return Int();
    std::vector<int> v1 = dig, v2 = rhs.dig;
    if (v1.size() < v2.size()) swap(v1, v2);
    std::vector<int> res(v1.size() * v2.size(), 0);
    for (int i = 0; i < v2.size(); ++i) {
        int car = 0;
        for (int j = 0; j < v1.size(); ++j) {
            int k = car + v1[j] * v2[i];
            res[j + i] += k % 10;
            car = k / 10;
        }
    }
    int car = 0;
    for (int i = 0; i < res.size(); ++i) {
        int k = car + res[i];
        res[i] = k % 10;
        car = k / 10;
    }
    while (car) {
        res.push_back(car % 10);
        car /= 10;
    }
    int ind = res.size() - 1;
    while (ind >= 0 && res[ind] == 0) --ind;
    std::vector<int> nvec;
    for (int i = 0; i <= ind; ++i) nvec.push_back(res[i]);
    return Int(nvec);
}
Int operator+(const int& n) const {
    return *this + Int(n);
}
Int operator-(const int& n) const {
    return *this - Int(n);
}
Int& operator+=(const Int& n) {
    *this = (*this + n);
    return *this;
}
Int& operator-=(const Int& n) {
    *this = (*this - n);
    return *this;
}
Int& operator+=(const int& n) {
    *this += Int(n);
    return *this;
}
Int& operator-=(const int& n) {
    *this -= Int(n);
    return *this;
}
Int& operator*=(const Int& n) {
    *this = *this * n;
    return *this;
}
Int& operator*=(const int& n) {
    *this *= Int(n);
    return *this;
}
Int& operator++(int) {
    *this += 1;
    return *this;
}
Int& operator--(int) {
    *this -= 1;
    return *this;
}
friend std::istream& operator>>(std::istream& in, Int& n) {
    std::string s; in >> s;
    n = Int(s);
    return in;
}
friend std::ostream& operator<<(std::ostream& out, const Int& n) {
    if (!n.sgn) out << "-";
}

```

```

    for (int i = n.dig.size() - 1; i >= 0; --i) out <<
        n.dig[i];
    return out;
}
};

```

## 2 Flow

### 2.1 Dinic's Algorithm

```

struct Dinic {
    int n, s, t;
    vector<int> level;
    struct Edge {
        int to, rev, cap;
        Edge() {}
        Edge(int a, int b, int c): to(a), cap(b), rev(c) {}
    };
    vector<Edge> G[maxn];
    bool bfs() {
        level.assign(n, -1);
        level[s] = 0;
        queue<int> que; que.push(s);
        while (que.size()) {
            int tmp = que.front(); que.pop();
            for (auto e : G[tmp]) {
                if (e.cap > 0 && level[e.to] == -1) {
                    level[e.to] = level[tmp] + 1;
                    que.push(e.to);
                }
            }
        }
        return level[t] != -1;
    }
    int flow(int now, int low) {
        if (now == t) return low;
        int ret = 0;
        for (auto &e : G[now]) {
            if (e.cap > 0 && level[e.to] == level[now] + 1) {
                int tmp = flow(e.to, min(e.cap, low - ret));
                e.cap -= tmp; G[e.to][e.rev].cap += tmp;
                ret += tmp;
            }
        }
        if (ret == 0) level[now] = -1;
        return ret;
    }
    Dinic(int _n, int _s, int _t): n(_n), s(_s), t(_t) {
        fill(G, G + maxn, vector<Edge>());
    }
    void add_edge(int a, int b, int c) {
        G[a].push_back(Edge(b, c, G[b].size()));
        G[b].push_back(Edge(a, 0, G[a].size() - 1));
    }
    int maxflow() {
        int ret = 0;
        while (bfs()) ret += flow(s, inf);
        return ret;
    }
};

```

### 2.2 Min cost Max flow

```

struct MCMF {
    struct Edge {
        int to, rev, cap, w;
        Edge() {}
        Edge(int a, int b, int c, int d): to(a), cap(b), w(
            c), rev(d) {}
    };
    int n, s, t;
    vector<int> p, id, d;
    bitset<maxn> inque;
    vector<Edge> G[maxn];
    pair<int, int> spfa() {
        p.assign(n, -1);

```

```

        d.assign(n, inf);
        id.assign(n, -1);
        d[s] = 0; p[s] = s;
        queue<int> que; que.push(s); inque[s] = true;
        while (que.size()) {
            int tmp = que.front(); que.pop();
            inque[tmp] = false;
            int i = 0;
            for (auto e : G[tmp]) {
                if (e.cap > 0 && d[e.to] > d[tmp] + e.w) {
                    d[e.to] = d[tmp] + e.w;
                    p[e.to] = tmp;
                    id[e.to] = i;
                    if (!inque[e.to]) que.push(e.to), inque[e.to]
                        = true;
                }
                ++i;
            }
        }
        if (d[t] == inf) return make_pair(-1, -1);
        int a = inf;
        for (int i = t; i != s; i = p[i]) {
            a = min(a, G[p[i]][id[i]].cap);
        }
        for (int i = t; i != s; i = p[i]) {
            Edge &e = G[p[i]][id[i]];
            e.cap -= a; G[e.to][e.rev].cap += a;
        }
        return make_pair(a, d[t]);
    }
    MCMF(int _n, int _s, int _t): n(_n), s(_s), t(_t) {
        fill(G, G + maxn, vector<Edge>());
    }
    void add_edge(int a, int b, int cap, int w) {
        G[a].push_back(Edge(b, cap, w, (int)G[b].size()));
        G[b].push_back(Edge(a, 0, -w, (int)G[a].size() - 1));
    }
    pair<int, int> maxflow() {
        int mxf = 0, mnc = 0;
        while (true) {
            pair<int, int> res = spfa();
            if (res.first == -1) break;
            mxf += res.first; mnc += res.first * res.second;
        }
        return make_pair(mxf, mnc);
    }
};

```

### 2.3 Maximum Bipartite Matching

```

struct MaximumMatching {
    vector<int> G[maxn], mt;
    int n;
    bitset<maxn> v;
    MaximumMatching(int n): n(n) {
        fill(G, G + maxn, vector<int>());
        v.reset();
    }
    void add_edge(int a, int b) {
        G[a].push_back(b);
    }
    bool dfs(int now) {
        v[now] = true;
        for (int u : G[now]) {
            if (mt[u] == -1 || !v[mt[u]] && dfs(mt[u])) {
                mt[u] = now;
                return true;
            }
        }
        return false;
    }
    int solve() {
        mt.assign(n, -1);
        int ret = 0;
        for (int i = 0; i < n; ++i) {
            memset(v, false, sizeof(v));
            if (dfs(i)) ++ret;
        }
        return ret;
    }
};

```

```

}
};

```

## 2.4 Maximum Weighted Bipartite Matching

```

struct Hungarian {
    vector<vector<int>> w;
    bitset<maxn> s, t;
    vector<int> lx, ly, mx, my, slack, prv;
    int n, matched;
    Hungarian() {}
    Hungarian(int _n): n(_n) {
        w = vector<vector<int>>(n, vector<int>(n));
        lx.resize(n); ly.resize(n); mx.assign(n, -1); my.
        assign(n, -1);
        slack.resize(n); prv.resize(n);
    }
    void add_edge(int a, int b, int c) {
        w[a][b] = c;
    }
    void add(int x) {
        s[x] = true;
        for (int i = 0; i < n; ++i) {
            if (lx[x] + ly[i] - w[x][i] < slack[i]) {
                slack[i] = lx[x] + ly[i] - w[x][i];
                prv[i] = x;
            }
        }
    }
    void augment(int now) {
        int x = prv[now], y = now;
        ++matched;
        while (true) {
            int tmp = mx[x]; mx[x] = y; my[y] = x; y = tmp;
            if (y == -1) return;
            x = prv[y];
        }
    }
    void relabel() {
        int delta = inf;
        for (int i = 0; i < n; ++i) if (!t[i]) delta = min(
            delta, slack[i]);
        for (int i = 0; i < n; ++i) if (s[i]) lx[i] -=
            delta;
        for (int i = 0; i < n; ++i) {
            if (t[i]) ly[i] += delta;
            else slack[i] -= delta;
        }
    }
    void go() {
        s.reset(); t.reset();
        fill(slack.begin(), slack.end(), inf);
        int root = 0;
        for (; root < n && mx[root] != -1; ++root);
        add(root);
        while (true) {
            relabel();
            int y = 0;
            for (; y < n; ++y) if (!t[y] && slack[y] == 0)
                break;
            if (my[y] == -1) return augment(y), void();
            add(my[y]); t[y] = true;
        }
    }
    int matching() {
        int ret = 0;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) lx[i] = max(lx[i], w[
                i][j]);
        }
        for (int i = 0; i < n; ++i) go();
        for (int i = 0; i < n; ++i) ret += w[i][mx[i]];
        return ret;
    }
};

```

## 3 Math

### 3.1 FFT

```

const double pi = acos(-1);
const complex<double> I(0, 1);
complex<double> omega[maxn + 1];

void prefft() {
    for (int i = 0; i <= maxn; ++i) omega[i] = exp(i * 2
        * pi / maxn * I);
}

void fft(vector<complex<double>>& a, int n, bool inv=
    false) {
    int basic = maxn / n;
    int theta = basic;
    for (int m = n; m >= 2; m >= 1) {
        int h = m >> 1;
        for (int i = 0; i < h; ++i) {
            complex<double> w = omega[inv ? maxn - (i * theta
                % maxn) : i * theta % maxn];
            for (int j = i; j < n; j += m) {
                int k = j + h;
                complex<double> x = a[j] - a[k];
                a[j] += a[k];
                a[k] = w * x;
            }
        }
        theta = (theta * 2) % maxn;
    }
    int i = 0;
    for (int j = 1; j < n - 1; ++j) {
        for (int k = n >> 1; k > (i ^ k); k >= 1);
        if (j < i) swap(a[i], a[j]);
    }
    if (inv) for (int i = 0; i < n; ++i) a[i] /= (double)
        n;
}

void invfft(vector<complex<double>>& a, int n) {
    fft(a, n, true);
}

```

### 3.2 NTT

```

const long long p = 2013265921, root = 31;
long long omega[maxn + 1];

long long fpow(long long a, long long n) {
    long long ret = 1ll;
    for (; n; n >>= 1) {
        if (n & 1) ret = ret * a % p;
        a = a * a % p;
    }
    return ret;
}

void prentt() {
    omega[0] = 1;
    long long r = fpow(root, (p - 1) / maxn);
    for (int i = 1; i <= maxn; ++i) omega[i] = omega[i -
        1] * r % p;
}

void ntt(vector<long long>& a, int n, bool inv = false)
{
    int basic = maxn / n;
    int theta = basic;
    for (int m = n; m >= 2; m >= 1) {
        int mh = m >> 1;
        for (int i = 0; i < mh; ++i) {
            long long w = omega[i * theta % maxn];
            for (int j = i; j < n; j += m) {
                int k = j + mh;
                long long x = a[j] - a[k];
                if (x < 0) x += p;
                a[j] += a[k];
                if (a[j] > p) a[j] -= p;
                a[k] = w * x % p;
            }
        }
    }
}

```

```

    }
    }
    theta = theta * 2 % maxn;
}
int i = 0;
for (int j = 1; j < n - 1; ++j) {
    for (int k = n >> 1; k > (i ^ k); k >>= 1);
    if (j < i) swap(a[i], a[j]);
}
if (!inv) return;
long long ni = fpow(n, p - 2);
reverse(a.begin() + 1, a.end());
for (int i = 0; i < n; ++i) a[i] = a[i] * ni % p;
}

```

### 3.3 Miller-Rabin

```

// n < 4759123141  chk = [2, 7, 61]
// n < 1122004669633  chk = [2, 13, 23, 1662803]
// n < 2^64  chk = [2, 325, 9375, 28178, 450775,
// 9780504, 1795265022]

long long fpow(long long a, long long n, long long mod)
{
    long long ret = 1LL;
    for (; n; n >>= 1) {
        if (n & 1) ret = (__int128)ret * (__int128)a % mod;
        a = (__int128)a * (__int128)a % mod;
    }
    return ret;
}
bool check(long long a, long long u, long long n, int t)
{
    a = fpow(a, u, n);
    if (a == 0) return true;
    if (a == 1 || a == n - 1) return true;
    for (int i = 0; i < t; ++i) {
        a = (__int128)a * (__int128)a % n;
        if (a == 1) return false;
        if (a == n - 1) return true;
    }
    return false;
}
bool is_prime(long long n) {
    if (n < 2) return false;
    if (n % 2 == 0) return n == 2;
    long long u = n - 1; int t = 0;
    for (; u & 1; u >>= 1, ++t);
    for (long long i : chk) {
        if (!check(i, u, n, t)) return false;
    }
    return true;
}

```

### 3.4 Pollard's Rho

```

long long f(long long x, long long mod) {
    return add(mul(x, x, mod), 1, mod);
}

long long pollard_rho(long long n) {
    if (n % 2 == 0) return 2;
    while (true) {
        long long y = 2, x = rand() % (n - 1) + 1, res = 1;
        for (int sz = 2; res == 1; sz <= 1) {
            for (int i = 0; i < sz && res == 1; ++i) {
                x = f(x, n);
                res = __gcd(abs(x - y), n);
            }
            y = x;
        }
        if (res && res != n) return res;
    }
}

```

### 3.5 $\lfloor \frac{n}{i} \rfloor$ Enumeration

```

vector<int> solve(int n) {
    vector<int> vec;
    for (int t = 1; t < n; t = (n / (n / (t + 1)))) vec.
        push_back(t);
    vec.push_back(n);
    vec.resize(unique(vec.begin(), vec.end()) - vec.begin()
        ());
    return vec;
}

```

### 3.6 $\mu$ function

```

int mu[maxn], pi[maxn];
vector<int> prime;

void sieve() {
    mu[1] = pi[1] = 1;
    for (int i = 2; i < maxn; ++i) {
        if (!pi[i]) {
            pi[i] = i;
            prime.push_back(i);
            mu[i] = -1;
        }
        for (int j = 0; i * prime[j] < maxn; ++j) {
            pi[i * prime[j]] = prime[j];
            mu[i * prime[j]] = -mu[i];
            if (i % prime[j] == 0) {
                mu[i * prime[j]] = 0;
                break;
            }
        }
    }
}

```

### 3.7 Extended GCD

```

template <typename T> tuple<T, T, T> extgcd(T a, T b) {
    if (!b) return make_tuple(a, 1, 0);
    T d, x, y;
    tie(d, x, y) = extgcd(b, a % b);
    return make_tuple(d, y, x - (a / b) * y);
}

```

## 4 Graph

### 4.1 Strongly Connected Components

```

struct SCC {
    vector<int> G[maxn], R[maxn], topo;
    int n, nscc;
    vector<int> scc, sz;
    bitset<maxn> v;
    void dfs(int now) {
        v[now] = true;
        for (int u : G[now]) if (!v[u]) {
            dfs(u);
        }
        topo.push_back(now);
    }
    void rdfs(int now) {
        v[now] = true;
        scc[now] = nscc;
        ++sz[nscc];
        for (int u : R[now]) if (!v[u]) {
            rdfs(u);
        }
    }
    SCC(): {}
    SCC(int n): n(n) {
        scc.assign(n, 0); sz.assign(n, 0);
    }
    void add_edge(int a, int b) {

```

```

    G[a].push_back(b);
    R[b].push_back(a);
}
void solve() {
    v.reset();
    for (int i = 0; i < n; ++i) if (!v[i]) dfs(i);
    reverse(topo.begin(), topo.end());
    v.reset();
    for (int i : topo) if (!v[i]) {
        ++nsc;
        rdfs(i);
    }
}
};

```

## 4.2 Heavy-Light Decomposition

```

struct HeavyLightDecomp {
    vector<int> G[maxn];
    vector<int> tin, top, dep, maxson, sz, p;
    int n, t;
    void dfs(int now, int fa, int d) {
        dep[now] = d;
        maxson[now] = -1;
        sz[now] = 1;
        p[now] = fa;
        for (int u : G[now]) if (u != fa) {
            dfs(u, now, d + 1);
            sz[now] += sz[u];
            if (maxson[now] == -1 || sz[u] > sz[maxson[now]])
                maxson[now] = u;
        }
    }
    void link(int now, int tp) {
        top[now] = tp;
        tin[now] = ++t;
        if (maxson[now] == -1) return;
        link(maxson[now], tp);
        for (int u : G[now]) if (u != p[now]) {
            if (u == maxson[now]) continue;
            link(u, u);
        }
    }
    HeavyLightDecomp(int n): n(n) {
        t = 0;
        tin.assign(n, 0); top.assign(n, 0); dep.assign(n, 0);
        maxson.assign(n, 0); sz.assign(n, 0); p.assign(n, 0);
    }
    void add_edge(int a, int b) {
        G[a].push_back(b);
        G[b].push_back(a);
    }
    void build() {
        dfs(0, -1, 0);
        link(0, 0);
    }
    int lca(int a, int b) {
        int ta = top[a], tb = top[b];
        while (ta != tb) {
            if (dep[ta] < dep[tb]) {
                swap(ta, tb); swap(a, b);
            }
            a = p[ta]; ta = top[a];
        }
        if (a == b) return a;
        return dep[a] < dep[b] ? a : b;
    }
    vector<pair<int, int>> get_path(int a, int b) {
        int ta = top[a], tb = top[b];
        vector<pair<int, int>> ret;
        while (ta != tb) {
            if (dep[ta] < dep[tb]) {
                swap(ta, tb); swap(a, b);
            }
            ret.push_back(make_pair(tin[ta], tin[a]));
            a = p[ta]; ta = top[a];
        }
    }
}

```

```

        ret.push_back(make_pair(min(tin[a], tin[b]), max(
            tin[a], tin[b])));
        return ret;
    }
};

```

## 4.3 Centroid Decomposition

```

vector<pair<int, int>> G[maxn];
int sz[maxn], mx[maxn];
bool v[maxn];
vector<int> vtx;

void get_center(int now) {
    v[now] = true; vtx.push_back(now);
    sz[now] = 1; mx[now] = 0;
    for (int u : G[now]) if (!v[u]) {
        get_center(u);
        mx[now] = max(mx[now], sz[u]);
        sz[now] += sz[u];
    }
}

void get_dis(int now, int d, int len) {
    dis[d][now] = cnt;
    v[now] = true;
    for (auto u : G[now]) if (!v[u.first]) {
        get_dis(u, d, len + u.second);
    }
}

void dfs(int now, int fa, int d) {
    get_center(now);
    int c = -1;
    for (int i : vtx) {
        if (max(mx[i], (int)vtx.size() - sz[i]) <= (int)vtx
            .size() / 2) c = i;
        v[i] = false;
    }
    get_dis(c, d, 0);
    for (int i : vtx) v[i] = false;
    v[c] = true; vtx.clear();
    dep[c] = d; p[c] = fa;
    for (auto u : G[c]) if (u.first != fa && !v[u.first]) {
        dfs(u.first, c, d + 1);
    }
}

```

## 4.4 Maximum Clique

```

struct MaxClique {
    static const int maxv = 64;
    unsigned long long adj[maxv];
    int n, ans;
    void init(int _n): n(_n) {
        for (int i = 0; i < n; ++i) adj[i] = 0ull;
    }
    void add_edge(int a, int b) {
        adj[a] |= (1ull << a);
        adj[b] |= (1ull << b);
    }
    void go(unsigned long long r, unsigned long long p,
        unsigned long long x, int c, int res) {
        if (c + res < ans) return;
        if (p == 0ull && x == 0ull) return ans = max(ans, c
        ), void();
        unsigned long long y = p | x; y &= ~y;
        unsigned long long q = p & (~adj[(int)log2(y)]);
        while (q) {
            int i = (int)log2(q & -q);
            go(r | (1ull << i), p & adj[i], x & adj[i], cnt +
            1, __builtin_popcountll(p & adj[i]));
            q &= ~(1ull << i);
            p &= ~(1ull << i);
            x |= (1ull << i);
        }
    }
}

```



```

int maxclique() {
    ans = 0;
    unsigned long long s = 0;
    if (n < maxv) s = (1ull << n) - 1;
    else for (int i = 0; i < n; ++i) s |= (1ull << i);
    go(0ull, s, 0ull, 0ull, n);
    return ans;
}
}

```

## 4.5 Biconnected Component

```

int tin[maxn], low[maxn], t, bccsz;
stack<int> st;
vector<int> bcc[maxn];

void dfs(int now, int fa) {
    tin[now] = ++t; low[now] = tin[now];
    st.push(now);
    for (int u : G[now]) if (u != fa) {
        if (!tin[u]) {
            dfs(u, now);
            low[now] = min(low[now], low[u]);
            if (low[u] >= tin[now]) {
                int v;
                ++bccsz;
                do {
                    v = st.top(); st.pop();
                    bcc[bccsz].push_back(v);
                } while (v != u);
                bcc[bccsz].push_back(now);
            }
        } else {
            low[now] = min(low[now], tin[u]);
        }
    }
}
}

```

## 4.6 2-Satisfiability

```

struct TwoSat {
    vector<int> G[maxn << 1];
    bitset<maxn << 1> v;
    vector<int> s;
    int c;
    bool dfs(int now) {
        if (v[now ^ 1]) return false;
        if (v[now]) return true;
        v[now] = true;
        s[c++] = now;
        for (int u : G[now]) if (!dfs(u)) return false;
        return true;
    }
    TwoSat() {
        s.assign(maxn << 1, 0);
        v.reset();
    }
    void add_edge(int a, int b) {
        G[a].push_back(b);
    }
    bool solve() {
        for (int i = 0; i < maxn << 1; i += 2) {
            if (!v[i] && !v[i + 1]) {
                c = 0;
                if (!dfs(i)) {
                    while (c) v[s[--c]] = false;
                    if (!dfs(i + 1)) return false;
                }
            }
        }
        return true;
    }
}

```

## 5 Data Structures

### 5.1 Dark Magic

```

#include <bits/stdc++.h>
#include <bits/extc++.h>
#include <ext/rope>
using namespace __gnu_pbds;
using namespace __gnu_cxx;
#include <ext/pb_ds/assoc_container.hpp>
typedef tree<int, null_type, std::less<int>,
            rb_tree_tag, tree_order_statistics_node_update>
            tree_set;
typedef cc_hash_table<int, int> umap;
typedef priority_queue<int> heap;

int main() {
    // rb tree
    tree_set s;
    s.insert(71); s.insert(22);
    assert(*s.find_by_order(0) == 22); assert(*s.
        find_by_order(1) == 71);
    assert(s.order_of_key(22) == 0); assert(s.
        order_of_key(71) == 1);
    s.erase(22);
    assert(*s.find_by_order(0) == 71); assert(s.
        order_of_key(71) == 0);
    // mergable heap
    heap a, b; a.join(b);
    // persistant
    rope<char> r[2];
    r[1] = r[0];
    std::string st = "abc";
    r[1].insert(0, st.c_str());
    r[1].erase(1, 1);
    std::cout << r[1].substr(0, 2) << std::endl;
    return 0;
}

```

### 5.2 Treap

```

#define tsize(t) (t ? t->size : 0)

struct treap {
    treap *lc, *rc;
    int val, pri, sz;
    treap(int v): val(v) {
        lc = rc = nullptr;
        pri = rand();
        sz = 1;
    }
    void pull() {
        sz = tsize(lc) + tsize(rc) + 1;
    }
};

treap *merge(treap *a, treap *b) {
    if (!a || !b) return a ? a : b;
    if (a->pri > b->pri) {
        a->rc = merge(a->rc, b);
        a->pull();
        return a;
    } else {
        b->lc = merge(a, b->lc);
        b->pull();
        return b;
    }
}

void split_by_val(treap *t, int k, treap *&a, treap *&b) {
    if (!t) {
        a = b = nullptr;
        return;
    }
    if (t->val <= k) {
        a = t;
        split(t->rc, k, a->rc, b);
    }
}

```

```

    a->pull();
} else {
    b = t;
    split(t->lc, k, a, b->lc);
    b->pull();
}
}

void split_by_size(treap *t, int k, treap *&a, treap *&b) {
    if (!t) {
        a = b = nullptr;
        return;
    }
    if (tsize(t->lc) + 1 <= k) {
        a = t;
        split(t->rc, k - tsize(t->lc) - 1, a->rc, b);
        a->pull();
    } else {
        b = t;
        split(t->lc, k, a, b->lc);
        b->pull();
    }
}
}

```

### 5.3 Persistent Disjoint Set

```

struct DisjointSet {
    int p[maxn], sz[maxn], n;
    vector<pair<int*, int*>> h;
    vector<int> sp;
    void init(int size) {
        n = size;
        for (int i = 0; i < n; ++i) p[i] = i, sz[i] = 1;
        sp.clear(); h.clear();
    }
    void assign(int *k, int v) {
        h.push_back(make_pair(k, *k));
        *k = v;
    }
    void save() {
        sp.push_back(h.size());
    }
    void undo() {
        int last = sp.back(); sp.pop_back();
        while (h.size() != last) {
            pair<int*, int> pi = h.back(); h.pop_back();
            *pi.first = pi.second;
        }
    }
    int find(int x) {
        if (x == p[x]) return x;
        return p[x] = find(p[x]);
    }
    void merge(int x, int y) {
        x = find(x); y = find(y);
        if (x == y) return;
        if (sz[x] < sz[y]) swap(x, y);
        assign(&sz[x], sz[x] + sz[y]);
        assign(&p[y], x);
    }
};

```

### 5.4 Leftlist Tree

```

namespace LeftlistTree {
    struct Node {
        T val;
        int s;
        Node *lc, *rc;
        Node(T v): val(v) {
            lc = rc = nullptr;
            s = 1;
        }
    };
    inline int rank(Node* t) {
        return t ? t->s : 0;
    }
}

```

```

Node *merge(Node *a, Node *b) {
    if (!a || !b) return a ? a : b;
    if (a->val < b->val) swap(a, b);
    a->rc = merge(a->rc, b);
    if (rank(a->lc) < rank(a->rc)) swap(a->lc, a->rc);
    a->s = rank(a->rc) + 1;
    return a;
}

void clear(Node *t) {
    if (!t) return;
    if (t->lc) clear(t->lc);
    if (t->rc) clear(t->rc);
    delete t;
}
}

```

## 6 Geometry

### 6.1 Points

```

using type = double;

struct pt {
    type x, y;
    pt(): x(0), y(0) {}
    pt(type x, type y): x(x), y(y) {}
    pt operator+(const pt& a) const { return pt(x + a.x,
        y + a.y); }
    pt operator-(const pt& a) const { return pt(x - a.x,
        y - a.y); }
    pt operator*(const type& t) const { return pt(x * t,
        y * t); }
    pt operator/(const type& t) const { return pt(x / t,
        y / t); }
    type operator*(const pt& a) const { return x * a.x +
        y * a.y; }
    type operator^(const pt& a) const { return x * a.y -
        y * a.x; }
    bool operator<(const pt& a) const { return x == a.x ?
        y < a.y : x < a.x; }
    bool operator>(const pt& a) const { return x == a.x ?
        y > a.y : x > a.x; }
    bool operator==(const pt& a) const { return x == a.x
        && y == a.y; }
    bool operator!=(const pt& a) const { return x != a.x
        || y != a.y; }
    double dist(const pt &p) const { return sqrt((p - *
        this) * (p - *this)); }
    friend double norm2(const pt &p) { return p * p; }
    friend double norm(const pt &p) { return sqrt(norm2(p
        )); }
};

```

### 6.2 Segment Intersection

```

type orient(const pt& o, const pt& a, const pt& b) {
    type t = (a - o) ^ (b - o);
    return t / max(1, abs(t));
}

// if parallel and min(p1, p2) != max(q1, q2) and p1 !=
// p2 and q1 != q2 ==> segment
// else ==> point

bool intersect(const pt& p1, const pt& p2, const pt& q1,
    const pt& q2) {
    if (q1 == q2 && p1 == p2) return p1 == q1;
    if (q1 == q2) return ((p2 - q1) ^ (p1 - q1)) == 0 &&
        q1.x >= min(p1.x, p2.x) && q1.x <= max(p1.x, p2.x);
    if (p1 == p2) return ((q1 - p1) ^ (q2 - p1)) == 0 &&
        p1.x >= min(q1.x, q2.x) && p1.x <= max(q1.x, q2.x);
    int c = (p2 - p1) ^ (q2 - q1);
    if (c == 0) {
        if (orient(p1, p2, q1)) return false;
        return (p1 - q1) * (p2 - q1) <= 0 || (p1 - q2) * (
            p2 - q2) <= 0 || (q1 - p1) * (q2 - p1) <= 0 || (q1
            - p2) * (q2 - p2) <= 0;
    }
}

```



```

    }
    return (orient(p1, p2, q1) * orient(p1, p2, q2) <= 0)
        && (orient(q1, q2, p1) * orient(q1, q2, p2) <= 0);
}

```

### 6.3 Line Intersection

```

pt intersection(const pt& p1, const pt& p2, const pt&
    q1, const pt& q2) {
    double f1 = (p2 - p1) ^ (q1 - p1);
    double f2 = (p2 - p1) ^ (p1 - q2);
    double f = f1 + f2;
    if (fabs(f) < eps) return pt(nan(""), nan(""));
    return q1 * (f2 / f) + q2 * (f1 / f);
}

```

### 6.4 Circle Intersection

```

vector<pt> circle_intersection(const pt& o1, const
    double& r1, const pt& o2, const double& r2) {
    double d2 = (o1 - o2) * (o1 - o2);
    double d = sqrt(d2);
    if (d > r1 + r2) return {};
    pt u = (o1 + o2) * 0.5 + (o1 - o2) * ((r2 * r2 - r1 *
        r1) / (2 * d2));
    double a = sqrt((r1 + r2 + d) * (r1 - r2 + d) * (r1 +
        r2 - d) * (-r1 + r2 + d));
    pt v = pt(o1.y - o2.y, -o1.x + o2.x) * a / (2 * d2);
    return { u + v, u - v };
}

```

### 6.5 Convex Hull

```

type cross(const pt& o, const pt& a, const pt& b) {
    return (a - o) ^ (b - o);
}

int rsd;

vector<pt> convex_hull(vector<pt> p) {
    sort(p.begin(), p.end());
    int m = 0;
    vector<pt> ret(2 * p.size());
    for (int i = 0; i < p.size(); ++i) {
        while (m >= 2 && cross(ret[m - 2], ret[m - 1], p[i])
            < 0) --m;
        ret[m++] = p[i];
    }
    rsd = m - 1;
    for (int i = p.size() - 2, t = m + 1; i >= 0; --i) {
        while (m >= t && cross(ret[m - 2], ret[m - 1], p[i])
            < 0) --m;
        ret[m++] = p[i];
    }
    ret.resize(m - 1);
    return ret;
}

```

### 6.6 Rotating Caliper

```

void rotating_caliper(vector<pt> p) {
    vector<pt> ch = convex_hull(p);
    int tbz = ch.size();
    int lpr = 0, rpr = rsd;
    // ch[lpr], ch[rpr]
    while (lpr < rsd || rpr < tbz - 1) {
        if (lpr < rsd && rpr < tbz - 1) {
            pt rvt = ch[rpr + 1] - ch[rpr];
            pt lvt = ch[lpr + 1] - ch[lpr];
            if ((lvt ^ rvt) < 0) ++lpr;
            else ++rpr;
        }
        else if (lpr == rsd) ++rpr;
    }
}

```

```

    else ++lpr;
    // ch[lpr], ch[rpr]
}
}

```

### 6.7 Closest Pair

```

pt p[maxn];

double dis(const pt& a, const pt& b) {
    return sqrt((a - b) * (a - b));
}

double closest_pair(int l, int r) {
    if (l == r) return inf;
    if (r - l == 1) return dis(p[l], p[r]);
    int m = (l + r) >> 1;
    double d = min(closest_pair(l, m), closest_pair(m +
        1, r));
    vector<int> vec;
    for (int i = m; i >= l && fabs(p[m].x - p[i].x) < d;
        --i) vec.push_back(i);
    for (int i = m + 1; i <= r && fabs(p[m].x - p[i].x) <
        d; ++i) vec.push_back(i);
    sort(vec.begin(), vec.end(), [=](const int& a, const
        int& b) { return p[a].y < p[b].y; });
    for (int i = 0; i < vec.size(); ++i) {
        for (int j = i + 1; j < vec.size() && fabs(p[vec[j]
            ].y - p[vec[i]].y) < d; ++j) {
            d = min(d, dis(p[vec[i]], p[vec[j]]));
        }
    }
    return d;
}

```

### 6.8 Circle

```

struct circle {
    pt o;
    type r;
    circle() {}
    circle(const pt &p, const type &d): o(p), r(d) {}
    circle(const type &x, const type &y, const type &d):
        o(pt(x, y)), r(d) {}
    double operator()(const { return pi * r * r; }
    bool overlap(const pt &p) const { return p.dist(o) <=
        r; }
    bool on(const pt &p) const { return p.dist(o) == r; }
};

```

### 6.9 Minimum Enclosing Circle

```

pt center(const pt &a, const pt &b, const pt &c) {
    pt p0 = b - a, p1 = c - a;
    double c1 = norm2(p0) * 0.5, c2 = norm2(p1) * 0.5;
    double d = p0 ^ p1;
    double x = a.x + (c1 * p1.y - c2 * p0.y) / d;
    double y = a.y + (c2 * p0.x - c1 * p1.x) / d;
    return pt(x, y);
}

```

```

circle min_enclosing(vector<pt> &p) {
    random_shuffle(p.begin(), p.end());
    double r = 0.0;
    pt cent;
    for (int i = 0; i < p.size(); ++i) {
        if (norm2(cent - p[i]) <= r) continue;
        cent = p[i];
        r = 0.0;
        for (int j = 0; j < i; ++j) {
            if (norm2(cent - p[j]) <= r) continue;
            cent = (p[i] + p[j]) / 2;
            r = norm2(p[j] - cent);
            for (int k = 0; k < j; ++k) {
                if (norm2(cent - p[k]) <= r) continue;
                cent = center(p[i], p[j], p[k]);
            }
        }
    }
}

```

```

        r = norm2(p[k] - cent);
    }
}
return circle(cent, sqrt(r));
}

```

## 6.10 Triangle Center

```

pt incenter(const pt &pa, const pt &pb, const pt &pc) {
    double a = norm(p[1] - pc), b = norm(p[2] - pa), c = norm
    (pa - pb);
    return (pa * a + pb * b + pc * c) / (a + b + c);
}

pt circumcenter(const pt &pa, const pt &pb, const pt &
pc) {
    pt ba = b - a, ca = c - a;
    double db = norm2(ba), dc = norm2(ca), d = 2.0 * (ba
    ^ ca);
    return a + pt(ba.y * dc - ca.y * db, ca.x * db - ba.x
    * dc) / 2;
}

pt orthocenter(const pt &pa, const pt &pb, const pt &pc
) {
    pt pa = pb - pa, ca = pc - pa, bc = pb - pc;
    double yp = ca.y * ca.y * bc.y;
    double a = (ca ^ ba);
    double x = (yp + ca.x * ba.y * pb.x - ba.x * ca.y *
    pc.x) / a;
    double y = -ba.x * (x - pc.x) / ba.y + ca.y;
    return pt(x, y);
}

```

## 7 String

### 7.1 KMP

```

int f[maxn];

int kmp(const string& a, const string& b) {
    f[0] = -1; f[1] = 0;
    for (int i = 1, j = 0; i < b.size() - 1; f[++i] = ++j)
    {
        if (b[i] == b[j]) f[i] = f[j];
        while (j != -1 && b[i] != b[j]) j = f[j];
    }
    for (int i = 0, j = 0; i - j + b.size() <= a.size();
    ++i, ++j) {
        while (j != -1 && a[i] != b[j]) j = f[j];
        if (j == b.size() - 1) return i - j;
    }
    return -1;
}

```

### 7.2 Suffix Array

```

struct SuffixArray {
    int sa[maxn], tmp[2][maxn], c[maxn], _lcp[maxn], r[
    maxn], n;
    string s;
    SparseTable st;
    void suffixarray() {
        int* rank = tmp[0];
        int* nRank = tmp[1];
        int A = 128;
        for (int i = 0; i < A; ++i) c[i] = 0;
        for (int i = 0; i < s.length(); ++i) c[rank[i] = s[
        i]]++;
        for (int i = 1; i < A; ++i) c[i] += c[i - 1];
        for (int i = s.length() - 1; i >= 0; --i) sa[--c[s[
        i]]] = i;
        for (int n = 1; n < s.length(); n *= 2) {

```

```

            for (int i = 0; i < A; ++i) c[i] = 0;
            for (int i = 0; i < s.length(); ++i) c[rank[i]
            ]++;
            for (int i = 1; i < A; ++i) c[i] += c[i - 1];
            int* sa2 = nRank;
            int r = 0;
            for (int i = s.length() - n; i < s.length(); ++i)
            sa2[r++] = i;
            for (int i = 0; i < s.length(); ++i) if (sa[i] >=
            n) sa2[r++] = sa[i] - n;
            for (int i = s.length() - 1; i >= 0; --i) sa[--c[
            rank[sa2[i]]]] = sa2[i];
            nRank[sa[0]] = r = 0;
            for (int i = 1; i < s.length(); ++i) {
                if (!(rank[sa[i - 1]] == rank[sa[i]] && sa[i -
                1] + n < s.length() && rank[sa[i - 1] + n] == rank[
                sa[i] + n])) r++;
                nRank[sa[i]] = r;
            }
            swap(rank, nRank);
            if (r == s.length() - 1) break;
            A = r + 1;
        }
    }
    void solve() {
        suffixarray();
        for (int i = 0; i < n; ++i) r[sa[i]] = i;
        int ind = 0; _lcp[0] = 0;
        for (int i = 0; i < n; ++i) {
            if (!r[i]) { ind = 0; continue; }
            while (i + ind < n && s[i + ind] == s[sa[r[i] -
            1] + ind]) ++ind;
            _lcp[r[i]] = ind ? ind-- : 0;
        }
        st = SparseTable(n, _lcp);
    }
    int lcp(int L, int R) {
        if (L == R) return n - L - 1;
        L = r[L]; R = r[R];
        if (L > R) swap(L, R);
        ++L;
        return st.query(L, R);
    }
    SuffixArray(string s): s(s), n(s.length()) {}
    SuffixArray() {}
};

```

### 7.3 Z Algorithm

```

int z[maxn];

void z_function(const string& s) {
    memset(z, 0, sizeof(z));
    z[0] = (int)s.length();
    int l = 0, r = 0;
    for (int i = 1; i < s.length(); ++i) {
        z[i] = max(0, min(z[i - l], r - i + 1));
        while (i + z[i] < s.length() && s[z[i]] == s[i + z[
        i]]) {
            l = i; r = i + z[i];
            ++z[i];
        }
    }
}

```

### 7.4 Manacher's Algorithm

```

int z[maxn];

int manacher(const string& s) {
    string t = ".";
    for (int i = 0; i < s.length(); ++i) t += s[i], t +=
    '.';
    int l = 0, r = 0;
    for (int i = 1; i < t.length(); ++i) {
        z[i] = (r > i ? min(z[2 * l - i], r - i) : 1);
        while (i - z[i] >= 0 && i + z[i] < t.length() && t[
        i - z[i]] == t[i + z[i]]) ++z[i];
    }
}

```

```

    if (i + z[i] > r) r = i + z[i], l = i;
}
int ans = 0;
for (int i = 1; i < t.length(); ++i) ans = max(ans, z[i] - 1);
return ans;
}

```

## 7.5 Smallest Rotation

```

string rotate(const string &s) {
    int n = s.length();
    string t = s + s;
    int i = 0, j = 1;
    while (i < n && j < n) {
        int k = 0;
        while (k < n && s[i + k] == s[j + k]) ++k;
        if (s[i + k] <= s[j + k]) j += k + 1;
        else i += k + 1;
        if (i == j) ++j;
    }
    int pos = (i < n ? i : j);
    return s.substr(pos, n);
}

```

## 7.6 Rolling Hash Primes

```

const int mod[] = { 479001599, 433494437, 1073807359,
    1442968193, 715827883 };
const int p[] = { 101, 233, 457, 173, 211 };

```

# 8 Dynamic Programming

## 8.1 Concept

1. Convex Hull Optimization:  $dp[i] = \max_{j < i} \{a[j]x[i] + b[j]\}$
2. Concave totally Montone:  $\forall i < i', j < j', f[i][j] \leq f[i'][j] \Rightarrow f[i][j'] \leq f[i'][j']$
3. Convex totally Montone:  $\forall i < i', j < j', f[i][j] \geq f[i'][j] \Rightarrow f[i][j'] \geq f[i'][j']$

## 8.2 Linear Convex Hull Optimization

```

struct ConvexHull {
    // Max convex hull
    deque<pair<int, int>> dq;
    bool check(const pair<int, int>& l1, const pair<int, int>& l2, int x) {
        // for min case, replace <= with >=
        return l1.first * x + l1.second <= l2.first * x + l2.second;
    }
    bool elim(const pair<int, int>& l1, const pair<int, int>& l2, const pair<int, int>& l3) {
        return (double)(l1.second - l2.second) / (double)(l2.first - l1.first) <= (double)(l2.second - l3.second) / (double)(l3.first - l2.first);
    }
    int query(int x) {
        while (dq.size() >= 2 && check(dq[0], dq[1], x)) dq.pop_front();
        return dq.front().first * x + dq.front().second;
    }
    void add(int a, int b) {
        while (dq.size() >= 2 && elim(dq[dq.size() - 1], dq[dq.size() - 2], make_pair(a, b))) dq.pop_back();
        dq.push_back(make_pair(a, b));
    }
};

```

## 8.3 Convex Hull Optimization

```

int dp[maxn], type;

struct line {
    int a, b, l, r;
    int operator()(const int& d) const { return a * d + b; }
    int intersect(const line& rhs) const { return (rhs.b - b) / (a - rhs.a); }
    bool operator<(const line& rhs) const { return type ? r < rhs.l : a < rhs.a; }
};

void solve() {
    set<line> st; st.insert((line){ a[0], b[0], -inf, inf });
    for (int i = 1; i <= n + 1; ++i) {
        dp[i] = (*st.lower_bound((line){ 0, 0, x[i], x[i] }));
        line nl = (line){ a[i], b[i], -inf, inf };
        auto it = st.lower_bound(nl);
        bool ins = true;
        while (it != st.end()) {
            if ((*it)(it->l) > nl(it->l) && (*it)(it->r) > nl(it->r)) {
                ins = false;
                break;
            }
            if ((*it)(it->l) <= nl(it->l) && (*it)(it->r) <= nl(it->r)) it = st.erase(it);
            else break;
        }
        if (!ins) continue;
        while (it != st.begin()) {
            --it;
            if ((*it)(it->l) <= nl(it->l) && (*it)(it->r) <= nl(it->r)) it = st.erase(it);
            else { ++it; break; }
        }
        if (it != st.begin()) {
            --it;
            line tmp = *it; it = st.erase(it);
            long long t = nl.intersect(tmp);
            tmp.r = t; nl.l = t + 1;
            st.insert(tmp);
        }
        if (it != st.end()) {
            line tmp = *it; it = st.erase(it);
            long long t = nl.intersect(tmp);
            tmp.l = t + 1; nl.r = t;
            st.insert(tmp);
        }
        st.insert(nl);
    }
}

```

## 8.4 Convex Montone Optimization

```

void convex() {
    deque<segment> dq; dq.push_back({ 0, 1, n });
    for (int i = 1; i <= n; ++i) {
        dp[i] = f(dq.front().i, i);
        while (dq.size() && dq.front().r < i + 1) dq.pop_front();
        dq.front().l = i + 1;
        while (dq.size() && f(i, dq.back().l) < f(dq.back().i, dq.back().l)) dq.pop_back();
        segment new_seg = { i, i + 1, n };
        if (dq.size()) {
            int c = bs();
            // c is the maximal k such that f(i, k) > f(dq.back().i, k)
            dq.back().r = c;
            new_seg.l = c + 1;
        }
        if (new_seg.l <= n) dq.push_back(new_seg);
    }
}

```

```
|}
```

## 8.5 Divide and Conquer Optimization

```
int dp[maxk][maxn], f[maxk][maxn];

void go(int k, int l, int r, int fl, int fr) {
    if (l > r) return;
    int m = (l + r) >> 1;
    f[k][m] = -1;
    for (int i = fl; i <= min(m - 1, fr); ++i) {
        int t = dp[k - 1][i] + f(i + 1, m);
        if (t > dp[k][m]) {
            dp[k][m] = t;
            f[k][m] = i;
        }
    }
    go(k, l, m - 1, fl, f[k][m]);
    go(k, m + 1, r, f[k][m], fr);
}

void solve() {
    for (int i = 1; i <= k; ++i) go(i, 1, n, 0, n - 1);
}
```