Notebook Regional 2025 Champán en Lata

UTN FRSF - Champán en Lata

2025

Contents

1 Template

```cpp
#include <bits/stdc++.h>
#define forr(i, a, b) for (int i = (a); i < (b); i++)
#define forn(i, n) forr(i, 0, n)
#define dforn(i, n) for (int i = (n) - 1; i >= 0; i--)
#define forall(it, v) for (auto it = v.begin(); it != v.end(); it++)
#define sz(c) ((int)c.size())
#define rsz resize
#define pb push_back
#define mp make_pair
#define lb lower_bound
#define ub upper_bound
#define fst first
#define snd second

#ifdef ANARAP
// local
#else
// judge
#endif

using namespace std;

typedef long long ll;
typedef pair<int, int> ii;

int main() {
// agregar g++ -DANARAP en compilacion
#ifdef ANARAP
  freopen("input.in", "r", stdin);
  // freopen("output.out","w", stdout);
#endif
  ios::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);
  return 0;
}
```

2   Math

2.1   Identidades

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$
$$\sum_{i=0}^{n} i\binom{n}{i} = n * 2^{n-1}$$
$$\sum_{i=m}^{n} i = \frac{n(n+1)}{2} - \frac{m(m-1)}{2} = \frac{(n+1-m)(n+m)}{2}$$
$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$
$$\sum_{i=0}^{n} i(i-1) = \frac{8}{6}(\frac{n}{2})(\frac{n}{2}+1)(n+1) \text{ (doubles)} \rightarrow \text{Sino ver caso impar y par}$$
$$\sum_{i=0}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} = [\sum_{i=1}^{n} i]^2$$
$$\sum_{i=0}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n}{30}$$
$$\sum_{i=0}^{n} i^p = \frac{(n+1)^{p+1}}{p+1} + \sum_{k=1}^{p} \frac{B_k}{p-k+1}\binom{p}{k}(n+1)^{p-k+1}$$
$$r = e - v + k + 1$$

Teorema de Pick: (Area, puntos interiores y puntos en el borde)
$$A = I + \frac{B}{2} - 1$$

2.2   Ec. Caracteristica

$$a_0 T(n) + a_1 T(n-1) + ... + a_k T(n-k) = 0$$
$$p(x) = a_0 x^k + a_1 x^{k-1} + ... + a_k$$

Sean $r_1, r_2, ..., r_q$ las raíces distintas, de mult. $m_1, m_2, ..., m_q$
$$T(n) = \sum_{i=1}^{q} \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$$

Las constantes c_{ij} se determinan por los casos base.

2.3   Modular operations

```cpp
const ll MOD = 1000000007;  // Change according to problem
// Only needed for MOD > 2^31
// Actually, for 2^31 < MOD < 2^63 it's usually better to use __int128
// and normal multiplication (* operator) instead of mulMod
// returns (a*b) %c, and minimize overfloor
ll mulMod(ll a, ll b, ll m = MOD) {  // O(log b)
  ll x = 0, y = a % m;
  while (b > 0) {
    if (b % 2 == 1) x = (x + y) % m;
    y = (y * 2) % m;
    b /= 2;
  }
  return x % m;
}
ll expMod(ll b, ll e, ll m = MOD) {  // O(log e)
  if (e < 0) return 0;
  ll ret = 1;
  while (e) {
    if (e & 1) ret = ret * b % m;  // ret = mulMod(ret,b,m); //if needed
    b = b * b % m;                 // b = mulMod(b,b,m);
    e >>= 1;
  }
  return ret;
}
ll sumMod(ll a, ll b, ll m = MOD) {
  a %= m;
  b %= m;
  if (a < 0) a += m;
  if (b < 0) b += m;
  return (a + b) % m;
}
ll difMod(ll a, ll b, ll m = MOD) {
  a %= m;
  b %= m;
  if (a < 0) a += m;
  if (b < 0) b += m;
  ll ret = a - b;
  if (ret < 0) ret += m;
  return ret;
}
ll divMod(ll a, ll b, ll m = MOD) { return mulMod(a, inverso(b), m); }
```

2.4   Chinese reminder theorem, extended euclid and diophantine equations

$$y = \sum_{j=1}^{n} (x_j * (\prod_{i=1, i\neq j}^{n} m_i)^{-1}_{m_j} * \prod_{i=1, i\neq j}^{n} m_i)$$

```cpp
// ecuacion diofantica lineal
// sea d=gcd(a,b); la ecuacion a * x + b * y = c tiene soluciones enteras si
// d|c. La siguiente funcion nos sirve para esto. De forma general sera:
// x = x0 + (b/d)n        x0 = xx*c/d
// y = y0 - (a/d)n        y0 = yy*c/d
// la funcion devuelve d
ll euclid(ll a, ll b, ll &xx, ll &yy) {
  if (!b) return xx = 1, yy = 0, a;
  ll d = euclid(b, a % b, yy, xx);
  return yy -= a / b * xx, d;
}

// Chinese remainder theorem (special case): find z such that
// z % m = a, z % n = b.  Here, z is unique modulo M = lcm(m, n)
// Return -1 when there is no solution
// CRT is associative and idempotent
ll crt(ll a, ll m, ll b, ll n) {
  if (n > m) swap(a, b), swap(m, n);
  ll x, y, g = euclid(m, n, x, y);
  if ((a - b) % g != 0) return -1; // comment to get RTE when there is no
         solution
  assert((a - b) % g == 0);
  x = (b - a) % n * x % n / g * m + a;
  return x < 0 ? x + m * n / g : x;
}

// Chinese remainder theorem: find z such that z % m[i] = r[i] for all i.
// Note that the solution is unique modulo M = lcm_i (m[i]).
// Return z. On failure, return -1.
// Note that we do not require the m[i]'s to be relatively prime.
ll crt(const vector<ll>& r, const vector<ll>& m) {
  assert(sz(r) == sz(m));
  ll ret = r[0], l = m[0];
  forr(i, 1, sz(m)) {
    ret = crt(ret, l, r[i], m[i]);
    l = lcm(l,m[i]);
    if (ret == -1) break;
  }
  return ret;
}
```

2.5   Modular inverse

```cpp
#define MAXMOD 15485867
ll inv[MAXMOD];       // inv[i]*i=1 mod MOD
void calc(int p) {  // O(p)
  inv[1] = 1;
  forr(i, 2, p) inv[i] = p - ((p / i) * inv[p % i]) % p;
}
int inverso(int x) {                      // O(log MOD)
  return expMod(x, eulerPhi(MOD) - 1);  // si mod no es primo(sacar a mano)
  return expMod(x, MOD - 2);            // si mod es primo
}
```

```cpp
// fact[i] = i!%MOD and ifact[i] = 1/(i!)%MOD
// inv is modular inverse function
ll fact[MAXN], ifact[MAXN];
void build_facts() {  // O(MAXN)
  fact[0] = 1;
  forr(i, 1, MAXN) fact[i] = fact[i - 1] * i % MOD;
  ifact[MAXN - 1] = inverso(fact[MAXN - 1]);
  dforn(i, MAXN - 1) ifact[i] = ifact[i + 1] * (i + 1) % MOD;
  return;
}
// n! / k!*(n-k)!
// assumes 0 <= n < MAXN
ll comb(ll n, ll k) {
  if (k < 0 || n < k) return 0;
  return fact[n] * ifact[k] % MOD * ifact[n - k] % MOD;
}
```

2.6   Combinatorics

```cpp
void cargarComb() {  // O(MAXN^2)
  forn(i, MAXN) {        // comb[i][k]=i tomados de a k = i!/(k!*(i-k)!)
    comb[0][i] = 0;
    comb[i][0] = comb[i][i] = 1;
    forr(k, 1, i) comb[i][k] = (comb[i - 1][k - 1] + comb[i - 1][k]) % MOD;
  }
}
ll lucas(ll n, ll k, int p) {  // (n,k)%p, needs comb[p][p] precalculated
  ll aux = 1;
  while (n + k) {
    aux = (aux * comb[n % p][k % p]) % p;
    n /= p, k /= p;
  }
  return aux;
}
```

2.7   Discrete Logarithm

```cpp
// O(sqrt(m)*log(m))
// returns x such that a^x = b (mod m) or -1 if inexistent
ll discrete_log(ll a, ll b, ll m) {
  a %= m, b %= m;
  if (b == 1) return 0;
  int cnt = 0;
  ll tmp = 1;
  for (ll g = __gcd(a, m); g != 1; g = __gcd(a, m)) {
    if (b % g) return -1;
    m /= g, b /= g;
    tmp = tmp * a / g % m;
    ++cnt;
    if (b == tmp) return cnt;
```

```
14    }
15    map<ll, int> w;
16    int s = (int)ceil(sqrt(m));
17    ll base = b;
18    forn(i, s) {
19      w[base] = i;
20      base = base * a % m;
21    }
22    base = expMod(a, s, m);
23    ll key = tmp;
24    forr(i, 1, s + 2) {
25      key = base * key % m;
26      if (w.count(key)) return i * s - w[key] + cnt;
27    }
28    return -1;
29 }
```

2.8   Fractions

```
1  struct frac {
2    int p, q;
3    frac(int p = 0, int q = 1) : p(p), q(q) { norm(); }
4    void norm() {
5      int a = gcd(q, p);
6      if (a) p /= a, q /= a;
7      else q = 1;
8      if (q < 0) q = -q, p = -p;
9    }
10    frac operator+(const frac& o) {
11      int a = gcd(o.q, q);
12      return frac(p * (o.q / a) + o.p * (q / a), q * (o.q / a));
13    }
14    frac operator-(const frac& o) {
15      int a = gcd(o.q, q);
16      return frac(p * (o.q / a) - o.p * (q / a), q * (o.q / a));
17    }
18    frac operator*(frac o) {
19      int a = gcd(o.p, q), b = gcd(p, o.q);
20      return frac((p / b) * (o.p / a), (q / a) * (o.q / b));
21    }
22    frac operator/(frac o) {
23      int a = gcd(o.q, q), b = gcd(p, o.p);
24      return frac((p / b) * (o.q / a), (q / a) * (o.p / b));
25    }
26    bool operator<(const frac& o) const { return p * o.q < o.p * q; }
27    bool operator==(frac o) { return p == o.p && q == o.q; }
28  };
```

2.9   Matrix exponentiation

```
1  typedef ll tipo;  // maybe use double or other depending on the problem
2  struct Mat {
3    int N;  // square matrix
4    vector<vector<tipo>> m;
5    Mat(int n) : N(n), m(n, vector<tipo>(n, 0)) {}
6    vector<tipo>& operator[](int p) { return m[p]; }
7    Mat operator*(Mat& b) {  // O(N^3), multiplication
8      assert(N == b.N);
9      Mat res(N);
10      forn(i, N) forn(j, N) forn(k, N)  // remove MOD if not needed
11          res[i][j] = (res[i][j] + m[i][k] * b[k][j]) % MOD;
12      return res;
13    }
14    Mat operator^(int k) {  // O(N^3 * logk), exponentiation
15      Mat res(N), aux = *this;
16      forn(i, N) res[i][i] = 1;
17      while (k)
18        if (k & 1) res = res * aux, k--;
19        else aux = aux * aux, k /= 2;
20      return res;
```

2.10   Primes, divisors and Phi

```cpp
#define MAXP 100000  // no necesariamente primo
int criba[MAXP + 1];
void crearCriba() {
  int w[] = {4, 2, 4, 2, 4, 6, 2, 6};
  for (int p = 25; p <= MAXP; p += 10) criba[p] = 5;
  for (int p = 9; p <= MAXP; p += 6) criba[p] = 3;
  for (int p = 4; p <= MAXP; p += 2) criba[p] = 2;
  for (int p = 7, cur = 0; p * p <= MAXP; p += w[cur++ & 7])
    if (!criba[p])
      for (int j = p * p; j <= MAXP; j += (p << 1))
        if (!criba[j]) criba[j] = p;
}
vector<int> primos;
void buscarPrimos() {
  crearCriba();
  forr(i, 2, MAXP + 1) if (!criba[i]) primos.push_back(i);
}

// factoriza bien numeros hasta MAXP^2, llamar a buscarPrimos antes
void fact(ll n, map<ll, ll>& f) {  // O (cant primos)
  forall(p, primos) {
    while (!(n % *p)) {
      f[*p]++;  // divisor found
      n /= *p;
    }
  }
  if (n > 1) f[n]++;
}

// factoriza bien numeros hasta MAXP, llamar crearCriba antes
void fact2(ll n, map<ll, ll>& f) {  // O (lg n)
  while (criba[n]) {
    f[criba[n]]++;
    n /= criba[n];
  }
  if (n > 1) f[n]++;
}

// Usar asi: divisores(fac, divs, fac.begin()); NO ESTA ORDENADO
void divisores(map<ll, ll>& f, vector<ll>& divs, map<ll, ll>::iterator it,
               ll n = 1) {
  if (it == f.begin()) divs.clear();
  if (it == f.end()) {
    divs.pb(n);
    return;
  }
  ll p = it->fst, k = it->snd;
  ++it;
  forn(_, k + 1) divisores(f, divs, it, n), n *= p;
}
```

```cpp
}
ll cantDivs(map<ll, ll>& f) {
  ll ret = 1;
  forall(it, f) ret *= (it->second + 1);
  return ret;
}
ll sumDivs(map<ll, ll>& f) {
  ll ret = 1;
  forall(it, f) {
    ll pot = 1, aux = 0;
    forn(i, it->snd + 1) aux += pot, pot *= it->fst;
    ret *= aux;
  }
  return ret;
}

ll eulerPhi(ll n) {  // con criba: O(lg n)
  map<ll, ll> f;
  fact(n, f);
  ll ret = n;
  forall(it, f) ret -= ret / it->first;
  return ret;
}
ll eulerPhi2(ll n) {  // O (sqrt n)
  ll r = n;
  forr(i, 2, n + 1) {
    if ((ll)i * i > n) break;
    if (n % i == 0) {
      while (n % i == 0) n /= i;
      r -= r / i;
    }
  }
  if (n != 1) r -= r / n;
  return r;
}
```

2.11   Phollard's Rho

```cpp
bool es_primo_prob(ll n, int a) {
  if (n == a) return true;
  ll s = 0, d = n - 1;
  while (d % 2 == 0) s++, d /= 2;
  ll x = expMod(a, d, n);
  if ((x == 1) || (x + 1 == n)) return true;
  forn(i, s - 1) {
    x = (x * x) % n; // mulMod(x, x, n); In most cases, it is necessary to use
         mulMod to avoid TLE
    if (x == 1) return false;
    if (x + 1 == n) return true;
  }
  return false;
}
bool rabin(ll n) {  // devuelve true si n es primo
  if (n == 1) return false;
  const int ar[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
  forn(j, 9) if (!es_primo_prob(n, ar[j])) return false;
  return true;
}
ll rho(ll n) {
  if ((n & 1) == 0) return 2;
  ll x = 2, y = 2, d = 1;
  ll c = rand() % n + 1;
  while (d == 1) {
    // may want to avoid mulMod if possible
    // maybe replace with * operator using __int128?
    x = (mulMod(x, x, n) + c) % n;
    y = (mulMod(y, y, n) + c) % n;
    y = (mulMod(y, y, n) + c) % n;
    if (x - y >= 0) d = gcd(n, x - y);
    else d = gcd(n, y - x);
  }
  return d == n ? rho(n) : d;
}
void factRho(ll n, map<ll, ll>& f) {  // O ((n ^ 1/4) * logn)
  if (n == 1) return;
  if (rabin(n)) {
    f[n]++;
    return;
  }
  ll factor = rho(n);
  factRho(factor, f);
  factRho(n / factor, f);
}
```

2.12   Guass-Jordan

```cpp
// https://cp-algorithms.com/linear_algebra/linear-system-gauss.html
const double EPS = 1e-9;
const int INF = 2;  // a value to indicate infinite solutions

int gauss(vector<vector<double> > a, vector<double>& ans) {
  int n = (int)a.size();
  int m = (int)a[0].size() - 1;

  vector<int> where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; ++col) {
    int sel = row;
    for (int i = row; i < n; ++i)
      if (abs(a[i][col]) > abs(a[sel][col])) sel = i;
    if (abs(a[sel][col]) < EPS) continue;
    for (int i = col; i <= m; ++i) swap(a[sel][i], a[row][i]);
    where[col] = row;

    for (int i = 0; i < n; ++i)
      if (i != row) {
        double c = a[i][col] / a[row][col];
        for (int j = col; j <= m; ++j) a[i][j] -= a[row][j] * c;
      }
    ++row;
  }

  ans.assign(m, 0);
  for (int i = 0; i < m; ++i)
    if (where[i] != -1) ans[i] = a[where[i]][m] / a[where[i]][i];
  for (int i = 0; i < n; ++i) {
    double sum = 0;
    for (int j = 0; j < m; ++j) sum += ans[j] * a[i][j];
    if (abs(sum - a[i][m]) > EPS) return 0;
  }

  for (int i = 0; i < m; ++i)
    if (where[i] == -1) return INF;
  return 1;
}
```

2.13   Guass-Jordan modular

```cpp
// inv -> modular inverse function
// disclaimer: not very well tested, but got AC on a problem with this
int gauss(vector<vector<int> > a, vector<int>& ans) {
  int n = (int)a.size();
  int m = (int)a[0].size() - 1;

  vector<int> where(m, -1);
  for (int col = 0, row = 0; col < m && row < n; ++col) {
    int sel = row;
    for (int i = row; i < n; ++i)
      if (a[i][col] > a[sel][col]) sel = i;
    if (a[sel][col] == 0) continue;
    for (int i = col; i <= m; ++i) swap(a[sel][i], a[row][i]);
    where[col] = row;

    for (int i = 0; i < n; ++i)
      if (i != row) {
        int c = (a[i][col] * inv(a[row][col])) % MOD;
        for (int j = col; j <= m; ++j)
          a[i][j] = (a[i][j] - a[row][j] * c % MOD + MOD) % MOD;
      }
    ++row;
  }
  ans.clear();
  ans.rsz(m, 0);
  for (int i = 0; i < m; ++i)
    if (where[i] != -1) ans[i] = (a[where[i]][m] * inv(a[where[i]][i])) % MOD;
  for (int i = 0; i < n; ++i) {
    int sum = 0;
    for (int j = 0; j < m; ++j) sum = (sum + ans[j] * a[i][j]) % MOD;
    if ((sum - a[i][m] + MOD) % MOD != 0) return 0;
  }

  for (int i = 0; i < m; ++i)
    if (where[i] == -1) return INF;
  return 1;
}
```

2.14   Guass-Jordan with bitset

```cpp
// https://cp-algorithms.com/linear_algebra/linear-system-gauss.html
// special case of gauss_jordan_mod with mod=2, bitset for efficiency
// finds lexicograhically minimal solution (0 < 1, False < True)
// for lexicographically maximal change your solution model accordingly
int gauss(vector<bitset<N> > a, int n, int m, bitset<N>& ans) {
  vector<int> where(m, -1);
  for (int col = m - 1, row = 0; col >= 0 && row < n; --col) {
    for (int i = row; i < n; ++i)
      if (a[i][col]) {
        swap(a[i], a[row]);
        break;
      }
    if (!a[row][col]) continue;
    where[col] = row;

    for (int i = 0; i < n; ++i)
      if (i != row && a[i][col]) a[i] ^= a[row];
    ++row;
  }
  ans.reset();
  forn(i, m) if (where[i] != -1) { ans[i] = a[where[i]][m] & a[where[i]][i]; }
  forn(i, n) if ((ans & a[i]).count() % 2 != a[i][m]) return 0;
  forn(i, m) if (where[i] == -1) return INF;
  return 1;
}
```

2.15   Simpson

```cpp
typedef long double T;
// polar coordinates: x=r*cos(theta), y=r*sin(theta),  f=(r*r)/2
T simpson(std::function<T(T)> f, T a, T b, int n = 10000) {  // O(n)
  T area = 0, h = (b - a) / T(n), fa = f(a), fb;
  forn(i, n) {
    fb = f(a + h * T(i + 1));
    area += fa + T(4) * f(a + h * T(i + 0.5)) + fb;
    fa = fb;
  }
  return area * h / T(6.);
}
```

2.16   Fast Fourier Transform (FFT)

```cpp
typedef __int128 T;
typedef double ld;
typedef vector<T> poly;
const T MAXN = (1 << 21);   // MAXN must be power of 2,
// MOD-1 needs to be a multiple of MAXN, big mod and primitive root for NTT
const T MOD = 2305843009255636993LL, RT = 5;
// const T MOD = 998244353, RT = 3;

// NTT
struct CD {
  T x;
  CD(T x_) : x(x_) {}
  CD() {}
};
T mulmod(T a, T b) { return a * b % MOD; }
T addmod(T a, T b) {
  T r = a + b;
  if (r >= MOD) r -= MOD;
  return r;
}
T submod(T a, T b) {
  T r = a - b;
  if (r < 0) r += MOD;
  return r;
}
CD operator*(const CD& a, const CD& b) { return CD(mulmod(a.x, b.x)); }
CD operator+(const CD& a, const CD& b) { return CD(addmod(a.x, b.x)); }
CD operator-(const CD& a, const CD& b) { return CD(submod(a.x, b.x)); }
vector<T> rts(MAXN + 9, -1);
CD root(int n, bool inv) {
  T r = rts[n] < 0 ? rts[n] = expMod(RT, (MOD - 1) / n) : rts[n];
  return CD(inv ? expMod(r, MOD - 2) : r);
}

// FFT
// struct CD {
//   ld r, i;
//   CD(ld r_ = 0, ld i_ = 0) : r(r_), i(i_) {}
//   ld real() const { return r; }
//   void operator/=(const int c) { r /= c, i /= c; }
// };
// CD operator*(const CD& a, const CD& b) {
//   return CD(a.r * b.r - a.i * b.i, a.r * b.i + a.i * b.r);
// }
// CD operator+(const CD& a, const CD& b) { return CD(a.r + b.r, a.i + b.i); }
// CD operator-(const CD& a, const CD& b) { return CD(a.r - b.r, a.i - b.i); }
// const ld pi = acos(-1.0);

CD cp1[MAXN + 9], cp2[MAXN + 9];
int R[MAXN + 9];
void dft(CD* a, int n, bool inv) {
  forn(i, n) if (R[i] < i) swap(a[R[i]], a[i]);
  for (int m = 2; m <= n; m *= 2) {
    // ld z=2*pi/m*(inv?-1:1); // FFT
    // CD wi=CD(cos(z),sin(z)); // FFT
    CD wi = root(m, inv);   // NTT
    for (int j = 0; j < n; j += m) {
      CD w(1);
      for (int k = j, k2 = j + m / 2; k2 < j + m; k++, k2++) {
        CD u = a[k];
        CD v = a[k2] * w;
        a[k] = u + v;
        a[k2] = u - v;
        w = w * wi;
      }
    }
  }
  // if(inv) forn(i,n) a[i]/=n; // FFT
  if (inv) {   // NTT
    CD z(expMod(n, MOD - 2));
    forn(i, n) a[i] = a[i] * z;
  }
}
poly multiply(poly& p1, poly& p2) {
  int n = sz(p1) + sz(p2) + 1;
  int m = 1, cnt = 0;
  while (m <= n) m += m, cnt++;
  forn(i, m) {
    R[i] = 0;
    forn(j, cnt) R[i] = (R[i] << 1) | ((i >> j) & 1);
  }
  forn(i, m) cp1[i] = 0, cp2[i] = 0;
  forn(i, sz(p1)) cp1[i] = p1[i];
  forn(i, sz(p2)) cp2[i] = p2[i];
  dft(cp1, m, false);
  dft(cp2, m, false);
  forn(i, m) cp1[i] = cp1[i] * cp2[i];
  dft(cp1, m, true);
  poly res;
  n -= 2;
  // forn(i,n) res.pb((T) floor(cp1[i].real()+0.5)); // FFT
  forn(i, n) res.pb(cp1[i].x);   // NTT
  return res;
}
```

2.17   Karatsuba

```cpp
template<typename T> void rec_kara(T* a, int one, T* b, int two, T* r) {
  if(min(one, two) <= 20) { // must be at least "<= 1"
    forn(i, one) forn(j, two) r[i+j] += a[i] * b[j];
    return;
  }
  const int x = min(one, two);
  if(one < two) rec_kara(a, x, b + x, two - x, r + x);
  if(two < one) rec_kara(a + x, one - x, b, x, r + x);
  const int n = (x + 1) / 2, right = x / 2;
  vector<T> tu(2 * n);
  rec_kara(a, n, b, n, tu.data());
```

```
12    forn(i, 2*n-1) {
13      r[i]  += tu[i];
14      r[i+n] -= tu[i];
15      tu[i] = 0;
16    }
17    rec_kara(a + n, right, b + n, right, tu.data());
18    forn(i, 2*right-1) r[i+n] -= tu[i], r[i+2*n] += tu[i];
19    tu[n-1] = a[n-1]; tu[2*n-1] = b[n-1];
20    forn(i, right) tu[i] = a[i]+a[i+n], tu[i+n] = b[i]+b[i+n];
21    rec_kara(tu.data(), n, tu.data() + n, n, r + n);
22  }
23  template<typename T> vector<T> multiply(vector<T> a, vector<T> b) {
24    if(a.empty() || b.empty()) return {};
25    vector<T> r(a.size() + b.size() - 1);
26    rec_kara(a.data(), a.size(), b.data(), b.size(), r.data());
27    return r;
28  }
```

2.18   Simplex

```
1  typedef double tipo;
2  typedef vector<tipo> vt;
3  // maximize c^T x s.t. Ax<=b, x>=0, returns pair (max val, solution vector)
4  pair<tipo, vt> simplex(vector<vt> A, vt b, vt c) {
5    int n = sz(b), m = sz(c);
6    tipo z = 0.;
7    vector<int> X(m), Y(n);
8    forn(i, m) X[i] = i;
9    forn(i, n) Y[i] = i + m;
10   auto pivot = [&](int x, int y) {
11     swap(X[y], Y[x]);
12     b[x] /= A[x][y];
13     forn(i, m) if (i != y) A[x][i] /= A[x][y];
14     A[x][y] = 1 / A[x][y];
15     forn(i, n) if (i != x && abs(A[i][y]) > EPS) {
16       b[i] -= A[i][y] * b[x];
17       forn(j, m) if (j != y) A[i][j] -= A[i][y] * A[x][j];
18       A[i][y] *= -A[x][y];
19     }
20     z += c[y] * b[x];
21     forn(i, m) if (i != y) c[i] -= c[y] * A[x][i];
22     c[y] *= -A[x][y];
23   };
24   while (1) {
25     int x = -1, y = -1;
26     tipo mn = -EPS;
27     forn(i, n) if (b[i] < mn) mn = b[i], x = i;
28     if (x < 0) break;
29     forn(i, m) if (A[x][i] < -EPS) {
30       y = i;
31       break;
32     }
33     assert(y >= 0);   // no solution to Ax<=b
34     pivot(x, y);
35   }
36   while (1) {
37     tipo mx = EPS;
38     int x = -1, y = -1;
39     forn(i, m) if (c[i] > mx) mx = c[i], y = i;
40     if (y < 0) break;
41     tipo mn = 1e200;
42     forn(i, n) if (A[i][y] > EPS && b[i] / A[i][y] < mn) {
43       mn = b[i] / A[i][y], x = i;
44     }
45     assert(x >= 0);   // c^T x is unbounded
46     pivot(x, y);
47   }
48   vt r(m);
49   forn(i, n) if (Y[i] < m) r[Y[i]] = b[i];
50   return {z, r};
51 }
```

2.19 Tablas y cotas (Primos, Divisores, Factoriales, etc)

Factoriales

$0! = 1$	$11! = 39.916.800$
$1! = 1$	$12! = 479.001.600 \ (\in \texttt{int})$
$2! = 2$	$13! = 6.227.020.800$
$3! = 6$	$14! = 87.178.291.200$
$4! = 24$	$15! = 1.307.674.368.000$
$5! = 120$	$16! = 20.922.789.888.000$
$6! = 720$	$17! = 355.687.428.096.000$
$7! = 5.040$	$18! = 6.402.373.705.728.000$
$8! = 40.320$	$19! = 121.645.100.408.832.000$
$9! = 362.880$	$20! = 2.432.902.008.176.640.000 \ (\in \texttt{tint})$
$10! = 3.628.800$	$21! = 51.090.942.171.709.400.000$

Primos

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103
107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313
317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433
439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563
569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673
677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811
821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941
947 953 967 971 977 983 991 997 1009 1013 1019 1021 1031 1033 1039 1049
1051 1061 1063 1069 1087 1091 1093 1097 1103 1109 1117 1123 1129 1151
1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249
1259 1277 1279 1283 1289 1291 1297 1301 1303 1307 1319 1321 1327 1361
1367 1373 1381 1399 1409 1423 1427 1429 1433 1439 1447 1451 1453 1459
1471 1481 1483 1487 1489 1493 1499 1511 1523 1531 1543 1549 1553 1559
1567 1571 1579 1583 1597 1601 1607 1609 1613 1619 1621 1627 1637 1657
1663 1667 1669 1693 1697 1699 1709 1721 1723 1733 1741 1747 1753 1759
1777 1783 1787 1789 1801 1811 1823 1831 1847 1861 1867 1871 1873 1877
1879 1889 1901 1907 1913 1931 1933 1949 1951 1973 1979 1987 1993 1997
1999 2003 2011 2017 2027 2029 2039 2053 2063 2069 2081

Primos cercanos a 10^n

9941 9949 9967 9973 10007 10009 10037 10039 10061 10067 10069 10079
99961 99971 99989 99991 100003 100019 100043 100049 100057 100069
999959 999961 999979 999983 1000003 1000033 1000037 1000039
9999943 9999971 9999973 9999991 10000019 10000079 10000103 10000121
99999941 99999959 99999971 99999989 100000007 100000037 100000039
100000049
999999893 999999929 999999937 1000000007 1000000009 1000000021
1000000033

Cantidad de primos menores que 10^n

$\pi(10^1) = 4$; $\pi(10^2) = 25$; $\pi(10^3) = 168$; $\pi(10^4) = 1229$; $\pi(10^5) = 9592$
$\pi(10^6) = 78.498$; $\pi(10^7) = 664.579$; $\pi(10^8) = 5.761.455$; $\pi(10^9) = 50.847.534$
$\pi(10^{10}) = 455.052,511$; $\pi(10^{11}) = 4.118.054.813$; $\pi(10^{12}) = 37.607.912.018$

2.20 Números Catalanes

Utiles para problemas de Combinatoria
$$Cat(n) = \frac{\binom{2n}{n}}{n+1} = \frac{(2n)!}{n! \, (n+1)!}$$
Con $Cat(0) = 1$.

Diferentes aplicaciones:
1. Contar la cantidad de diferentes arboles binarios con n nodos que se pueden armar.
2. Contar las formas en que un polígono convexo de $n + 2$ lados puede ser triangulado.
3. Contar la cantidad de caminos monotonos a lo largo de los lados de una grilla $n * n$, que no cruzan la diagonal.
4. Contar el número de expresiones que contienen n pares de paréntesis correctamente colocados

2.20.1 Primeros 25 Catalanes

1 1 2 5 14 42 132 429 1430 4862 16796 58786 208012 742900 2674440 9694845
35357670 129644790 477638700 1767263190 6564120420 24466267020
91482563640 343059613650 1289904147324 4861946401452

3   Geometry

3.1   Point

```cpp
typedef long double T; // double could be faster but less precise
typedef long double ld;
const T EPS = 1e-9; // if T is integer, set to 0
const T INF = 1e18;
struct pto{
  T x, y;
  pto() : x(0), y(0) {}
  pto(T _x, T _y) : x(_x), y(_y) {}
  pto operator+(pto b) { return pto(x+b.x, y+b.y); }
  pto operator-(pto b) { return pto(x-b.x, y-b.y); }
  pto operator+(T k) { return pto(x+k, y+k); }
  pto operator*(T k) { return pto(x*k, y*k); }
  pto operator/(T k) { return pto(x/k, y/k); }
  // dot product
  T operator*(pto b) { return x*b.x + y*b.y; }
  // module of cross product, a^b>0 if angle_cw(u,v)<180
  T operator^(pto b) { return x*b.y - y*b.x; }
  // vector projection of this above b
  pto proj(pto b) { return b*((*this)*b) / (b*b); }
  T norm_sq() { return x*x + y*y; }
  ld norm() { return sqrtl(x*x + y*y); }
  ld dist(pto b) { return (b - (*this)).norm(); }
  //rotate by theta rads CCW w.r.t. origin (0,0)
  pto rotate(T ang) {
    return pto(x*cosl(ang) - y*sinl(ang), x*sinl(ang) + y*cosl(ang));
  }
  // true if this is at the left side of line ab
  bool left(pto a, pto b) { return ((a-*this) ^ (b-*this)) > 0; }
  bool operator<(const pto &b) const {
    return x < b.x-EPS || (abs(x - b.x) <= EPS && y < b.y-EPS);
  }
  bool operator==(pto b){ return abs(x-b.x)<=EPS && abs(y-b.y)<=EPS; }
};
ld angle(pto a, pto o, pto b) {
  pto oa = a-o, ob = b-o;
  return atan2l(oa^ob, oa*ob);
}
ld angle(pto a, pto b) { // smallest angle bewteen a and b
  ld cost = (a*b) / a.norm() / b.norm();
  return acosl(max(ld(-1.), min(ld(1.), cost)));
}
```

3.2   Radial order

```cpp
// radial sort around point O in CCW direction starting from vector v
struct cmp {
  pto o, v;
  cmp(pto no, pto nv) : o(no), v(nv) {}
  bool half(pto p) {
```

```cpp
    assert(!(p.x == 0 && p.y == 0));  // (0,0) isn't well defined
    return (v ^ p) < 0 || ((v ^ p) == 0 && (v * p) < 0);
  }
  bool operator()(pto& p1, pto& p2) {
    return mp(half(p1 - o), T(0)) < mp(half(p2 - o), ((p1 - o) ^ (p2 - o)));
  }
};
```

3.3   Line

```cpp
int sgn(T x) { return x < 0 ? -1 : !!x; }
struct line {
  T a, b, c;   // Ax+By=C
  line() {}
  line(T a_, T b_, T c_) : a(a_), b(b_), c(c_) {}
  // TO DO: check negative C (multiply everything by -1)
  line(pto u, pto v) : a(v.y - u.y), b(u.x - v.x), c(a * u.x + b * u.y) {}
  int side(pto v) { return sgn(a * v.x + b * v.y - c); }
  bool inside(pto v) { return abs(a * v.x + b * v.y - c) <= EPS; }
  bool parallel(line v) { return abs(a * v.b - v.a * b) <= EPS; }
  pto inter(line v) {
    T det = a * v.b - v.a * b;
    if (abs(det) <= EPS) return pto(INF, INF);
    return pto(v.b * c - b * v.c, a * v.c - v.a * c) / det;
  }
};
```

3.4   Segment

```cpp
struct segm {
  pto s, e;
  segm(pto s_, pto e_) : s(s_), e(e_) {}
  pto closest(pto b) {
    pto bs = b - s, es = e - s;
    ld l = es * es;
    if (abs(l) <= EPS) return s;
    ld t = (bs * es) / l;
    if (t < 0.) return s;         // comment for lines
    else if (t > 1.) return e;  // comment for lines
    return s + (es * t);
  }
  bool inside(pto b) { return abs(s.dist(b) + e.dist(b) - s.dist(e)) < EPS; }
  pto inter(segm b) {  // if a and b are collinear, returns one point
    if ((*this).inside(b.s)) return b.s;
    if ((*this).inside(b.e)) return b.e;
    pto in = line(s, e).inter(line(b.s, b.e));
    if ((*this).inside(in) && b.inside(in)) return in;
    return pto(INF, INF);
  }
};
```

3.5   Circle

```
1   #define sqr(a) ((a)*(a))
2   pto perp(pto a){return pto(-a.y, a.x);}
3   line bisector(pto a, pto b){
4     line l = line(a, b); pto m = (a+b)/2;
5     return line(-l.b, l.a, -l.b*m.x+l.a*m.y);
6   }
7   struct circle{
8     pto o; T r;
9     circle(){}
10    circle(pto a, pto b, pto c) {
11      o = bisector(a, b).inter(bisector(b, c));
12      r = o.dist(a);
13    }
14    bool inside(pto p) { return (o-p).norm_sq() <= r*r+EPS; }
15    bool inside(circle c) { // this inside of c
16      T d = (o - c.o).norm_sq();
17      return d <= (c.r-r) * (c.r-r) + EPS;
18    }
19    // circle containing p1 and p2 with radius r
20    // swap p1, p2 to get snd solution
21    circle* circle2PtoR(pto a, pto b, T r_) {
22          ld d2 = (a-b).norm_sq(), det = r_*r_/d2 - ld(0.25);
23          if(det < 0) return nullptr;
24      circle *ret = new circle();
25          ret->o = (a+b)/ld(2) + perp(b-a)*sqrt(det);
26          ret->r = r_;
27      return ret;
28    }
29    pair<pto, pto> tang(pto p) {
30      pto m = (p+o)/2;
31      ld d = o.dist(m);
32      ld a = r*r/(2*d);
33      ld h = sqrtl(r*r - a*a);
34      pto m2 = o + (m-o)*a/d;
35      pto per = perp(m-o)/d;
36      return make_pair(m2 - per*h, m2 + per*h);
37    }
38    vector<pto> inter(line l) {
39      ld a = l.a, b = l.b, c = l.c - l.a*o.x - l.b*o.y;
40      pto xy0 = pto(a*c/(a*a + b*b), b*c/(a*a + b*b));
41      if(c*c > r*r*(a*a + b*b) + EPS) {
42        return {};
43      }else if(abs(c*c - r*r*(a*a + b*b)) < EPS) {
44        return { xy0 + o };
45      }else{
46        ld m = sqrtl((r*r - c*c/(a*a + b*b))/(a*a + b*b));
47        pto p1 = xy0 + (pto(-b,a)*m);
48        pto p2 = xy0 + (pto(b,-a)*m);
49        return { p1 + o, p2 + o };
50      }
51    }
52    vector<pto> inter(circle c) {
53      line l;
54      l.a = o.x - c.o.x;
55      l.b = o.y - c.o.y;
56      l.c = (sqr(c.r)-sqr(r)+sqr(o.x)-sqr(c.o.x)+sqr(o.y)-sqr(c.o.y))/2.0;
57      return (*this).inter(l);
58    }
59    ld inter_triangle(pto a, pto b) { // area of intersection with oab
60      if(abs((o-a)^(o-b)) <= EPS) return 0.;
61      vector<pto> q = {a}, w = inter(line(a,b));
62      if(sz(w) == 2) forn(i,sz(w)) if((a-w[i])*(b-w[i]) < -EPS) q.pb(w[i]);
63      q.pb(b);
64      if(sz(q) == 4 && (q[0]-q[1])*(q[2]-q[1]) > EPS) swap(q[1], q[2]);
65      ld s = 0;
66      forn(i, sz(q)-1){
67        if(!inside(q[i]) || !inside(q[i+1])) {
68          s += r*r*angle((q[i]-o),q[i+1]-o)/T(2);
69        }
70        else s += abs((q[i]-o)^(q[i+1]-o)/2);
71      }
72      return s;
73    }
74  };
75  vector<ld> inter_circles(vector<circle> c){
76    vector<ld> r(sz(c)+1); // r[k]: area covered by at least k circles
77    forn(i, sz(c)) {        // O(n^2 log n) (high constant)
78      int k = 1;
79      cmp s(c[i].o, pto(1,0));
80      vector<pair<pto,int>> p = {
81        {c[i].o + pto(1,0)*c[i].r, 0},
82        {c[i].o - pto(1,0)*c[i].r, 0}};
83      forn(j, sz(c)) if(j != i) {
84        bool b0 = c[i].inside(c[j]), b1 = c[j].inside(c[i]);
85        if(b0 && (!b1 || i<j)) k++;
86        else if(!b0 && !b1) {
87          vector<pto> v = c[i].inter(c[j]);
88          if(sz(v) == 2) {
89            p.pb({v[0], 1}); p.pb({v[1], -1});
90            if(s(v[1], v[0])) k++;
91          }
92        }
93      }
94      sort(p.begin(), p.end(), [&](pair<pto,int> a, pair<pto,int> b) {
95        return s(a.fst,b.fst); });
96      forn(j,sz(p)) {
97        pto p0 = p[j? j-1: sz(p)-1].fst, p1 = p[j].fst;
98        ld a = angle(p0 - c[i].o, p1 - c[i].o);
99        r[k]+=(p0.x-p1.x)*(p0.y+p1.y)/ld(2)+c[i].r*c[i].r*(a-sinl(a))/ld(2);
100       k += p[j].snd;
101     }
102   }
103   return r;
104 }
```

3.6   Polygon

```
1  struct poly{
2    vector<pto> pt;
3    poly(){}
4    poly(vector<pto> pt_) : pt(pt_) {}
5    void delete_collinears() { // delete collinear points
6      deque<pto> nxt; int len = 0;
7      forn(i,sz(pt)) {
8        if(len>1 && abs((pt[i]-nxt[len-2])^(nxt[len-1]-nxt[len-2])) <= EPS)
9          nxt.pop_back(), len--;
10       nxt.pb(pt[i]); len++;
11     }
12     if(len>2 && abs((nxt[1]-nxt[len-1])^(nxt[0]-nxt[len-1])) <= EPS)
13       nxt.pop_front(), len--;
14     if(len>2 && abs((nxt[len-1]-nxt[len-2])^(nxt[0]-nxt[len-2])) <= EPS)
15       nxt.pop_back(), len--;
16     pt.clear(); forn(i,sz(nxt)) pt.pb(nxt[i]);
17   }
18   void normalize() {
19     delete_collinears();
20     if(pt[2].left(pt[0], pt[1])) reverse(pt.begin(), pt.end()); //make it CW
21     int n = sz(pt), pi = 0;
22     forn(i, n)
23       if(pt[i].x<pt[pi].x || (pt[i].x==pt[pi].x && pt[i].y<pt[pi].y))
24         pi = i;
25     rotate(pt.begin(), pt.begin()+pi, pt.end());
26   }
27   bool is_convex() { // delete collinear points first
28     int N = sz(pt);
29     if(N < 3) return false;
30     bool isLeft = pt[0].left(pt[1], pt[2]);
31     forr(i, 1, sz(pt))
32       if(pt[i].left(pt[(i+1)%N], pt[(i+2)%N]) != isLeft)
33         return false;
34     return true;
35   }
36   // for convex or concave polygons
37   // excludes boundaries, check it manually
38   bool inside(pto p) { // O(n)
39     bool c = false;
40     forn(i, sz(pt)) {
41       int j = (i+1)%sz(pt);
42       if((pt[j].y>p.y) != (pt[i].y > p.y) &&
43       (p.x < (pt[i].x-pt[j].x)*(p.y-pt[j].y)/(pt[i].y-pt[j].y)+pt[j].x))
44         c = !c;
45     }
46     return c;
47   }
48   bool inside_convex(pto p) { // O(lg(n)) normalize first
49     if(p.left(pt[0], pt[1]) || p.left(pt[sz(pt)-1], pt[0])) return false;
50     int a = 1, b = sz(pt)-1;
51     while(b-a > 1){
52       int c = (a+b)/2;
53       if(!p.left(pt[0], pt[c])) a = c;
54       else b = c;
55     }
56     return !p.left(pt[a], pt[a+1]);
57   }
58   // cuts this along line ab and return the left side
59   // (swap a, b for the right one)
60   poly cut(pto a, pto b) { // O(n)
61     vector<pto> ret;
62     forn(i, sz(pt)) {
63       ld left1 = (b-a)^(pt[i]-a), left2 = (b-a)^(pt[(i+1)%sz(pt)]-a);
64       if(left1 >= 0) ret.pb(pt[i]);
65       if(left1*left2 < 0)
66         ret.pb(line(pt[i], pt[(i+1)%sz(pt)]).inter(line(a, b)));
67     }
68     return poly(ret);
69   }
70   // cuts this with line ab and returns the range [from, to] that is
71   // strictly on the left side (note that indexes are circular)
72   ii cut(pto u, pto v) { // O(log(n)) for convex polygons
73     int n = sz(pt); pto dir = v-u;
74     int L = farthest(pto(dir.y,-dir.x));
75     int R = farthest(pto(-dir.y,dir.x));
76     if(!pt[L].left(u,v)) swap(L,R);
77     if(!pt[L].left(u,v)) return mp(-1,-1); // line doesn't cut the poly
78
79     ii ans;
80     int l = L, r = L > R ? R+n : R;
81     while(l<r) {
82       int med = (l+r+1)/2;
83       if(pt[med >= n ? med-n : med].left(u,v)) l = med;
84       else r = med-1;
85     }
86     ans.snd = l >= n ? l-n : l;
87
88     l = R, r = L < R ? L+n : L;
89     while(l<r) {
90       int med = (l+r)/2;
91       if(!pt[med >= n ? med-n : med].left(u,v)) l = med+1;
92       else r = med;
93     }
94     ans.fst = l >= n ? l-n : l;
95
96     return ans;
97   }
98   // addition of convex polygons
99   poly minkowski(poly p) { // O(n+m) n=|this|,m=|p|
100    this->normalize(); p.normalize();
101    vector<pto> a = (*this).pt, b = p.pt;
102    a.pb(a[0]); a.pb(a[1]);
103    b.pb(b[0]); b.pb(b[1]);
104    vector<pto> sum;
105    int i = 0, j = 0;
106    while(i < sz(a)-2 || j < sz(b)-2) {
107      sum.pb(a[i]+b[j]);
108      T cross = (a[i+1]-a[i])^(b[j+1]-b[j]);
109      if(cross <= 0 && i < sz(a)-2) i++;
110      if(cross >= 0 && j < sz(b)-2) j++;
```

```
111        }
112      return poly(sum);
113    }
114    pto farthest(pto v) { // O(log(n)) for convex polygons
115      if(sz(pt)<10) {
116        int k=0;
117        forr(i,1,sz(pt)) if(v * (pt[i] - pt[k]) > EPS) k = i;
118        return pt[k];
119      }
120      pt.pb(pt[0]);
121      pto a=pt[1] - pt[0];
122      int s = 0, e = sz(pt)-1, ua = v*a > EPS;
123      if(!ua && v*(pt[sz(pt)-2]-pt[0]) <= EPS){ pt.pop_back(); return pt[0];}
124      while(1) {
125        int m = (s+e)/2; pto c = pt[m+1]-pt[m];
126        int uc = v*c > EPS;
127        if(!uc && v*(pt[m-1]-pt[m]) <= EPS){ pt.pop_back(); return pt[m];}
128        if(ua && (!uc || v*(pt[s]-pt[m]) > EPS)) e = m;
129        else if(ua || uc || v*(pt[s]-pt[m]) >= -EPS) s = m, a = c, ua = uc;
130        else e = m;
131        assert(e > s+1);
132      }
133    }
134    ld inter_circle(circle c){ // area of intersection with circle
135      ld r = 0.;
136      forn(i,sz(pt)) {
137        int j = (i+1)%sz(pt); ld w = c.inter_triangle(pt[i], pt[j]);
138        if(((pt[j]-c.o)^(pt[i]-c.o)) > 0) r += w;
139        else r -= w;
140      }
141      return abs(r);
142    }
143    // area ellipse = M_PI*a*b where a and b are the semi axis lengths
144    // area triangle = sqrt(s*(s-a)(s-b)(s-c)) where s=(a+b+c)/2
145    ld area(){ // O(n)
146      ld area = 0;
147      forn(i, sz(pt)) area += pt[i]^pt[(i+1)%sz(pt)];
148      return abs(area)/ld(2);
149    }
150    // returns one pair of most distant points
151    pair<pto,pto> callipers() { // O(n), for convex  poly, normalize first
152      int n = sz(pt);
153      if(n <= 2) return {pt[0], pt[1%n]};
154      pair<pto,pto> ret = {pt[0], pt[1]};
155      T maxi = 0; int j = 1;
156      forn(i,sz(pt)) {
157        while(((pt[(i+1)%n]-pt[i])^(pt[(j+1)%n]-pt[j]))<-EPS)j=(j+1)%sz(pt);
158        if(pt[i].dist(pt[j]) > maxi+EPS)
159          ret = {pt[i], pt[j]}, maxi = pt[i].dist(pt[j]);
160      }
161      return ret;
162    }
163    pto centroid(){ // (barycenter, mass center, needs float points)
164      int n = sz(pt);
165      pto r(0,0); ld t=0;
166      forn(i,n) {
167        r = r + (pt[i] + pt[(i+1)%n]) * (pt[i] ^ pt[(i+1)%n]);
168        t += pt[i] ^ pt[(i+1)%n];
169      }
170      return r/t/3;
171    }
172 };
173 // Dynamic convex hull trick (based on poly struct)
174 vector<poly> w;
175 void add(pto q) { // add(q), O(log^2(n))
176   vector<pto> p = {q};
177   while(!w.empty() && sz(w.back().pt) < 2*sz(p)){
178     for(pto v : w.back().pt) p.pb(v);
179     w.pop_back();
180   }
181   w.pb(poly(CH(p))); // CH = convex hull, must delete collinears
182 }
183 T query(pto v) { // max(q*v:q in w), O(log^2(n))
184   T r = -INF;
185   for(auto& p : w) r = max(r, p.farthest(v)*v);
186   return r;
187 }
```

3.7   Halfplane

```
1  struct halfplane {  // left half plane
2    pto u, uv;
3    int id;
4    ld angle;
5    halfplane() {}
6    halfplane(pto u_, pto v_) : u(u_), uv(v_ - u_), angle(atan2l(uv.y, uv.x)) {}
7    bool operator<(halfplane h) const { return angle < h.angle; }
8    bool out(pto p) { return (uv ^ (p - u)) < -EPS; }
9    pto inter(halfplane& h) {
10     T alpha = ((h.u - u) ^ h.uv) / (uv ^ h.uv);
11     return u + (uv * alpha);
12   }
13 };
14 vector<pto> intersect(vector<halfplane> h) {
15   pto box[4] = {{INF, INF}, {-INF, INF}, {-INF, -INF}, {INF, -INF}};
16   forn(i, 4) h.pb(halfplane(box[i], box[(i + 1) % 4]));
17   sort(h.begin(), h.end());
18   deque<halfplane> dq;
19   int len = 0;
20   forn(i, sz(h)) {
21     while (len > 1 && h[i].out(dq[len - 1].inter(dq[len - 2]))) {
22       dq.pop_back(), len--;
23     }
24     while (len > 1 && h[i].out(dq[0].inter(dq[1]))) { dq.pop_front(), len--; }
25     if (len > 0 && abs(h[i].uv ^ dq[len - 1].uv) <= EPS) {
26       if (h[i].uv * dq[len - 1].uv < 0.) { return vector<pto>(); }
27       if (h[i].out(dq[len - 1].u)) {
28         dq.pop_back(), len--;
29       } else continue;
30     }
31     dq.pb(h[i]);
32     len++;
33   }
34   while (len > 2 && dq[0].out(dq[len - 1].inter(dq[len - 2]))) {
35     dq.pop_back(), len--;
36   }
37   while (len > 2 && dq[len - 1].out(dq[0].inter(dq[1]))) {
38     dq.pop_front(), len--;
39   }
40   if (len < 3) return vector<pto>();
41   vector<pto> inter;
42   forn(i, len) inter.pb(dq[i].inter(dq[(i + 1) % len]));
43   return inter;
44 }
```

3.8   Convex Hull

```
1  // returns convex hull of p in CCW order
2  // left must return >=0 to delete collinear points
3  vector<pto> CH(vector<pto>& p) {
4    if (sz(p) < 3) return p;  // edge case, keep line or point
5    vector<pto> ch;
6    sort(p.begin(), p.end());
7    forn(i, sz(p)) {  // lower hull
8      while (sz(ch) >= 2 && ch[sz(ch) - 1].left(ch[sz(ch) - 2], p[i]))
9        ch.pop_back();
10     ch.pb(p[i]);
11   }
12   ch.pop_back();
13   int k = sz(ch);
14   dforn(i, sz(p)) {  // upper hull
15     while (sz(ch) >= k + 2 && ch[sz(ch) - 1].left(ch[sz(ch) - 2], p[i]))
16       ch.pop_back();
17     ch.pb(p[i]);
18   }
19   ch.pop_back();
20   return ch;
21 }
```

3.9   Dynamic Convex Hull

```
1  struct semi_chull {
2    set<pto> pt;  // maintains semi chull without collinears points
3    // in case we want them on the set, make the changes commented below
4    bool check(pto p) {
5      if (pt.empty()) return false;
6      if (*pt.rbegin() < p) return false;
7      if (p < *pt.begin()) return false;
8      auto it = pt.lower_bound(p);
9      if (it->x == p.x) return p.y <= it->y;  // change? for collinears
10     pto b = *it;
11     pto a = *prev(it);
12     return ((b - p) ^ (a - p)) + EPS >= 0;  // change? for collinears
13   }
14   void add(pto p) {
15     if (check(p)) return;
16     pt.erase(p);
17     pt.insert(p);
18     auto it = pt.find(p);
19     while (true) {
20       if (next(it) == pt.end() || next(next(it)) == pt.end()) break;
21       pto a = *next(it), b = *next(next(it));
22       if (((b - a) ^ (p - a)) + EPS >= 0) {  // change? for collinears
23         pt.erase(next(it));
24       } else break;
25     }
26     it = pt.find(p);
27     while (true) {
```

```
28      if (it == pt.begin() || prev(it) == pt.begin()) break;
29      pto a = *prev(it), b = *prev(prev(it));
30      if (((b - a) ^ (p - a)) - EPS <= 0) {  // change? for collinears
31        pt.erase(prev(it));
32      } else break;
33    }
34  }
35 };
36 struct CHD {
37   semi_chull sup, inf;
38   void add(pto p) { sup.add(p), inf.add(p * (-1)); }
39   bool check(pto p) { return sup.check(p) && inf.check(p * (-1)); }
40 };
```

3.10   Convex Hull Trick

```
1  struct CHT {
2    deque<pto> h;
3    T f = 1, pos;
4    CHT(bool min_ = 0) : f(min_ ? 1 : -1), pos(0) {}  // min_=1 for min queries
5    void add(pto p) {  // O(1), pto(m,b) <=> y = mx + b
6      p = p * f;
7      if (h.empty()) {
8        h.pb(p);
9        return;
10     }
11     // p.x should be the lower/greater hull x
12     assert(p.x <= h[0].x || p.x >= h.back().x);
13     if (p.x <= h[0].x) {
14       while (sz(h) > 1 && h[0].left(p, h[1])) h.pop_front(), pos--;
15       h.push_front(p), pos++;
16     } else {
17       while (sz(h) > 1 && h[sz(h) - 1].left(h[sz(h) - 2], p)) h.pop_back();
18       h.pb(p);
19     }
20     pos = min(max(T(0), pos), T(sz(h) - 1));
21   }
22   T get(T x) {
23     pto q = {x, 1};
24     // O(log) query for unordered x
25     int L = 0, R = sz(h) - 1, M;
26     while (L < R) {
27       M = (L + R) / 2;
28       if (h[M + 1] * q <= h[M] * q) L = M + 1;
29       else R = M;
30     }
31     return h[L] * q * f;
32     // O(1) query for ordered x
33     while (pos > 0 && h[pos - 1] * q < h[pos] * q) pos--;
34     while (pos < sz(h) - 1 && h[pos + 1] * q < h[pos] * q) pos++;
35     return h[pos] * q * f;
36   }
37 };
```

3.11   Li-Chao tree

```
1  typedef long long T;
2  const T INF = 1e18;
3  // Li-Chao works for any function such that any pair of the functions
4  // inserted intersect at most once with each other. Most problems are
5  // about lines, but you may want to adapt this struct to your function
6  struct line {
7    T m, b;
8    line() {}
9    line(T m_, T b_) {
10     m = m_;
11     b = b_;
12   }
13   T f(T x) { return m * x + b; }
14   line operator+(line l) { return line(m + l.m, b + l.b); }
15   line operator*(T k) { return line(m * k, b * k); }
16 };
17 struct li_chao {
18   vector<line> cur, add;
19   vector<int> L, R;
20   T f, minx, maxx;
21   line identity;
22   int cnt;
23   void new_node(line cur_, int l = -1, int r = -1) {
24     cur.pb(cur_);
25     add.pb(line(0, 0));
26     L.pb(l);
27     R.pb(r);
28     cnt++;
29   }
30   li_chao(bool min_, T minx_, T maxx_) {  // for min: min_=1, for max: min_=0
31     f = min_ ? 1 : -1;
32     identity = line(0, INF);
33     minx = minx_;
34     maxx = maxx_;
35     cnt = 0;
36     new_node(identity);  // root id is 0
37   }
38   // only needed when "adding" lines lazily
39   void apply(int id, line to_add_) {
40     add[id] = add[id] + to_add_;
41     cur[id] = cur[id] + to_add_;
42   }
43   // this method is needed even when no lazy is used, to avoid
44   // null pointers and other problems in the code
45   void push_lazy(int id) {
46     if (L[id] == -1) {
47       new_node(identity);
48       L[id] = cnt - 1;
49     }
50     if (R[id] == -1) {
51       new_node(identity);
52       R[id] = cnt - 1;
53     }
54     // code below only needed when lazy ops are needed
```

```cpp
    apply(L[id], add[id]);
    apply(R[id], add[id]);
    add[id] = line(0, 0);
  }
  // only needed when "adding" lines lazily
  void push_line(int id, T tl, T tr) {
    T m = (tl + tr) / 2;
    insert_line(L[id], cur[id], tl, m);
    insert_line(R[id], cur[id], m, tr);
    cur[id] = identity;
  }
  // O(log), or persistent return int instead of void
  void insert_line(int id, line new_line, T l, T r) {
    T m = (l + r) / 2;
    bool lef = new_line.f(l) < cur[id].f(l);
    bool mid = new_line.f(m) < cur[id].f(m);
    // uncomment for persistent
    // line to_push = new_line, to_keep = cur[id];
    // if(mid) swap(to_push,to_keep);
    if (mid) swap(new_line, cur[id]);

    if (r - l == 1) {
      // uncomment for persistent
      // new_node(to_keep);
      // return cnt-1;
      return;
    }
    push_lazy(id);
    if (lef != mid) {
      // uncomment for persistent
      // int lid = insert_line(L[id],to_push, l, m);
      // new_node(to_keep, lid, R[id]);
      // return cnt-1;
      insert_line(L[id], new_line, l, m);
    } else {
      // uncomment for persistent
      // int rid = insert_line(R[id],to_push, m, r);
      // new_node(to_keep, L[id], rid);
      // return cnt-1;
      insert_line(R[id], new_line, m, r);
    }
  }
  // for persistent, return int instead of void
  void insert_line(int id, line new_line) {
    insert_line(id, new_line * f, minx, maxx);
  }
  // O(log^2) doesn't support persistence
  void insert_segm(int id, line new_line, T l, T r, T tl, T tr) {
    if (tr <= l || tl >= r || tl >= tr || l >= r) return;
    if (tl >= l && tr <= r) {
      insert_line(id, new_line, tl, tr);
      return;
    }
    push_lazy(id);
    T m = (tl + tr) / 2;
    insert_segm(L[id], new_line, l, r, tl, m);
    insert_segm(R[id], new_line, l, r, m, tr);
  }
  // [l,r)
  void insert_segm(int id, line new_line, T l, T r) {
    insert_segm(id, new_line * f, l, r, minx, maxx);
  }
  // O(log^2) doesn't support persistence
  void add_line(int id, line to_add_, T l, T r, T tl, T tr) {
    if (tr <= l || tl >= r || tl >= tr || l >= r) return;
    if (tl >= l && tr <= r) {
      apply(id, to_add_);
      return;
    }
    push_lazy(id);
    push_line(id, tl, tr);  // comment if insert isn't used
    T m = (tl + tr) / 2;
    add_line(L[id], to_add_, l, r, tl, m);
    add_line(R[id], to_add_, l, r, m, tr);
  }
  void add_line(int id, line to_add_, T l, T r) {
    add_line(id, to_add_ * f, l, r, minx, maxx);
  }
  // O(log)
  T get(int id, T x, T tl, T tr) {
    if (tl + 1 == tr) return cur[id].f(x);
    push_lazy(id);
    T m = (tl + tr) / 2;
    if (x < m) return min(cur[id].f(x), get(L[id], x, tl, m));
    else return min(cur[id].f(x), get(R[id], x, m, tr));
  }
  T get(int id, T x) { return get(id, x, minx, maxx) * f; }
};
```

3.12   KD tree

```cpp
bool cmpx(pto a, pto b) { return a.x + EPS < b.x; }
bool cmpy(pto a, pto b) { return a.y + EPS < b.y; }
struct kd_tree {
  pto p;
  T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF;
  kd_tree *l, *r;
  T distance(pto q) {
    T x = min(max(x0, q.x), x1);
    T y = min(max(y0, q.y), y1);
    return (pto(x, y) - q).norm_sq();
  }
  kd_tree(vector<pto>&& pts) : p(pts[0]) {
    l = nullptr, r = nullptr;
    forn(i, sz(pts)) {
      x0 = min(x0, pts[i].x), x1 = max(x1, pts[i].x);
      y0 = min(y0, pts[i].y), y1 = max(y1, pts[i].y);
    }
    if (sz(pts) > 1) {
      sort(pts.begin(), pts.end(), x1 - x0 >= y1 - y0 ? cmpx : cmpy);
      int m = sz(pts) / 2;
      l = new kd_tree({pts.begin(), pts.begin() + m});
      r = new kd_tree({pts.begin() + m, pts.end()});
    }
  }
  void nearest(pto q, int k, priority_queue<pair<T, pto>>& ret) {
    if (l == nullptr) {
      // avoid query point as answer
      // if(p == q) return;
      ret.push({(q - p).norm_sq(), p});
      while (sz(ret) > k) ret.pop();
      return;
    }
    kd_tree *al = l, *ar = r;
    T bl = l->distance(q), br = r->distance(q);
    if (bl > br) swap(al, ar), swap(bl, br);
    al->nearest(q, k, ret);
    if (br < ret.top().fst) ar->nearest(q, k, ret);
    while (sz(ret) > k) ret.pop();
  }
  priority_queue<pair<T, pto>> nearest(pto q, int k) {
    priority_queue<pair<T, pto>> ret;
    forn(i, k) ret.push({INF * INF, pto(INF, INF)});
    nearest(q, k, ret);
    return ret;
  }
};
```

3.13   Voronoi

```cpp
// Returns planar graph representing Delaunay's triangulation.
// Edges for each vertex are in ccw order.
// To use doubles replace __int128 for long double in line 51
pto pinf = pto(INF, INF);
typedef struct QuadEdge* Q;
struct QuadEdge {
  int id, used;
  pto o;
  Q rot, nxt;
  QuadEdge(int id_ = -1, pto o_ = pinf)
      : id(id_), used(0), o(o_), rot(0), nxt(0) {}
  Q rev() { return rot->rot; }
  Q next() { return nxt; }
  Q prev() { return rot->next()->rot; }
  pto dest() { return rev()->o; }
};

Q edge(pto a, pto b, int ida, int idb) {
  Q e1 = new QuadEdge(ida, a);
  Q e2 = new QuadEdge(idb, b);
  Q e3 = new QuadEdge;
  Q e4 = new QuadEdge;
  tie(e1->rot, e2->rot, e3->rot, e4->rot) = {e3, e4, e2, e1};
  tie(e1->nxt, e2->nxt, e3->nxt, e4->nxt) = {e1, e2, e4, e3};
  return e1;
}

void splice(Q a, Q b) {
  swap(a->nxt->rot->nxt, b->nxt->rot->nxt);
  swap(a->nxt, b->nxt);
}

void del_edge(Q& e, Q ne) {
  splice(e, e->prev());
  splice(e->rev(), e->rev()->prev());
  delete e->rev()->rot;
  delete e->rev();
  delete e->rot;
  delete e;
  e = ne;
}

Q conn(Q a, Q b) {
  Q e = edge(a->dest(), b->o, a->rev()->id, b->id);
  splice(e, a->rev()->prev());
  splice(e->rev(), b);
  return e;
}

auto area(pto p, pto q, pto r) { return (q - p) ^ (r - q); }

// is p in circumference formed by (a,b,c)?
bool in_c(pto a, pto b, pto c, pto p) {
  // Warning: this number is O(max_coord^4).
```

```
55    // Consider using doubles or an alternative method for this function
56    __int128 p2 = p * p, A = a * a - p2, B = b * b - p2, C = c * c - p2;
57    return area(p, a, b) * C + area(p, b, c) * A + area(p, c, a) * B > EPS;
58  }
59
60  pair<Q, Q> build_tr(vector<pto>& p, int l, int r) {
61    if (r - l + 1 <= 3) {
62      Q a = edge(p[l], p[l + 1], l, l + 1), b = edge(p[l + 1], p[r], l + 1, r);
63      if (r - l + 1 == 2) return {a, a->rev()};
64      splice(a->rev(), b);
65      auto ar = area(p[l], p[l + 1], p[r]);
66      Q c = abs(ar) > EPS ? conn(b, a) : 0;
67      if (ar >= -EPS) return {a, b->rev()};
68      return {c->rev(), c};
69    }
70    int m = (l + r) / 2;
71    Q la, ra, lb, rb;
72    tie(la, ra) = build_tr(p, l, m);
73    tie(lb, rb) = build_tr(p, m + 1, r);
74    while (1) {
75      if (ra->dest().left(lb->o, ra->o)) ra = ra->rev()->prev();
76      else if (lb->dest().left(lb->o, ra->o)) lb = lb->rev()->next();
77      else break;
78    }
79    Q b = conn(lb->rev(), ra);
80    auto valid = [&](Q e) { return b->o.left(e->dest(), b->dest()); };
81    if (ra->o == la->o) la = b->rev();
82    if (lb->o == rb->o) rb = b;
83    while (1) {
84      Q L = b->rev()->next();
85      if (valid(L))
86        while (in_c(b->dest(), b->o, L->dest(), L->next()->dest()))
87          del_edge(L, L->next());
88      Q R = b->prev();
89      if (valid(R))
90        while (in_c(b->dest(), b->o, R->dest(), R->prev()->dest()))
91          del_edge(R, R->prev());
92      if (!valid(L) && !valid(R)) break;
93      if (!valid(L) || (valid(R) && in_c(L->dest(), L->o, R->o, R->dest())))
94        b = conn(R, b->rev());
95      else b = conn(b->rev(), L->rev());
96    }
97    return {la, rb};
98  }
99
100 vector<vector<int>> delaunay(vector<pto> v) {
101   int n = sz(v);
102   auto tmp = v;
103   vector<int> id(n);
104   iota(id.begin(), id.end(), 0);
105   sort(id.begin(), id.end(), [&](int l, int r) { return v[l] < v[r]; });
106   forn(i, n) v[i] = tmp[id[i]];
107   assert(unique(v.begin(), v.end()) == v.end());
108   vector<vector<int>> g(n);
109   int col = 1;
110   forr(i, 2, n) col &= abs(area(v[i], v[i - 1], v[i - 2])) <= EPS;
```

```
111   if (col) {
112     forr(i, 1, n) g[id[i - 1]].pb(id[i]), g[id[i]].pb(id[i - 1]);
113   } else {
114     Q e = build_tr(v, 0, n - 1).fst;
115     vector<Q> edg = {e};
116     for (int i = 0; i < sz(edg); e = edg[i++]) {
117       for (Q at = e; !at->used; at = at->next()) {
118         at->used = 1;
119         g[id[at->id]].pb(id[at->rev()->id]);
120         edg.pb(at->rev());
121       }
122     }
123   }
124   return g;
125 }
```

3.14   All point pairs

```
1  // after each step() execution pt is sorted by dot product of the event
2  struct all_point_pairs {  // O(n*n*log(n*n)), must add id, u, v to pto
3    vector<pto> pt, ev;
4    vector<int> idx;
5    int cur_step;
6    all_point_pairs(vector<pto> pt_) : pt(pt_) {
7      idx = vector<int>(sz(pt));
8      forn(i, sz(pt)) forn(j, sz(pt)) if (i != j) {
9        pto p = pt[j] - pt[i];
10       p.u = pt[i].id, p.v = pt[j].id;
11       ev.pb(p);
12     }
13     sort(ev.begin(), ev.end(), cmp(pto(0, 0), pto(1, 0)));
14     pto start(ev[0].y, -ev[0].x);
15     sort(pt.begin(), pt.end(),
16          [&](pto& u, pto& v) { return u * start < v * start; });
17     forn(i, sz(idx)) idx[pt[i].id] = i;
18     cur_step = 0;
19   }
20   bool step() {
21     if (cur_step >= sz(ev)) return false;
22     int u = ev[cur_step].u, v = ev[cur_step].v;
23     swap(pt[idx[u]], pt[idx[v]]);
24     swap(idx[u], idx[v]);
25     cur_step++;
26     return true;
27   }
28 };
```

4   Data structures

4.1   Indexed set

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
//<key,mapped type,comparator,...>
typedef tree<int,null_type,less<int>,rb_tree_tag,
  tree_order_statistics_node_update> indexed_set;
// find_by_order(i) returns iterator to the i-th elemnt
// order_of_key(k): returns position of the lower bound of k (0-indexed)
// Ej: 12, 100, 505, 1000, 10000.
// order_of_key(10) == 0, order_of_key(100) == 1,
// order_of_key(707) == 3, order_of_key(9999999) == 5
```

4.2   Hash Table

```cpp
struct Hash {  // similar logic for any other data type
  size_t operator()(const vector<int>& v) const {
    size_t s = 0;
    for (auto& e : v) s ^= hash<int>()(e) + 0x9e3779b9 + (s << 6) + (s >> 2);
    return s;
  }
};
unordered_set<vector<int>, Hash> s;  // unordered_map<key, value, hasher>
```

4.3   Union find

4.3.1   Classic DSU

```cpp
struct UnionFind {
  int nsets;
  vector<int> f, setsz;  // f[i] = parent of node i
  UnionFind(int n) : nsets(n), f(n, -1), setsz(n, 1) {}
  int comp(int x) { return (f[x] == -1 ? x : f[x] = comp(f[x])); }  // O(1)
  bool join(int i, int j) {  // returns true if already in the same set
    int a = comp(i), b = comp(j);
    if (a != b) {
      if (setsz[a] > setsz[b]) swap(a, b);
      f[a] = b;  // the bigger group (b) now represents the smaller (a)
      nsets--, setsz[b] += setsz[a];
    }
    return a == b;
  }
};
```

4.3.2   DSU with rollbacks

```cpp
struct dsu_save {
  int v, rnkv, u, rnku;
  dsu_save() {}
  dsu_save(int _v, int _rnkv, int _u, int _rnku)
      : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};
struct dsu_with_rollbacks {
  vector<int> p, rnk;
  int comps;
  stack<dsu_save> op;
  dsu_with_rollbacks() {}
  dsu_with_rollbacks(int n) {
    p.rsz(n), rnk.rsz(n);
    forn(i, n) { p[i] = i, rnk[i] = 0; }
    comps = n;
  }
  int find_set(int v) { return (v == p[v]) ? v : find_set(p[v]); }
  bool unite(int v, int u) {
    v = find_set(v), u = find_set(u);
    if (v == u) return false;
    comps--;
    if (rnk[v] > rnk[u]) swap(v, u);
    op.push(dsu_save(v, rnk[v], u, rnk[u]));
    p[v] = u;
    if (rnk[u] == rnk[v]) rnk[u]++;
    return true;
  }
  void rollback() {
    if (op.empty()) return;
    dsu_save x = op.top();
    op.pop(), comps++;
    p[x.v] = x.v, rnk[x.v] = x.rnkv;
    p[x.u] = x.u, rnk[x.u] = x.rnku;
  }
};
```

4.4   Segment tree

4.4.1   ST static

```
1  // Solo para funciones idempotentes (como min y max, pero no sum)
2  // Usar la version dynamic si la funcion no es idempotente
3  struct RMQ {
4  #define LVL 10  // LVL such that 2^LVL>n
5    tipo vec[LVL][1 << (LVL + 1)];
6    tipo& operator[](int p) { return vec[0][p]; }
7    tipo get(int i, int j) {  // intervalo [i,j) - O(1)
8      int p = 31 - __builtin_clz(j - i);
9      return min(vec[p][i], vec[p][j - (1 << p)]);
10   }
11   void build(int n) {  // O(nlogn)
12     int mp = 31 - __builtin_clz(n);
13     forn(p, mp) forn(x, n - (1 << p)) vec[p + 1][x] =
14        min(vec[p][x], vec[p][x + (1 << p)]);
15   }
16 };  // Use: define LVL y tipo; insert data with []; call build; answer queries
```

4.4.2   ST dynamic

```
1  typedef ll tipo;
2  const tipo neutro = 0;
3  tipo oper(const tipo& a, const tipo& b) { return a + b; }
4  struct ST {
5    int sz;
6    vector<tipo> t;
7    ST(int n) {
8      sz = 1 << (32 - __builtin_clz(n));
9      t = vector<tipo>(2 * sz, neutro);
10   }
11   tipo& operator[](int p) { return t[sz + p]; }
12   void updall() { dforn(i, sz) t[i] = oper(t[2 * i], t[2 * i + 1]); }
13   tipo get(int i, int j) { return get(i, j, 1, 0, sz); }
14   tipo get(int i, int j, int n, int a, int b) {  // O(log n), [i, j)
15     if (j <= a || b <= i) return neutro;
16     if (i <= a && b <= j) return t[n];  // n = node of range [a,b)
17     int c = (a + b) / 2;
18     return oper(get(i, j, 2 * n, a, c), get(i, j, 2 * n + 1, c, b));
19   }
20   void set(int p, tipo val) {  // O(log n)
21     p += sz;
22     while (p > 0 && t[p] != val) {
23       t[p] = val;
24       p /= 2;
25       val = oper(t[p * 2], t[p * 2 + 1]);
26     }
27   }
28 };  // Use: definir oper tipo neutro,
29 // cin >> n; ST st(n); forn(i, n) cin >> st[i]; st.updall();
```

4.4.3   ST lazy

```
1  typedef ll Elem;
2  typedef ll Alt;
3  const Elem neutro = 0;
4  const Alt neutro2 = 0;
5  Elem oper(const Elem& a, const Elem& b) { return a + b; }
6  struct ST {
7    int sz;
8    vector<Elem> t;
9    vector<Alt> dirty;  // Alt and Elem could be different types
10   ST(int n) {
11     sz = 1 << (32 - __builtin_clz(n));
12     t = vector<Elem>(2 * sz, neutro);
13     dirty = vector<Alt>(2 * sz, neutro2);
14   }
15   Elem& operator[](int p) { return t[sz + p]; }
16   void updall() { dforn(i, sz) t[i] = oper(t[2 * i], t[2 * i + 1]); }
17   void push(int n, int a, int b) {  // push dirt to n's child nodes
18     if (dirty[n] != neutro2) {        // n = node of range [a,b)
19       t[n] += dirty[n] * (b - a);    // CHANGE for your problem
20       if (n < sz) {
21         dirty[2 * n] += dirty[n];        // CHANGE for your problem
22         dirty[2 * n + 1] += dirty[n];  // CHANGE for your problem
23       }
24       dirty[n] = neutro2;
25     }
26   }
27   Elem get(int i, int j, int n, int a, int b) {  // O(lgn)
28     if (j <= a || b <= i) return neutro;
29     push(n, a, b);                        // adjust value before using it
30     if (i <= a && b <= j) return t[n];  // n = node of range [a,b)
31     int c = (a + b) / 2;
32     return oper(get(i, j, 2 * n, a, c), get(i, j, 2 * n + 1, c, b));
33   }
34   Elem get(int i, int j) { return get(i, j, 1, 0, sz); }
35   // altera los valores en [i, j) con una alteracion de val
36   void update(Alt val, int i, int j, int n, int a, int b) {  // O(lgn)
37     push(n, a, b);
38     if (j <= a || b <= i) return;
39     if (i <= a && b <= j) {
40       dirty[n] += val;  // CHANGE for your problem
41       push(n, a, b);
42       return;
43     }
44     int c = (a + b) / 2;
45     update(val, i, j, 2 * n, a, c), update(val, i, j, 2 * n + 1, c, b);
46     t[n] = oper(t[2 * n], t[2 * n + 1]);
47   }
48   void update(Alt val, int i, int j) { update(val, i, j, 1, 0, sz); }
49 };  // Use: definir operacion, neutros, Alt, Elem, uso de dirty
50 // cin >> n; ST st(n); forn(i,n) cin >> st[i]; st.updall()
```

4.4.4   ST persistente

```cpp
typedef int tipo;
const tipo neutro = 0;
tipo oper(const tipo& a, const tipo& b) { return a + b; }
struct ST {
  int n;
  vector<tipo> st;
  vector<int> L, R;
  ST(int nn) : n(nn), st(1, neutro), L(1, 0), R(1, 0) {}
  int new_node(tipo v, int l = 0, int r = 0) {
    int id = sz(st);
    st.pb(v), L.pb(l), R.pb(r);
    return id;
  }
  int init(vector<tipo>& v, int l, int r) {
    if (l + 1 == r) return new_node(v[l]);
    int m = (l + r) / 2, a = init(v, l, m), b = init(v, m, r);
    return new_node(oper(st[a], st[b]), a, b);
  }
  int update(int cur, int pos, tipo val, int l, int r) {
    int id = new_node(st[cur], L[cur], R[cur]);
    if (l + 1 == r) {
      st[id] = val;
      return id;
    }
    int m = (l + r) / 2, ASD;  // MUST USE THE ASD!!!
    if (pos < m) ASD = update(L[id], pos, val, l, m), L[id] = ASD;
    else ASD = update(R[id], pos, val, m, r), R[id] = ASD;
    st[id] = oper(st[L[id]], st[R[id]]);
    return id;
  }
  tipo get(int cur, int from, int to, int l, int r) {
    if (to <= l || r <= from) return neutro;
    if (from <= l && r <= to) return st[cur];
    int m = (l + r) / 2;
    return oper(get(L[cur], from, to, l, m), get(R[cur], from, to, m, r));
  }
  int init(vector<tipo>& v) { return init(v, 0, n); }
  int update(int root, int pos, tipo val) {
    return update(root, pos, val, 0, n);
  }
  tipo get(int root, int from, int to) { return get(root, from, to, 0, n); }
};  // usage: ST st(n); root = st.init(v) (or root = 0);
// new_root = st.update(root,i,x); st.get(root,l,r);
```

4.4.5   ST implicit

```cpp
typedef int tipo;
const tipo neutro = 0;
tipo oper(const tipo& a, const tipo& b) { return a + b; }
// Compressed segtree, it works for any range (even negative indexes)
struct ST {
  ST *lc, *rc;
  tipo val;
  int L, R;
  ST(int l, int r, tipo x = neutro) {
    lc = rc = nullptr;
    L = l, R = r, val = x;
  }
  ST(int l, int r, ST* lp, ST* rp) {
    if (lp != nullptr && rp != nullptr && lp->L > rp->L) swap(lp, rp);
    lc = lp, rc = rp;
    L = l, R = r;
    val = oper(lp == nullptr ? neutro : lp->val,
               rp == nullptr ? neutro : rp->val);
  }
  // O(log(R-L)), parameter 'isnew' only needed when persistent
  // This operation inserts at most 2 nodes to the tree, i.e. the
  // total memory used by the tree is O(N), where N is the number
  // of times this 'set' function is called. (2*log nodes when persistent)
  void set(int p, tipo x, bool isnew = false) {  // return ST* for persistent
    // uncomment for persistent
    // if(!isnew) {
    //   ST* newnode = new ST(L, R, lc, rc);
    //   return newnode->set(p, x, true);
    // }
    if (L + 1 == R) {
      val = x;
      return;  // 'return this;' for persistent
    }
    int m = (L + R) / 2;
    ST** c = p < m ? &lc : &rc;
    if (!*c) *c = new ST(p, p + 1, x);
    else if ((*c)->L <= p && p < (*c)->R) {
      // replace by comment for persistent
      (*c)->set(p, x);
      // *c = (*c)->set(p,x);
    } else {
      int l = L, r = R;
      while ((p < m) == ((*c)->L < m)) {
        if (p < m) r = m;
        else l = m;
        m = (l + r) / 2;
      }
      *c = new ST(l, r, *c, new ST(p, p + 1, x));
      // The code above, inside this else block, could be
      // replaced by the following 2 lines when the complete
      // range has the form [0, 2^k)
      // int rm = (1<<(32-__builtin_clz(p^(*c)->L)))-1;
      // *c = new ST(p & ~rm, (p | rm)+1, *c, new ST(p, p+1, x));
    }
```

```
55        val = oper(lc ? lc->val : neutro, rc ? rc->val : neutro);
56        // return this; // uncomment for persistent
57    }
58    tipo get(int ql, int qr) {   // O(log(R-L))
59        if (qr <= L || R <= ql) return neutro;
60        if (ql <= L && R <= qr) return val;
61        return oper(lc ? lc->get(ql, qr) : neutro, rc ? rc->get(ql, qr) : neutro);
62    }
63 };  // Usage: 1- RMQ st(MIN_INDEX, MAX_INDEX) 2- normally use set/get
```

4.4.6   ST 2d

```
1  #define operacion(x, y) max(x, y)
2  int n, m;
3  int a[MAXN][MAXN], st[2 * MAXN][2 * MAXN];
4  void build() {   // O(n*m)
5    forn(i, n) forn(j, m) st[i + n][j + m] = a[i][j];
6    forn(i, n) dforn(j, m)  // build st of row i+n (each row independently)
7        st[i + n][j] = operacion(st[i + n][j << 1], st[i + n][j << 1 | 1]);
8    dforn(i, n) forn(j, 2 * m)  // build st of ranges of rows
9        st[i][j] = operacion(st[i << 1][j], st[i << 1 | 1][j]);
10 }
11 void upd(int x, int y, int v) {   // O(logn * logm)
12   st[x + n][y + m] = v;
13   for (int j = y + m; j > 1; j >>= 1)  // update ranges containing y+m
14     st[x + n][j >> 1] = operacion(st[x + n][j], st[x + n][j ^ 1]);
15   for (int i = x + n; i > 1; i >>= 1)  // in each range that contains row x+n
16     for (int j = y + m; j; j >>= 1)       // update the ranges that contains y+m
17       st[i >> 1][j] = operacion(st[i][j], st[i ^ 1][j]);
18 }
19 int query(int x0, int x1, int y0, int y1) {   // O(logn * logm)
20   int r = NEUT;
21   // start at the bottom and move up each time
22   for (int i0 = x0 + n, i1 = x1 + n; i0 < i1; i0 >>= 1, i1 >>= 1) {
23     int t[4], q = 0;
24     // if the whole segment of row node i0 is included, then move right
25     if (i0 & 1) t[q++] = i0++;
26     // if the whole segment of row node i1-1 is included, then move left
27     if (i1 & 1) t[q++] = --i1;
28     forn(k, q) for (int j0 = y0 + m, j1 = y1 + m; j0 < j1; j0 >>= 1, j1 >>= 1)
29         {
30       if (j0 & 1) r = operacion(r, st[t[k]][j0++]);
31       if (j1 & 1) r = operacion(r, st[t[k]][--j1]);
32     }
33   }
34   return r;
35 }
```

4.5   Merge sort tree

```
1  typedef ii datain;      // data that goes into the DS
2  typedef int query;      // info related to a query
3  typedef bool dataout;   // data that results from a query
4  struct DS {
5    set<datain> s;  // replace set with what's needed for the problem
6    void insert(const datain& x) {
7      // modify this method according to problem
8      // the example below is "disjoint intervals" (i.e. union of ranges)
9      datain xx = x;   // copy to avoid changing original
10     if (xx.fst >= xx.snd) return;
11     auto at = s.lower_bound(xx);
12     auto it = at;
13     if (at != s.begin() && (--at)->snd >= xx.fst) xx.fst = at->fst, --it;
14     for (; it != s.end() && it->fst <= xx.snd; s.erase(it++))
15       xx.snd = max(xx.snd, it->snd);
16     s.insert(xx);
17   }
18   void get(const query& q, dataout& ans) {
19     // modify this method according to problem
20     // the example below is "is there any range covering q?"
21     set<datain>::iterator ite = s.ub(mp(q + 1, 0));
22     if (ite != s.begin() && prev(ite)->snd > q) ans = true;
23   }
24 };
25 struct MST {
26   int sz;
27   vector<DS> t;
28   MST(int n) {
29     sz = 1 << (32 - __builtin_clz(n));
30     t = vector<DS>(2 * sz);
31   }
32   void insert(int i, int j, datain& x) { insert(i, j, x, 1, 0, sz); }
33   void insert(int i, int j, datain& x, int n, int a, int b) {
34     if (j <= a || b <= i) return;
35     if (i <= a && b <= j) {
36       t[n].insert(x);
37       return;
38     }
39     // needed when want to update ranges that intersec with [i,j)
40     // usually only needed on range-query + point-update problem
41     // t[n].insert(x);
42     int c = (a + b) / 2;
43     insert(i, j, x, 2 * n, a, c);
44     insert(i, j, x, 2 * n + 1, c, b);
45   }
46   void get(int i, int j, query& q, dataout& ans) {
47     return get(i, j, q, ans, 1, 0, sz);
48   }
49   void get(int i, int j, query& q, dataout& ans, int n, int a, int b) {
50     if (j <= a || b <= i) return;
51     if (i <= a && b <= j) {
52       t[n].get(q, ans);
53       return;
54     }
```

```
55      // needed when want to get from ranges that intersec with [i,j)
56      // usually only needed on point-query + range-update problem
57      // t[n].get(q, ans);
58      int c = (a + b) / 2;
59      get(i, j, q, ans, 2 * n, a, c);
60      get(i, j, q, ans, 2 * n + 1, c, b);
61    }
62  };  // Use: 1- definir todo lo necesario en DS, 2- usar
```

4.6   Fenwick tree

```
1  struct FenwickTree {
2    int N;                  // maybe replace vector with unordered_map when "many 0s"
3    vector<tipo> ft;  // for more dimensions, make ft multi-dimensional
4    FenwickTree(int n) : N(n), ft(n + 1) {}
5    void upd(int i0, tipo v) {  // add v to i0th element (0-based)
6      // add extra fors for more dimensions
7      for (int i = i0 + 1; i <= N; i += i & -i) ft[i] += v;
8    }
9    tipo get(int i0) {  // get sum of range [0,i0)
10     tipo r = 0;        // add extra fors for more dimensions
11     for (int i = i0; i; i -= i & -i) r += ft[i];
12     return r;
13   }
14   tipo get_sum(int i0, int i1) {  // get sum of range [i0,i1) (0-based)
15     return get(i1) - get(i0);
16   }
17 };
```

4.7   Link-cut tree

```
1  const int N_DEL = 0, N_VAL = 0; // neutral elements for delta & values
2  inline int u_oper(int x, int y){ return x + y; } // update operation
3  inline int q_oper(int lval, int rval){ return lval + rval; } // query
       operation
4  inline int u_segm(int d, int len){return d==N_DEL?N_DEL:d*len;} // upd segment
5  inline int u_delta(int d1, int d2){ // update delta
6    if(d1==N_DEL) return d2;
7    if(d2==N_DEL) return d1;
8    return u_oper(d1, d2);
9  }
10 inline int a_delta(int v, int d){ // apply delta
11   return d==N_DEL ? v : u_oper(d, v);
12 }
13
14 // Splay tree
15 struct node_t{
16   int szi, n_val, t_val, d;
17   bool rev;
18   node_t *c[2], *p;
19   node_t(int v) : szi(1), n_val(v), t_val(v), d(N_DEL), rev(0), p(0){
```

```
20     c[0]=c[1]=0;
21   }
22   bool is_root(){return !p || (p->c[0] != this && p->c[1] != this);}
23   void push(){
24     if(rev){
25       rev=0; swap(c[0], c[1]);
26       forr(x,0,2) if(c[x]) c[x]->rev^=1;
27     }
28     n_val = a_delta(n_val, d); t_val=a_delta(t_val, u_segm(d, szi));
29     forr(x,0,2) if(c[x])
30     c[x]->d = u_delta(d, c[x]->d);
31     d=N_DEL;
32   }
33   void upd();
34 };
35 typedef node_t* node;
36 int get_sz(node r){return r ? r->szi : 0;}
37 int get_tree_val(node r){
38   return r ? a_delta(r->t_val, u_segm(r->d,r->szi)) : N_VAL;
39 }
40 void node_t::upd() {
41   t_val=q_oper(q_oper(get_tree_val(c[0]),a_delta(n_val,d)),get_tree_val(c[1]))
        ;
42   szi = 1 + get_sz(c[0]) + get_sz(c[1]);
43 }
44 void conn(node c, node p, int is_left){
45   if(c) c->p = p;
46   if(is_left>=0) p->c[!is_left] = c;
47 }
48 void rotate(node x){
49   node p = x->p, g = p->p;
50   bool gCh=p->is_root(), is_left = x==p->c[0];
51   conn(x->c[is_left],p,is_left);
52   conn(p,x,!is_left);
53   conn(x,g,gCh?-1:(p==g->c[0]));
54   p->upd();
55 }
56 void splay(node x){
57   while(!x->is_root()){
58     node p = x->p, g = p->p;
59     if(!p->is_root()) g->push();
60     p->push(); x->push();
61     if(!p->is_root()) rotate((x==p->c[0])==(p==g->c[0])? p : x);
62     rotate(x);
63   }
64   x->push(); x->upd();
65 }
66
67 // Link-cut Tree
68 // Keep information of a tree (or forest) and allow to make many types of
69 // operations (see them below) in an efficient way. Internally, each node of
70 // the tree will have at most 1 "preferred" child, and as a consequence, the
71 // tree can be seen as a set of independent "preferred" paths. Each of this
72 // paths is basically a list, represented with a splay tree, where the
73 // "implicit key" (for the BST) of each element is the depth of the
74 // corresponding node in the original tree (or forest). Also, each of these
```

```cpp
75  // preferred paths (except one of them), will know who its "father path" is,
76  // i.e. will know the preferred path of the father of the top-most node.
77
78  // Make the path from the root to 'x' to be a "preferred path", and also make
79  // 'x' to be the root of its splay tree (not the root of the original tree).
80  node expose(node x){
81    node last = 0;
82    for(node y=x; y; y=y->p)
83    splay(y), y->c[0] = last, y->upd(), last = y;
84    splay(x);
85    return last;
86  }
87  void make_root(node x){expose(x);x->rev^=1;}
88  node get_root(node x){
89    expose(x);
90    while(x->c[1]) x = x->c[1];
91    splay(x);
92    return x;
93  }
94  node lca(node x, node y){expose(x); return expose(y);}
95  bool connected(node x, node y){
96    expose(x); expose(y);
97    return x==y ? 1 : x->p!=0;
98  }
99  // makes x son of y
100 void link(node x, node y){ make_root(x); x->p=y; }
101 void cut(node x, node y){ make_root(x); expose(y); y->c[1]->p=0; y->c[1]=0; }
102 node father(node x){
103   expose(x);
104   node r = x->c[1];
105   if(!r) return 0;
106   while(r->c[0]) r = r->c[0];
107   return r;
108 }
109 // cuts x from its father keeping tree root
110 void cut(node x){ expose(father(x)); x->p = 0; }
111 int query(node x, node y){
112   make_root(x); expose(y);
113   return get_tree_val(y);
114 }
115 void update(node x, node y, int d){
116   make_root(x); expose(y); y->d=u_delta(y->d,d);
117 }
118 node lift_rec(node x, int k){
119   if(!x) return 0;
120   if(k == get_sz(x->c[0])){ splay(x); return x; }
121   if(k < get_sz(x->c[0])) return lift_rec(x->c[0],k);
122   return lift_rec(x->c[1], k-get_sz(x->c[0])-1);
123 }
124 // k-th ancestor of x (lift(x,1) is x's father)
125 node lift(node x, int k){ expose(x);return lift_rec(x,k); }
126 // distance from x to its tree root
127 int depth(node x){ expose(x);return get_sz(x)-1; }
```

4.8   Implicit treap

```cpp
1   // An array represented as a treap, where the "key" is the index.
2   // However, the key is not stored explicitly, but can be calculated as
3   // the sum of the sizes of the left child of the ancestors where the node
4   // is in the right subtree of it.
5   // (commented parts are specific to range sum queries and other problems)
6   // rng = random number generator, works better than rand in some cases
7   mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
8   typedef struct item* pitem;
9   struct item {
10    int pr, cnt, val;
11    bool rev;   // for reverse operation
12    int sum;    // for range query
13    int add;    // for lazy prop
14    pitem l, r;
15    pitem p;   // ptr to parent, for getRoot
16    item(int val) : pr(rng()), cnt(1), val(val), rev(false), sum(val), add(0) {
17      l = r = p = NULL;
18    }
19  };
20  void push(pitem node) {
21    if (node) {
22      // for reverse operation
23      if (node->rev) {
24        swap(node->l, node->r);
25        if (node->l) node->l->rev ^= true;
26        if (node->r) node->r->rev ^= true;
27        node->rev = false;
28      }
29      // for lazy prop
30      node->val += node->add, node->sum += node->cnt * node->add;
31      if (node->l) node->l->add += node->add;
32      if (node->r) node->r->add += node->add;
33      node->add = 0;
34    }
35  }
36  int cnt(pitem t) { return t ? t->cnt : 0; }
37  int sum(pitem t) { return t ? push(t), t->sum : 0; }  // for range query
38  void upd_cnt(pitem t) {
39    if (t) {
40      t->cnt = cnt(t->l) + cnt(t->r) + 1;
41      t->sum = t->val + sum(t->l) + sum(t->r);   // for range sum
42      if (t->l) t->l->p = t;                      // for getRoot
43      if (t->r) t->r->p = t;                      // for getRoot
44      t->p = NULL;                                // for getRoot
45    }
46  }
47  void split(pitem node, pitem& L, pitem& R, int sz) {  // sz: wanted size for L
48    if (!node) {
49      L = R = 0;
50      return;
51    }
52    push(node);
53    // If node's left child has at least sz nodes, go left
54    if (sz <= cnt(node->l)) split(node->l, L, node->l, sz), R = node;
```

```
55    // Else, go right changing wanted sz
56    else split(node->r, node->r, R, sz - 1 - cnt(node->l)), L = node;
57    upd_cnt(node);
58  }
59  void merge(pitem& result, pitem L, pitem R) {  // O(log)
60    push(L), push(R);
61    if (!L || !R) result = L ? L : R;
62    else if (L->pr > R->pr) merge(L->r, L->r, R), result = L;
63    else merge(R->l, L, R->l), result = R;
64    upd_cnt(result);
65  }
66  void insert(pitem& node, pitem x, int pos) {  // 0-index O(log)
67    pitem l, r;
68    split(node, l, r, pos);
69    merge(l, l, x);
70    merge(node, l, r);
71  }
72  void erase(pitem& node, int pos) {  // 0-index O(log)
73    if (!node) return;
74    push(node);
75    if (pos == cnt(node->l)) merge(node, node->l, node->r);
76    else if (pos < cnt(node->l)) erase(node->l, pos);
77    else erase(node->r, pos - 1 - cnt(node->l));
78    upd_cnt(node);
79  }
80  // reverse operation
81  void reverse(pitem& node, int L, int R) {  //[L, R) O(log)
82    pitem t1, t2, t3;
83    split(node, t1, t2, L);
84    split(t2, t2, t3, R - L);
85    t2->rev ^= true;
86    merge(node, t1, t2);
87    merge(node, node, t3);
88  }
89  // lazy add
90  void add(pitem& node, int L, int R, int x) {  //[L, R) O(log)
91    pitem t1, t2, t3;
92    split(node, t1, t2, L);
93    split(t2, t2, t3, R - L);
94    t2->add += x;
95    merge(node, t1, t2);
96    merge(node, node, t3);
97  }
98  // range query get
99  int get(pitem& node, int L, int R) {  //[L, R) O(log)
100   pitem t1, t2, t3;
101   split(node, t1, t2, L);
102   split(t2, t2, t3, R - L);
103   push(t2);
104   int ret = t2->sum;
105   merge(node, t1, t2);
106   merge(node, node, t3);
107   return ret;
108 }
109 void push_all(pitem t) {  // for getRoot
110   if (t->p) push_all(t->p);
```

```
111   push(t);
112 }
113 pitem getRoot(pitem t, int& pos) {  // get root and position for node t
114   push_all(t);
115   pos = cnt(t->l);
116   while (t->p) {
117     pitem p = t->p;
118     if (t == p->r) pos += cnt(p->l) + 1;
119     t = p;
120   }
121   return t;
122 }
123 void output(pitem t) {  // useful for debugging
124   if (!t) return;
125   push(t);
126   output(t->l);
127   cout << ' ' << t->val;
128   output(t->r);
129 }
```

4.9   Treap (not implicit)

```
1  typedef struct item* pitem;
2  struct item {
3    // pr = randomized priority, key = BST value, cnt = size of subtree
4    int pr, key, cnt;
5    pitem l, r;
6    item(int key) : key(key), pr(rand()), cnt(1), l(NULL), r(NULL) {}
7  };
8  int cnt(pitem node) { return node ? node->cnt : 0; }
9  void upd_cnt(pitem node) {
10   if (node) node->cnt = cnt(node->l) + cnt(node->r) + 1;
11 }
12 // splits t in l and r - l: <= key, r: > key
13 void split(pitem node, int key, pitem& L, pitem& R) {  // O(log)
14   if (!node) L = R = 0;
15   // if cur > key, go left to split and cur is part of R
16   else if (key < node->key) split(node->l, key, L, node->l), R = node;
17   // if cur <= key, go right to split and cur is part of L
18   else split(node->r, key, node->r, R), L = node;
19   upd_cnt(node);
20 }
21 // 1) go down the BST following the key of the new node (x), until
22 //  you reach NULL or a node with lower pr than the new one.
23 // 2.1) if you reach NULL, put the new node there
24 // 2.2) if you reach a node with lower pr, split the subtree rooted at that
25 // node, put the new one there and put the split result as children of it
26 void insert(pitem& node, pitem x) {  // O(log)
27   if (!node) node = x;
28   else if (x->pr > node->pr) split(node, x->key, x->l, x->r), node = x;
29   else insert(x->key <= node->key ? node->l : node->r, x);
30   upd_cnt(node);
31 }
32 // Assumes that the key of every element in L <= to the keys in R
```

```cpp
void merge(pitem& result, pitem L, pitem R) {  // O(log)
  // If one of the nodes is NULL, the merge result is the other node
  if (!L || !R) result = L ? L : R;
  // if L has higher priority than R, put L and update it's right child
  // with the merge result of L->r and R
  else if (L->pr > R->pr) merge(L->r, L->r, R), result = L;
  // if R has higher priority than L, put R and update it's left child
  // with the merge result of L and R->l
  else merge(R->l, L, R->l), result = R;
  upd_cnt(result);
}
// go down the BST following the key to erase. When the key is found,
// replace that node with the result of merging it children
void erase(pitem& node, int key) {  // O(log), (erases only 1 repetition)
  if (node->key == key) merge(node, node->l, node->r);
  else erase(key < node->key ? node->l : node->r, key);
  upd_cnt(node);
}
// union of two treaps
void unite(pitem& t, pitem L, pitem R) {  // O(M*log(N/M))
  if (!L || !R) {
    t = L ? L : R;
    return;
  }
  if (L->pr < R->pr) swap(L, R);
  pitem p1, p2;
  split(R, L->key, p1, p2);
  unite(L->l, L->l, p1);
  unite(L->r, L->r, p2);
  t = L;
  upd_cnt(t);
}
pitem kth(pitem t, int k) {  // element at "position" k
  if (!t) return 0;
  if (k == cnt(t->l)) return t;
  return k < cnt(t->l) ? kth(t->l, k) : kth(t->r, k - cnt(t->l) - 1);
}
pair<int, int> lb(pitem t, int key) {  // position and value of lower_bound
  if (!t) return {0, 1 << 30};          // (special value)
  if (key > t->key) {
    auto w = lb(t->r, key);
    w.fst += cnt(t->l) + 1;
    return w;
  }
  auto w = lb(t->l, key);
  if (w.fst == cnt(t->l)) w.snd = t->key;
  return w;
}
```

4.10 STL rope

```cpp
#include <ext/rope>
using namespace __gnu_cxx;
rope<int> s;
// Sequence with O(log(n)) random access, insert, erase at any position
// s.push_back(x)
// s.append(other_rope)
// s.insert(i,x)
// s.insert(i,other_rope) // insert rope r at position i
// s.erase(i,k) // erase subsequence [i,i+k)
// s.substr(i,k) // return new rope corresponding to subsequence [i,i+k)
// s[i] // access ith element (cannot modify)
// s.mutable_reference_at(i) // acces ith element (allows modification)
// s.begin() and s.end() are const iterators (use mutable_begin(), mutable_end
    ()
// to allow modification)
```

4.11 BIGInt

```cpp
#define BASEXP 6
#define BASE 1000000
#define LMAX 1000
struct bint {
  int l;
  ll n[LMAX];
  bint(ll x = 0) {
    l = 1;
    forn(i, LMAX) {
      if (x) l = i + 1;
      n[i] = x % BASE;
      x /= BASE;
    }
  }
  bint(string x) {
    l = (x.size() - 1) / BASEXP + 1;
    fill(n, n + LMAX, 0);
    ll r = 1;
    forn(i, sz(x)) {
      n[i / BASEXP] += r * (x[x.size() - 1 - i] - '0');
      r *= 10;
      if (r == BASE) r = 1;
    }
  }
  void out() {
    cout << n[l - 1];
    dforn(i, l - 1) printf("%6.6llu", n[i]);  // 6=BASEXP!
  }
  void invar() {
    fill(n + l, n + LMAX, 0);
    while (l > 1 && !n[l - 1]) l--;
  }
};
```

```
34  bint operator+(const bint& a, const bint& b) {
35    bint c;
36    c.l = max(a.l, b.l);
37    ll q = 0;
38    forn(i, c.l) q += a.n[i] + b.n[i], c.n[i] = q % BASE, q /= BASE;
39    if (q) c.n[c.l++] = q;
40    c.invar();
41    return c;
42  }
43  pair<bint, bool> lresta(const bint& a, const bint& b)  // c = a - b
44  {
45    bint c;
46    c.l = max(a.l, b.l);
47    ll q = 0;
48    forn(i, c.l) q += a.n[i] - b.n[i], c.n[i] = (q + BASE) % BASE,
49                                       q = (q + BASE) / BASE - 1;
50    c.invar();
51    return make_pair(c, !q);
52  }
53  bint& operator-=(bint& a, const bint& b) { return a = lresta(a, b).first; }
54  bint operator-(const bint& a, const bint& b) { return lresta(a, b).first; }
55  bool operator<(const bint& a, const bint& b) { return !lresta(a, b).second; }
56  bool operator<=(const bint& a, const bint& b) { return lresta(b, a).second; }
57  bool operator==(const bint& a, const bint& b) { return a <= b && b <= a; }
58  bint operator*(const bint& a, ll b) {
59    bint c;
60    ll q = 0;
61    forn(i, a.l) q += a.n[i] * b, c.n[i] = q % BASE, q /= BASE;
62    c.l = a.l;
63    while (q) c.n[c.l++] = q % BASE, q /= BASE;
64    c.invar();
65    return c;
66  }
67  bint operator*(const bint& a, const bint& b) {
68    bint c;
69    c.l = a.l + b.l;
70    fill(c.n, c.n + b.l, 0);
71    forn(i, a.l) {
72      ll q = 0;
73      forn(j, b.l) q += a.n[i] * b.n[j] + c.n[i + j], c.n[i + j] = q % BASE,
74                                         q /= BASE;
75      c.n[i + b.l] = q;
76    }
77    c.invar();
78    return c;
79  }
80  pair<bint, ll> ldiv(const bint& a, ll b) {  // c = a / b ; rm = a % b
81    bint c;
82    ll rm = 0;
83    dforn(i, a.l) {
84      rm = rm * BASE + a.n[i];
85      c.n[i] = rm / b;
86      rm %= b;
87    }
88    c.l = a.l;
89    c.invar();
```

```
90     return make_pair(c, rm);
91  }
92  bint operator/(const bint& a, ll b) { return ldiv(a, b).first; }
93  ll operator%(const bint& a, ll b) { return ldiv(a, b).second; }
94  pair<bint, bint> ldiv(const bint& a, const bint& b) {
95    bint c;
96    bint rm = 0;
97    dforn(i, a.l) {
98      if (rm.l == 1 && !rm.n[0]) rm.n[0] = a.n[i];
99      else {
100       dforn(j, rm.l) rm.n[j + 1] = rm.n[j];
101       rm.n[0] = a.n[i];
102       rm.l++;
103     }
104     ll q = rm.n[b.l] * BASE + rm.n[b.l - 1];
105     ll u = q / (b.n[b.l - 1] + 1);
106     ll v = q / b.n[b.l - 1] + 1;
107     while (u < v - 1) {
108       ll m = (u + v) / 2;
109       if (b * m <= rm) u = m;
110       else v = m;
111     }
112     c.n[i] = u;
113     rm -= b * u;
114   }
115   c.l = a.l;
116   c.invar();
117   return make_pair(c, rm);
118 }
119 bint operator/(const bint& a, const bint& b) { return ldiv(a, b).first; }
120 bint operator%(const bint& a, const bint& b) { return ldiv(a, b).second; }
```

4.12   Gain cost set

```
1   // stores pairs (benefit,cost) (erases non-optimal pairs)
2   // Note that these pairs will be increasing by g and increasing by c
3   // If we insert a pair that is included in other, the big one will be deleted
4   // For lis 2d, create a GCS por each possible length, use as (-g, c) and
5   // binary search looking for the longest length where (-g, c) could be added
6   struct GCS {
7     set<ii> s;
8     void add(int g, int c) {
9       ii x = {g, c};
10      auto p = s.lower_bound(x);
11      if (p != s.end() && p->snd <= x.snd) return;
12      if (p != s.begin()) {  // erase pairs with less or eq benefit and more
13          cost
14        --p;
15        while (p->snd >= x.snd) {
16          if (p == s.begin()) {
17            s.erase(p);
18            break;
19          }
20          s.erase(p--);
```

```
20        }
21      }
22      s.insert(x);
23    }
24    int get(int gain) {  // min cost for the benefit greater or equal to gain
25      auto p = s.lower_bound((ii){gain, -INF});
26      int r = p == s.end() ? INF : p->snd;
27      return r;
28    }
29 };
```

5   Strings

5.1   Z function

```
1  // z[i] = length of longest substring starting from s[i] that is prefix of s
2  // z[i] = max k: s[0,k) == s[i,i+k)
3  vector<int> zFunction(string& s) {
4    int l = 0, r = 0, n = sz(s);
5    vector<int> z(n, 0);
6    forr(i, 1, n) {
7      if (i <= r) z[i] = min(r - i + 1, z[i - l]);
8      while (i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
9      if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
10   }
11   return z;
12 }
13 void match(string& T, string& P) {  // Text, Pattern -- O(|T|+|P|)
14   string s = P + '$' + T;  //'$' should be a character that is not present in
         T
15   vector<int> z = zFunction(s);
16   forr(i, sz(P) + 1, sz(s)) if (z[i] == sz(P));  // match found, idx = i-sz(P)
         -1
17 }
```

5.2   KMP

```
1  // b[i] = longest border of t[0,i) = length of the longest prefix of
2  // the substring P[0..i-1] that is also suffix of the substring P[0..i]
3  // For "AABAACAABAA", b[i] = {-1, 0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5}
4  vector<int> kmppre(string& P) {  //
5    vector<int> b(sz(P) + 1);
6    b[0] = -1;
7    int j = -1;
8    forn(i, sz(P)) {
9      while (j >= 0 && P[i] != P[j]) j = b[j];
10     b[i + 1] = ++j;
11   }
12   return b;
13 }
14 void kmp(string& T, string& P) {  // Text, Pattern -- O(|T|+|P|)
15   int j = 0;
16   vector<int> b = kmppre(P);
17   forn(i, sz(T)) {
18     while (j >= 0 && T[i] != P[j]) j = b[j];
19     if (++j == sz(P)) {
20       // Match at i-j+1, do something
21       j = b[j];
22     }
23   }
24 }
```

5.3   Manacher

```
1  int d1[MAXN];  // d1[i] = max odd palindrome centered on i
2  int d2[MAXN];  // d2[i] = max even palindrome centered on i
3  // s   aabbaacaabbaa
4  // d1 1111117111111
5  // d2 0103010010301
6  void manacher(string& s) {  // O(|S|) - find longest palindromic substring
7    int l = 0, r = -1, n = s.size();
8    forn(i, n) {  // build d1
9      int k = i > r ? 1 : min(d1[l + r - i], r - i);
10     while (i + k < n && i - k >= 0 && s[i + k] == s[i - k]) k++;
11     d1[i] = k--;
12     if (i + k > r) l = i - k, r = i + k;
13   }
14   l = 0, r = -1;
15   forn(i, n) {  // build d2
16     int k = (i > r ? 0 : min(d2[l + r - i + 1], r - i + 1)) + 1;
17     while (i + k <= n && i - k >= 0 && s[i + k - 1] == s[i - k]) k++;
18     d2[i] = --k;
19     if (i + k - 1 > r) l = i - k, r = i + k - 1;
20   }
21 }
```

5.4   Booth's algorithm

```
1  // Booth's algorithm
2  // Find lexicographically minimal string rotation in O(|S|)
3  int booth(string S) {
4    S += S;  // Concatenate string to it self to avoid modular arithmetic
5    int n = sz(S);
6    vector<int> f(n, -1);
7    int k = 0;  // Least rotation of string found so far
8    forr(j, 1, n) {
9      char sj = S[j];
10     int i = f[j - k - 1];
11     while (i != -1 and sj != S[k + i + 1]) {
12       if (sj < S[k + i + 1]) k = j - i - 1;
13       i = f[i];
14     }
15     if (sj != S[k + i + 1]) {
16       if (sj < S[k]) k = j;
17       f[j - k] = -1;
18     } else {
19       f[j - k] = i + 1;
20     }
21   }
22   return k;  // Lexicographically minimal string rotation's position
23 }
```

5.5   Hashing

5.5.1   Classic hashing (with substring hash)

```
1  // P should be a prime number, could be randomly generated,
2  // sometimes is good to make it close to alphabet size
3  // MOD[i] must be a prime of this order, could be randomly generated
4  const int P = 1777771, MOD[2] = {999727999, 1070777777};
5  const int PI[2] = {325255434, 10018302};  // PI[i] = P^-1 % MOD[i]
6  struct Hash {
7    vector<int> h[2], pi[2];
8    vector<ll> vp[2];  // Only used if getChanged is used (delete it if not)
9    Hash(string& s) {
10     forn(k, 2) h[k].rsz(s.size() + 1), pi[k].rsz(s.size() + 1),
11         vp[k].rsz(s.size() + 1);
12     forn(k, 2) {
13       h[k][0] = 0;
14       vp[k][0] = pi[k][0] = 1;
15       ll p = 1;
16       forr(i, 1, sz(s) + 1) {
17         h[k][i] = (h[k][i - 1] + p * s[i - 1]) % MOD[k];
18         pi[k][i] = (1LL * pi[k][i - 1] * PI[k]) % MOD[k];
19         vp[k][i] = p = (p * P) % MOD[k];
20       }
21     }
22   }
23   ll get(int s, int e) {  // get hash value of the substring [s, e)
24     ll H[2];
25     forn(i, 2) {
26       H[i] = (h[i][e] - h[i][s] + MOD[i]) % MOD[i];
27       H[i] = (1LL * H[i] * pi[i][s]) % MOD[i];
28     }
29     return (H[0] << 32) | H[1];
30   }
31   // get hash value of [s, e) if origVal in pos is changed to val
32   // Assumes s <= pos < e. If multiple changes are needed,
33   // do what is done in the for loop for every change
34   ll getChanged(int s, int e, int pos, int val, int origVal) {
35     ll hv = get(s, e), hh[2];
36     hh[1] = hv & ((1LL << 32) - 1);
37     hh[0] = hv >> 32;
38     forn(i, 2) hh[i] = (hh[i] + vp[i][pos] * (val - origVal + MOD[i])) % MOD[i
          ];
39     return (hh[0] << 32) | hh[1];
40   }
41 };
```

5.5.2   Simple hashing (no substring hash)

```
1  // P should be a prime number, could be randomly generated,
2  // sometimes is good to make it close to alphabet size
3  // MOD[i] must be a prime of this order, could be randomly generated
4  const int P = 1777771, MOD[2] = {999727999, 1070777777};
5  const int PI[2] = {325255434, 10018302};  // PI[i] = P^-1 % MOD[i]
6  struct Hash {
7    ll h[2];
8    vector<ll> vp[2];
9    deque<int> x;
10   Hash(vector<int>& s) {
11     forn(i, sz(s)) x.pb(s[i]);
12     forn(k, 2) vp[k].rsz(s.size() + 1);
13     forn(k, 2) {
14       h[k] = 0;
15       vp[k][0] = 1;
16       ll p = 1;
17       forr(i, 1, sz(s) + 1) {
18         h[k] = (h[k] + p * s[i - 1]) % MOD[k];
19         vp[k][i] = p = (p * P) % MOD[k];
20       }
21     }
22   }
23   // Put the value val in position pos and update the hash value
24   void change(int pos, int val) {
25     forn(i, 2) h[i] = (h[i] + vp[i][pos] * (val - x[pos] + MOD[i])) % MOD[i];
26     x[pos] = val;
27   }
28   // Add val to the end of the current string
29   void push_back(int val) {
30     int pos = sz(x);
31     x.pb(val);
32     forn(k, 2) {
33       assert(pos <= sz(vp[k]));
34       if (pos == sz(vp[k])) vp[k].pb(vp[k].back() * P % MOD[k]);
35       ll p = vp[k][pos];
36       h[k] = (h[k] + p * val) % MOD[k];
37     }
38   }
39   // Delete the first element of the current string
40   void pop_front() {
41     assert(sz(x) > 0);
42     forn(k, 2) {
43       h[k] = (h[k] - x[0] + MOD[k]) % MOD[k];
44       h[k] = h[k] * PI[k] % MOD[k];
45     }
46     x.pop_front();
47   }
48   ll getHashVal() { return (h[0] << 32) | h[1]; }
49 };
```

5.5.3   Hashing 128 bits

```
1  typedef __int128 bint;  // needs gcc compiler?
2  const bint MOD = 212345678987654321LL, P = 1777771, PI = 106955741089659571LL;
3  struct Hash {
4    vector<bint> h, pi;
5    Hash(string& s) {
6      assert((P * PI) % MOD == 1);
7      h.resize(s.size() + 1), pi.resize(s.size() + 1);
8      h[0] = 0, pi[0] = 1;
9      bint p = 1;
10     forr(i, 1, sz(s) + 1) {
11       h[i] = (h[i - 1] + p * s[i - 1]) % MOD;
12       pi[i] = (pi[i - 1] * PI) % MOD;
13       p = (p * P) % MOD;
14     }
15   }
16   ll get(int s, int e) {  // get hash value of the substring [s, e)
17     return (((h[e] - h[s] + MOD) % MOD) * pi[s]) % MOD;
18   }
19 };
```

5.6   Trie

```
1  struct Trie {
2    map<char, Trie> m;  // Trie* when using persistence
3    // For persistent trie only. Call "clone" probably from
4    // "add" and/or other methods, to implement persistence.
5    void clone(int pos) {
6      Trie* prev = NULL;
7      if (m.count(pos)) prev = m[pos];
8      m[pos] = new Trie();
9      if (prev != NULL) {
10       m[pos]->m = prev->m;
11       // copy other relevant data
12     }
13   }
14   void add(const string& s, int p = 0) {
15     if (s[p]) m[s[p]].add(s, p + 1);
16   }
17   void dfs() {
18     // Do stuff
19     forall(it, m) it->second.dfs();
20   }
21 };
```

5.7   Aho Corasick

```cpp
struct Node {
  map<char, int> next, go;
  int p, link, leafLink;
  char pch;
  vector<int> leaf;
  Node(int pp, char c) : p(pp), link(-1), leafLink(-1), pch(c) {}
};
struct AhoCorasick {
  vector<Node> t = {Node(-1, -1)};
  void add_string(string s, int id) {
    int v = 0;
    for (char c : s) {
      if (!t[v].next.count(c)) {
        t[v].next[c] = sz(t);
        t.pb(Node(v, c));
      }
      v = t[v].next[c];
    }
    t[v].leaf.pb(id);
  }
  int go(int v, char c) {
    if (!t[v].go.count(c)) {
      if (t[v].next.count(c)) t[v].go[c] = t[v].next[c];
      else t[v].go[c] = v == 0 ? 0 : go(get_link(v), c);
    }
    return t[v].go[c];
  }
  int get_link(int v) {  // suffix link
    if (t[v].link < 0) {
      if (!v || !t[v].p) t[v].link = 0;
      else t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
  }
  // like suffix link, but only going to the root or to a node with
  // a non-emtpy "leaf" list. Copy only if needed
  int get_leaf_link(int v) {
    if (t[v].leafLink < 0) {
      if (!v || !t[v].p) t[v].leafLink = 0;
      else if (!t[get_link(v)].leaf.empty()) t[v].leafLink = t[v].link;
      else t[v].leafLink = get_leaf_link(t[v].link);
    }
    return t[v].leafLink;
  }
};
```

5.8   Suffix array

5.8.1   Slow version O(n*logn*logn)

```cpp
pair<int, int> sf[MAXN];
bool sacomp(int lhs, int rhs) { return sf[lhs] < sf[rhs]; }
vector<int> constructSA(string& s) {  // O(n log^2(n))
  int n = s.size();                    // (sometimes fast enough)
  vector<int> sa(n), r(n);
  forn(i, n) r[i] = s[i];  // r[i]: equivalence class of s[i..i+m]
  for (int m = 1; m < n; m *= 2) {
    // sf[i] = {r[i], r[i+m]}, used to sort for next equivalence classes
    forn(i, n) sa[i] = i, sf[i] = {r[i], i + m < n ? r[i + m] : -1};
    stable_sort(sa.begin(), sa.end(), sacomp);  // O(n log(n))
    r[sa[0]] = 0;
    // if sf[sa[i]] == sf[sa[i-1]] then same equivalence class
    forr(i, 1, n) r[sa[i]] = sf[sa[i]] != sf[sa[i - 1]] ? i : r[sa[i - 1]];
  }
  return sa;
}
```

5.8.2   Fast version O(n*logn)

```cpp
#define RB(x)  (x < n ? r[x] : 0)
void csort(vector<int>& sa, vector<int>& r, int k) {  // counting sort O(n)
  int n = sa.size();
  vector<int> f(max(255, n), 0), t(n);
  forn(i, n) f[RB(i + k)]++;
  int sum = 0;
  forn(i, max(255, n)) f[i] = (sum += f[i]) - f[i];
  forn(i, n) t[f[RB(sa[i] + k)]++] = sa[i];
  sa = t;
}
vector<int> constructSA(string& s) {  // O(n logn)
  int n = s.size(), rank;
  vector<int> sa(n), r(n), t(n);
  forn(i, n) sa[i] = i, r[i] = s[i];  // r[i]: equivalence class of s[i..i+k]
  for (int k = 1; k < n; k *= 2) {
    csort(sa, r, k);
    csort(sa, r, 0);         // counting sort, O(n)
    t[sa[0]] = rank = 0;  // t : equivalence classes array for next size
    forr(i, 1, n) {
      // check if sa[i] and sa[i-1] are in te same equivalence class
      if (r[sa[i]] != r[sa[i - 1]] || RB(sa[i] + k) != RB(sa[i - 1] + k))
        rank++;
      t[sa[i]] = rank;
    }
    r = t;
    if (r[sa[n - 1]] == n - 1) break;
  }
  return sa;
}
```

5.8.3 Longest common prefix (LCP)

```cpp
// LCP(sa[i], sa[j]) = min(lcp[i+1], lcp[i+2], ..., lcp[j])
// example: "banana", sa = {5,3,1,0,4,2}, lcp = {0,1,3,0,0,2}
// Num of dif substrings: (n*n+n)/2 - (sum over lcp array)
// Build suffix array (sa) before calling
vector<int> computeLCP(string& s, vector<int>& sa) {
  int n = s.size(), L = 0;
  vector<int> lcp(n), plcp(n), phi(n);
  phi[sa[0]] = -1;
  forr(i, 1, n) phi[sa[i]] = sa[i - 1];
  forn(i, n) {
    if (phi[i] < 0) {
      plcp[i] = 0;
      continue;
    }
    while (s[i + L] == s[phi[i] + L]) L++;
    plcp[i] = L;
    L = max(L - 1, 0);
  }
  forn(i, n) lcp[i] = plcp[sa[i]];
  return lcp;  // lcp[i]=LCP(sa[i-1],sa[i])
}
```

5.9 Suffix automaton

```cpp
// The substrings of S can be decomposed into equivalence classes
// 2 substr are of the same class if they have the same set of endpos
// Example: endpos("bc") = {2, 4, 6} in "abcbcbc"
// Each class is a node of the automaton.
// Len is the longest substring of each class
// Link in state X is the state where the longest suffix of the longest
// substring in X, with a different endpos set, belongs
// The links form a tree rooted at 0
// last is the state of the whole string after each extention
struct state {
  int len, link;
  map<char, int> next;
};  // clear next!!
state st[MAXN];
int sz, last;
void sa_init() {
  last = st[0].len = 0;
  sz = 1;
  st[0].link = -1;
}
void sa_extend(char c) {
  int k = sz++, p;  // k = new state
  st[k].len = st[last].len + 1;
  // while c is not present in p assign it as edge to the new state and
  // move through link (note that p always corresponds to a suffix state)
  for (p = last; p != -1 && !st[p].next.count(c); p = st[p].link)
    st[p].next[c] = k;
  if (p == -1) st[k].link = 0;
  else {
    // state p already goes to state q through char c. Then, link of k
    // should go to a state with len = st[p].len + 1 (because of c)
    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len) st[k].link = q;
    else {
      // q is not the state we are looking for. Then, we
      // create a clone of q (w) with the desired length
      int w = sz++;
      st[w].len = st[p].len + 1;
      st[w].next = st[q].next;
      st[w].link = st[q].link;
      // go through links from p and while next[c] is q, change it to w
      for (; p != -1 && st[p].next[c] == q; p = st[p].link) st[p].next[c] = w;
      // change link of q from p to w, and finally set link of k to w
      st[q].link = st[k].link = w;
    }
  }
  last = k;
}
//  input: abcbcbc
//  i,link,len,next
//  0 -1 0 (a,1) (b,5) (c,7)
//  1 0 1 (b,2)
//  2 5 2 (c,3)
//  3 7 3 (b,4)
//  4 9 4 (c,6)
//  5 0 1 (c,7)
//  6 11 5 (b,8)
//  7 0 2 (b,9)
//  8 9 6 (c,10)
//  9 5 3 (c,11)
//  10 11 7
//  11 7 4 (b,8)
```

5.10   Suffix tree

```
1   const int INF = 1e6 + 10;   // INF > n
2   const int MAXLOG = 20;       // 2^MAXLOG > 2*n
3   // The SuffixTree of S is the compressed trie that would result after
4   // inserting every suffix of S.
5   // As it is a COMPRESSED trie, some edges may correspond to strings,
6   // instead of chars, and the compression is done in a way that every
7   // vertex that doesn't correspond to a suffix and has only one
8   // descendent, is omitted.
9   struct SuffixTree {
10    vector<char> s;
11    vector<map<int, int>> to;  // fst char of substring on edge -> node
12    // s[fpos[i], fpos[i]+len[i]) is the substring on the edge between
13    // i's father and i.
14    // link[i] goes to the node that corresponds to the substring that
15    // result after "removing" the first character of the substring that
16    // i represents. Only defined for internal (non-leaf) nodes.
17    vector<int> len, fpos, link;
18    SuffixTree(int nn = 0) {  // O(nn). nn should be the expected size
19      s.reserve(nn), to.reserve(2 * nn), len.reserve(2 * nn);
20      fpos.reserve(2 * nn), link.reserve(2 * nn);
21      make_node(0, INF);
22    }
23    int node = 0, pos = 0, n = 0;
24    int make_node(int p, int l) {
25      fpos.pb(p), len.pb(l), to.pb({}), link.pb(0);
26      return sz(to) - 1;
27    }
28    void go_edge() {
29      while (pos > len[to[node][s[n - pos]]]) {
30        node = to[node][s[n - pos]];
31        pos -= len[node];
32      }
33    }
34    void add(char c) {
35      s.pb(c), n++, pos++;
36      int last = 0;
37      while (pos > 0) {
38        go_edge();
39        int edge = s[n - pos];
40        int& v = to[node][edge];
41        int t = s[fpos[v] + pos - 1];
42        if (v == 0) {
43          v = make_node(n - pos, INF);
44          link[last] = node;
45          last = 0;
46        } else if (t == c) {
47          link[last] = node;
48          return;
49        } else {
50          int u = make_node(fpos[v], pos - 1);
51          to[u][c] = make_node(n - 1, INF);
52          to[u][t] = v;
53          fpos[v] += pos - 1, len[v] -= pos - 1;
54          v = u, link[last] = u, last = u;
55          }
56        if (node == 0) pos--;
57        else node = link[node];
58      }
59    }
60    // Call this after you finished building the SuffixTree to correctly
61    // set some values of the vector 'len' that currently have a big
62    // value (because of INF usage). Note that you shouldn't call 'add'
63    // anymore after calling this method.
64    void finishedAdding() {
65      forn(i, sz(len)) if (len[i] + fpos[i] > n) len[i] = n - fpos[i];
66    }
67    // From here, copy only if needed!!
68    // Map each suffix with it corresponding leaf node
69    // vleaf[i] = node id of leaf of suffix s[i..n)
70    // Note that the last character of the string must be unique
71    // Use 'buildLeaf' not 'dfs' directly. Also 'finishedAdding' must
72    // be called before calling 'buildLeaf'.
73    // When this is needed, usually binary lifting (vp) and depths are
74    // also needed.
75    // Usually you also need to compute extra information in the dfs.
76    vector<int> vleaf, vdepth;
77    vector<vector<int>> vp;
78    void dfs(int cur, int p, int curlen) {
79      if (cur > 0) curlen += len[cur];
80      vdepth[cur] = curlen;
81      vp[cur][0] = p;
82      if (to[cur].empty()) {
83        assert(0 < curlen && curlen <= n);
84        assert(vleaf[n - curlen] == -1);
85        vleaf[n - curlen] = cur;
86        // here maybe compute some extra info
87      } else forall(it, to[cur]) {
88          dfs(it->snd, cur, curlen);
89          // maybe change return type and here compute extra info
90        }
91      // maybe return something here related to extra info
92    }
93    void buildLeaf() {
94      vdepth.rsz(sz(to), 0);                 // tree size
95      vleaf.rsz(n, -1);                      // string size
96      vp.rsz(sz(to), vector<int>(MAXLOG));   // tree size * log
97      dfs(0, 0, 0);
98      forr(k, 1, MAXLOG) forn(i, sz(to)) vp[i][k] = vp[vp[i][k - 1]][k - 1];
99      forn(i, n) assert(vleaf[i] != -1);
100   }
101 };
```

6   Grafos

6.1   Dijkstra

```cpp
struct Dijkstra {              // WARNING: ii usually needs to be pair<ll, int>
  vector<vector<ii>> G;  // ady. list with pairs (weight, dst)
  vector<ll> dist;
  // vector<int> vp; // for path reconstruction (parent of each node)
  int N;
  Dijkstra(int n) : G(n), N(n) {}
  void addEdge(int a, int b, ll w) { G[a].pb(mp(w, b)); }
  void run(int src) {  // O(|E| log |V|)
    dist = vector<ll>(N, INF);
    // vp = vector<int>(N, -1);
    priority_queue<ii, vector<ii>, greater<ii>> Q;
    Q.push(make_pair(0, src)), dist[src] = 0;
    while (sz(Q)) {
      int node = Q.top().snd;
      ll d = Q.top().fst;
      Q.pop();
      if (d > dist[node]) continue;
      forall(it, G[node]) if (d + it->fst < dist[it->snd]) {
        dist[it->snd] = d + it->fst;
        // vp[it->snd] = node;
        Q.push(mp(dist[it->snd], it->snd));
      }
    }
  }
};
```

6.2   Floyd-Warshall

```cpp
// Min path between every pair of nodes in directed graph
// G[i][j] initially needs weight of edge (i, j) or INF
// be careful with multiedges and loops when assigning to G
int G[MAX_N][MAX_N];
void floyd() {  // O(N^3)
  forn(k, N) forn(i, N) if (G[i][k] != INF) forn(j, N) if (G[k][j] != INF)
    G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
}
bool inNegCycle(int v) { return G[v][v] < 0; }
// checks if there's a neg. cycle in path from a to b
bool hasNegCycle(int a, int b) {
  forn(i, N) if (G[a][i] != INF && G[i][i] < 0 && G[i][b] != INF) return true;
  return false;
}
```

6.3   Bellman-Ford

```cpp
// Mas lento que Dijsktra, pero maneja arcos con peso negativo
//
// Can solve systems of "difference inequalities":
// 1. for each inequality x_i - x_j <= k add an edge j->i with weight k
// 2. create an extra node Z and add an edge Z->i with weigth 0 for
//   each variable x_i in the inequalities
// 3. run(Z): if negcycle, no solution, otherwise "dist" is a solution
//
// Can transform a graph to get all edges of positive weight
//("Jhonson algorigthm"):
// 1. Create an extra node Z and add edge Z->i with weight 0 for all
//   nodes i
// 2. Run bellman ford from Z
// 3. For each original edge a->b (with weight w), change its weigt to
//   be w+dist[a]-dist[b] (where dist is the result of step 2)
// 4. The shortest paths in the old and new graph are the same (their
//   weight result may differ, but the paths are the same)
// Note that this doesn't work well with negative cycles, but you can
// identify them before step 3 and then ignore all new weights that
// result in a negative value when executing step 3.
struct BellmanFord {
  vector<vector<ii>> G;  // ady. list with pairs (weight, dst)
  vector<ll> dist;
  int N;
  BellmanFord(int n) : G(n), N(n) {}
  void addEdge(int a, int b, ll w) { G[a].pb(mp(w, b)); }
  void run(int src) {  // O(VE)
    dist = vector<ll>(N, INF);
    dist[src] = 0;
    forn(i, N - 1) forn(j, N) if (dist[j] != INF) forall(it, G[j])
        dist[it->snd] = min(dist[it->snd], dist[j] + it->fst);
  }

  bool hasNegCycle() {
    forn(j, N) if (dist[j] != INF)
        forall(it, G[j]) if (dist[it->snd] > dist[j] + it->fst) return true;
    // inside if: all points reachable from it->snd will have -INF
    // distance. However this is not enough to identify which exact
    // nodes belong to a neg cycle, nor even which can reach a neg
    // cycle. To do so, you need to run SCC (kosaraju) and check
    // whether each SCC hasNegCycle independently. For those that
    // do hasNegCycle, then all of its nodes are part of a (not
    // necessarily simple) neg cycle.
    return false;
  }
};
```

6.4   Kruskal

```
struct Edge {
  int a, b, w;
};
bool operator<(const Edge& a, const Edge& b) { return a.w < b.w; }
// Minimun Spanning Tree in O(E log E)
ll kruskal(vector<Edge> &E, int n) {
  ll cost = 0; sort(E.begin(), E.end());
  UnionFind uf(n);
  forall(it, E) if(!uf.join(it->a, it->b))
    cost += it->w;
  return cost;
}
```

6.5   Prim

```
vector<ii> G[MAXN];
bool taken[MAXN];
priority_queue<ii, vector<ii>, greater<ii> > pq;  // min heap
void process(int v) {
  taken[v] = true;
  forall(e, G[v]) if (!taken[e->second]) pq.push(*e);
}
// Minimun Spanning Tree in O(n^2)
ll prim() {
  zero(taken);
  process(0);
  ll cost = 0;
  while (sz(pq)) {
    ii e = pq.top();
    pq.pop();
    if (!taken[e.second]) cost += e.first, process(e.second);
  }
  return cost;
}
```

6.6   Kosaraju SCC

```
struct Kosaraju {
  vector<vector<int>> G, gt;
  // nodos 0...N-1 ; componentes 0...cantcomp-1
  int N, cantcomp;
  vector<int> comp, used;
  stack<int> pila;
  Kosaraju(int n) : G(n), gt(n), N(n), comp(n) {}
  void addEdge(int a, int b) { G[a].pb(b), gt[b].pb(a); }
  void dfs1(int nodo) {
    used[nodo] = 1;
    forall(it, G[nodo]) if (!used[*it]) dfs1(*it);
    pila.push(nodo);
```

```
  }
  void dfs2(int nodo) {
    used[nodo] = 2;
    comp[nodo] = cantcomp - 1;
    forall(it, gt[nodo]) if (used[*it] != 2) dfs2(*it);
  }
  void run() {
    cantcomp = 0;
    used = vector<int>(N, 0);
    forn(i, N) if (!used[i]) dfs1(i);
    while (!pila.empty()) {
      if (used[pila.top()] != 2) {
        cantcomp++;
        dfs2(pila.top());
      }
      pila.pop();
    }
  }
};
```

6.7   2-SAT + Tarjan SCC

```
// Usage:
// 1. Create with n = number of variables (0-indexed)
// 2. Add restrictions through the existing methods, using ~X for
//    negating variable X for example.
// 3. Call satisf() to check whether there is a solution or not.
// 4. Find a valid assigment by looking at verdad[cmp[2*X]] for each
//    variable X
struct Sat2 {
  // We have a vertex representing a variable and other for its
  // negation. Every edge stored in G represents an implication.
  vector<vector<int>> G;
  // idx[i]=index assigned in the dfs
  // lw[i]=lowest index(closer from the root) reachable from i
  // verdad[cmp[2*i]]=valor de la variable i
  int N, qidx, qcmp;
  vector<int> lw, idx, cmp, verdad;
  stack<int> q;
  Sat2(int n) : G(2 * n), N(n) {}
  void tjn(int v) {
    lw[v] = idx[v] = ++qidx;
    q.push(v), cmp[v] = -2;
    forall(it, G[v]) if (!idx[*it] || cmp[*it] == -2) {
      if (!idx[*it]) tjn(*it);
      lw[v] = min(lw[v], lw[*it]);
    }
    if (lw[v] == idx[v]) {
      int x;
      do { x = q.top(), q.pop(), cmp[x] = qcmp; } while (x != v);
      verdad[qcmp] = (cmp[v ^ 1] < 0);
      qcmp++;
    }
  }
```

```
33  bool satisf() {  // O(N)
34    idx = lw = verdad = vector<int>(2 * N, 0);
35    cmp = vector<int>(2 * N, -1);
36    qidx = qcmp = 0;
37    forn(i, N) {
38      if (!idx[2 * i]) tjn(2 * i);
39      if (!idx[2 * i + 1]) tjn(2 * i + 1);
40    }
41    forn(i, N) if (cmp[2 * i] == cmp[2 * i + 1]) return false;
42    return true;
43  }
44  // a -> b, here ids are transformed to avoid negative numbers
45  void addimpl(int a, int b) {
46    a = a >= 0 ? 2 * a : 2 * (~a) + 1;
47    b = b >= 0 ? 2 * b : 2 * (~b) + 1;
48    G[a].pb(b), G[b ^ 1].pb(a ^ 1);
49  }
50  void addor(int a, int b) { addimpl(~a, b); }  // a | b = ~a -> b
51  void addeq(int a, int b) {                    // a = b, a <-> b (iff)
52    addimpl(a, b);
53    addimpl(b, a);
54  }
55  void addxor(int a, int b) { addeq(a, ~b); }  // a xor b
56  void force(int x, bool val) {                // force x to take val
57    if (val) addimpl(~x, x);
58    else addimpl(x, ~x);
59  }
60  // At most 1 true in all v
61  void atmost1(vector<int> v) {
62    int auxid = N;
63    N += sz(v);
64    G.rsz(2 * N);
65    forn(i, sz(v)) {
66      addimpl(auxid, ~v[i]);
67      if (i) {
68        addimpl(auxid, auxid - 1);
69        addimpl(v[i], auxid - 1);
70      }
71      auxid++;
72    }
73    assert(auxid == N);
74  }
75 };
```

6.8   Articulation points

```
1  int N;
2  vector<int> G[1000000];
3  // V[i]=node number(if visited), L[i]= lowest V[i] reachable from i
4  int qV, V[1000000], L[1000000], P[1000000];
5  void dfs(int v, int f) {
6    L[v] = V[v] = ++qV;
7    forall(it, G[v]) if (!V[*it]) {
8      dfs(*it, v);
```

```
9      L[v] = min(L[v], L[*it]);
10     P[v] += L[*it] >= V[v];
11    }
12    else if (*it != f) L[v] = min(L[v], V[*it]); }
13  int cantart() {  // O(n)
14    qV = 0;
15    zero(V), zero(P);
16    dfs(1, 0);
17    P[1]--;
18    int q = 0;
19    forn(i, N) if (P[i]) q++;
20    return q;
21  }
```

6.9   Biconnected components and bridges

```
1  struct Bicon {
2    vector<vector<int>> G;
3    struct edge {
4      int u, v, comp;
5      bool bridge;
6    };
7    vector<edge> ve;
8    void addEdge(int u, int v) {
9      G[u].pb(sz(ve)), G[v].pb(sz(ve));
10     ve.pb({u, v, -1, false});
11   }
12   // d[i] = dfs id
13   // b[i] = lowest id reachable from i
14   // art[i]>0 iff i is an articulation point
15   // nbc = total # of biconnected comps
16   // nart = total # of articulation points
17   vector<int> d, b, art;
18   int n, t, nbc, nart;
19   Bicon(int nn) {
20     n = nn;
21     t = nbc = nart = 0;
22     b = d = vector<int>(n, -1);
23     art = vector<int>(n, 0);
24     G = vector<vector<int>>(n);
25     ve.clear();
26   }
27   stack<int> st;
28   void dfs(int u, int pe) {  // O(n + m)
29     b[u] = d[u] = t++;
30     forall(eid, G[u]) if (*eid != pe) {
31       int v = ve[*eid].u ^ ve[*eid].v ^ u;
32       if (d[v] == -1) {
33         st.push(*eid);
34         dfs(v, *eid);
35         if (b[v] > d[u]) ve[*eid].bridge = true;  // bridge
36         if (b[v] >= d[u]) {                        // art
37           if (art[u]++ == 0) nart++;
38           int last;  // start biconnected
```

```
39            do {
40              last = st.top();
41              st.pop();
42              ve[last].comp = nbc;
43            } while (last != *eid);
44            nbc++;   // end biconnected
45          }
46          b[u] = min(b[u], b[v]);
47        } else if (d[v] < d[u]) {   // back edge
48          st.push(*eid);
49          b[u] = min(b[u], d[v]);
50        }
51      }
52    }
53    void run() { forn(i, n) if (d[i] == -1) art[i]--, dfs(i, -1); }
54    // block-cut tree (copy only if needed)
55    vector<set<int>> bctree;      // set to dedup
56    vector<int> artid;            // art nodes to tree node (-1 for !arts)
57    void buildBlockCutTree() {  // call run first!!
58      // node id: [0, nbc) -> bc, [nbc, nbc+nart) -> art
59      int ntree = nbc + nart, auxid = nbc;
60      bctree = vector<set<int>>(ntree);
61      artid = vector<int>(n, -1);
62      forn(i, n) if (art[i] > 0) {
63        forall(eid, G[i]) {   // edges always bc <-> art
64          // depending on the problem, may want
65          // to add more data on bctree edges
66          bctree[auxid].insert(ve[*eid].comp);
67          bctree[ve[*eid].comp].insert(auxid);
68        }
69        artid[i] = auxid++;
70      }
71    }
72    int getTreeIdForGraphNode(int u) {
73      if (artid[u] != -1) return artid[u];
74      if (!G[u].empty()) return ve[G[u][0]].comp;
75      return -1;   // for nodes with no neighbours in G
76    }
77  };
```

6.10   LCA + Climb

```
1  #define lg(x) (31 - __builtin_clz(x))   //=floor(log2(x))
2  // Usage: 1) Create 2) Add edges 3) Call build 4) Use
3  struct LCA {
4    int N, LOGN, ROOT;
5    // vp[k][node] holds the 2^k ancestor of node
6    // L[v] holds the level of v
7    vector<int> L;
8    vector<vector<int>> vp, G;
9    LCA(int n, int root) : N(n), LOGN(lg(n) + 1), ROOT(root), L(n), G(n) {
10     // Here you may want to replace the default from root to other
11     // value, like maybe -1.
12     vp = vector<vector<int>>(LOGN, vector<int>(n, root));
```

```
13    }
14    void addEdge(int a, int b) { G[a].pb(b), G[b].pb(a); }
15    void dfs(int node, int p, int lvl) {
16      vp[0][node] = p, L[node] = lvl;
17      forall(it, G[node]) if (*it != p) dfs(*it, node, lvl + 1);
18    }
19    void build() {
20      // Here you may also want to change the 2nd param to -1
21      dfs(ROOT, ROOT, 0);
22      forn(k, LOGN - 1) forn(i, N) vp[k + 1][i] = vp[k][vp[k][i]];
23    }
24    int climb(int a, int d) {   // O(lgn)
25      if (!d) return a;
26      dforn(i, lg(L[a]) + 1) if (1 << i <= d) a = vp[i][a], d -= 1 << i;
27      return a;
28    }
29    int lca(int a, int b) {   // O(lgn)
30      if (L[a] < L[b]) swap(a, b);
31      a = climb(a, L[a] - L[b]);
32      if (a == b) return a;
33      dforn(i, lg(L[a]) + 1) if (vp[i][a] != vp[i][b]) a = vp[i][a], b = vp[i][b
34          ];
35      return vp[0][a];
36    }
37    int dist(int a, int b) {   // returns distance between nodes
38      return L[a] + L[b] - 2 * L[lca(a, b)];
39    }
40  };
```

6.11   Virtual tree

```
1   // Usage: (VT = VirtualTree)
2   // 1- Build the LCA and use it for creating 1 VT instance
3   // 2- Call updateVT every time you want
4   // 3- Between calls of updateVT you probably want to use the tree, imp
5   // and VTroot variables from this struct to solve your problem
6   struct VirtualTree {
7     // n = #nodes full tree
8     // curt used for computing tin and tout
9     int n, curt;
10    LCA* lca;
11    vector<int> tin, tout;
12    vector<vector<ii>> tree;  // {node, dist}, only parent -> child dire
13    // imp[i] = true iff i was part of 'newv' from last time that
14    // updateVT was called (note that LCAs are not imp)
15    vector<bool> imp;
16    void dfs(int node, int p) {
17      tin[node] = curt++;
18      forall(it, lca->G[node]) if (*it != p) dfs(*it, node);
19      tout[node] = curt++;
20    }
21    VirtualTree(LCA* l) {  // must call l.build() before
22      lca = l, n = sz(l->G), lca = l, curt = 0;
23      tin.rsz(n), tout.rsz(n), tree.rsz(n), imp.rsz(n);
24      dfs(l->ROOT, l->ROOT);
25    }
26    bool isAncestor(int a, int b) { return tin[a] < tin[b] && tout[a] > tout[b];
            }
27    int VTroot = -1;  // root of the current VT
28    vector<int> v;    // list of nodes of current VT (includes LCAs)
29    void updateVT(vector<int>& newv) {  // O(sz(newv)*log)
30      assert(!newv.empty());           // this method assumes non-empty
31      auto cmp = [this](int a, int b) { return tin[a] < tin[b]; };
32      forn(i, sz(v)) tree[v[i]].clear(), imp[v[i]] = false;
33      v = newv;
34      sort(v.begin(), v.end(), cmp);
35      set<int> s;
36      forn(i, sz(v)) s.insert(v[i]), imp[v[i]] = true;
37      forn(i, sz(v) - 1) s.insert(lca->lca(v[i], v[i + 1]));
38      v.clear();
39      forall(it, s) v.pb(*it);
40      sort(v.begin(), v.end(), cmp);
41      stack<int> st;
42      forn(i, sz(v)) {
43        while (!st.empty() && !isAncestor(st.top(), v[i])) st.pop();
44        assert(i == 0 || !st.empty());
45        if (!st.empty()) tree[st.top()].pb(mp(v[i], lca->dist(st.top(), v[i])));
46        st.push(v[i]);
47      }
48      VTroot = v[0];
49    }
50  };
```

6.12   Heavy Light Decomposition

```
1   // Usage: 1. HLD(# nodes) 2. add tree edges 3. build() 4. use it
2   struct HLD {
3     vector<int> w, p, dep;   // weight,father,depth
4     vector<vector<int>> g;
5     HLD(int n) : w(n), p(n), dep(n), g(n), pos(n), head(n) {}
6     void addEdge(int a, int b) { g[a].pb(b), g[b].pb(a); }
7     void build() { p[0] = -1, dep[0] = 0, dfs1(0), curpos = 0, hld(0, -1); }
8     void dfs1(int x) {
9       w[x] = 1;
10      for (int y : g[x]) if (y != p[x]) {
11        p[y] = x, dep[y] = dep[x] + 1, dfs1(y);
12        w[x] += w[y];
13      }
14    }
15    int curpos;
16    vector<int> pos, head;
17    void hld(int x, int c) {
18      if (c < 0) c = x;
19      pos[x] = curpos++, head[x] = c;
20      int mx = -1;
21      for (int y : g[x]) if (y != p[x] && (mx < 0 || w[mx] < w[y])) mx = y;
22      if (mx >= 0) hld(mx, c);
23      for (int y : g[x]) if (y != mx && y != p[x]) hld(y, -1);
24    }
25    // Here ST is segtree static/dynamic/lazy or other DS according to problem
26    tipo query(int x, int y, ST& st) {  // ST tipo
27      tipo r = neutro;
28      while (head[x] != head[y]) {
29        if (dep[head[x]] > dep[head[y]]) swap(x, y);
30        r = oper(r, st.get(pos[head[y]], pos[y] + 1));  // ST oper
31        y = p[head[y]];
32      }
33      if (dep[x] > dep[y]) swap(x, y);         // now x is lca
34      r = oper(r, st.get(pos[x], pos[y] + 1));  // ST oper
35      return r;
36    }
37  };
38  // for point updates: st.set(pos[x], v) (x = node, v = new value)
39  // for lazy range updates: something similar to the query method
40  // for queries on edges: - assign values of edges to "child" node
41  //                       - change pos[x] to pos[x]+1 in query (line 34)
```

6.13   Centroid Decomposition

```cpp
// Usage: 1. Centroid(# nodes), 2. add tree edges, 3. build(), 4. use it
struct Centroid {
  vector<vector<int>> g;
  vector<int> vp, vsz;
  vector<bool> taken;
  Centroid(int n) : g(n), vp(n), vsz(n), taken(n) {}
  void addEdge(int a, int b) { g[a].pb(b), g[b].pb(a); }
  void build() { centroid(0, -1, -1); }  // O(nlogn)
  int dfs(int node, int p) {
    vsz[node] = 1;
    forall(it, g[node]) if (*it != p && !taken[*it])
      vsz[node] += dfs(*it, node);
    return vsz[node];
  }
  void centroid(int node, int p, int cursz) {
    if (cursz == -1) cursz = dfs(node, -1);
    forall(it, g[node]) if (!taken[*it] && vsz[*it] > cursz / 2) {
      vsz[node] = 0, centroid(*it, p, cursz);
      return;
    }
    taken[node] = true, vp[node] = p;
    // do something using node as centroid
    forall(it, g[node]) if (!taken[*it]) centroid(*it, node, -1);
  }
};
```

6.14   Tree Reroot

```cpp
struct Edge {
  int u, v;  // maybe add more data, depending on the problem
};
// USAGE:
// 1- define all the logic in SubtreeData
// 2- create a reroot and add all the edges
// 3- call Reroot.run()
struct SubtreeData {
  // Define here what data you need for each subtree
  SubtreeData() {}  // just empty
  SubtreeData(int node) {
    // Initialize the data here as if this new subtree
    // has size 1, and its only node is 'node'
  }
  void merge(Edge* e, SubtreeData& s) {
    // Modify this subtree's data to reflect that 's' is being
    // merged into 'this' through the edge 'e'.
    // When e == NULL, then no edge is present, but then, 'this'
    // and 's' have THE SAME ROOT (be CAREFUL with this).
    // These 2 subtrees don't have any other shared nodes nor edges.
  }
};
struct Reroot {
```

```cpp
  int N;  // # of nodes
  // vresult[i] = SubtreeData for the tree where i is the root
  // this should be what you need as result
  vector<SubtreeData> vresult, vs;
  vector<Edge> ve;
  vector<vector<int>> g;  // the tree as a bidirectional graph
  Reroot(int n) : N(n), vresult(n), vs(n), ve(0), g(n) {}
  void addEdge(Edge e) {  // will be added in both ways
    g[e.u].pb(sz(ve));
    g[e.v].pb(sz(ve));
    ve.pb(e);
  }
  void dfs1(int node, int p) {
    vs[node] = SubtreeData(node);
    forall(e, g[node]) {
      int nxt = node ^ ve[*e].u ^ ve[*e].v;
      if (nxt == p) continue;
      dfs1(nxt, node);
      vs[node].merge(&ve[*e], vs[nxt]);
    }
  }
  void dfs2(int node, int p, SubtreeData fromp) {
    vector<SubtreeData> vsuf(sz(g[node]) + 1);
    int pos = sz(g[node]);
    SubtreeData pref = vsuf[pos] = SubtreeData(node);
    vresult[node] = vs[node];
    dforall(e, g[node]) {  // dforall = forall in reverse
      pos--;
      vsuf[pos] = vsuf[pos + 1];
      int nxt = node ^ ve[*e].u ^ ve[*e].v;
      if (nxt == p) {
        pref.merge(&ve[*e], fromp);
        vresult[node].merge(&ve[*e], fromp);
        continue;
      }
      vsuf[pos].merge(&ve[*e], vs[nxt]);
    }
    assert(pos == 0);
    forall(e, g[node]) {
      pos++;
      int nxt = node ^ ve[*e].u ^ ve[*e].v;
      if (nxt == p) continue;
      SubtreeData aux = pref;
      aux.merge(NULL, vsuf[pos]);
      dfs2(nxt, node, aux);
      pref.merge(&ve[*e], vs[nxt]);
    }
  }
  void run() {
    dfs1(0, 0);
    dfs2(0, 0, SubtreeData());
  }
};
```

6.15   Diameter of a tree

```cpp
vector<int> G[MAXN];
int n, m, p[MAXN], d[MAXN], d2[MAXN];
int bfs(int r, int* d) {
  queue<int> q;
  d[r] = 0, q.push(r);
  int v;
  while (sz(q)) {
    v = q.front();
    q.pop();
    forall(it, G[v]) if (d[*it] == -1) {
      d[*it] = d[v] + 1, p[*it] = v, q.push(*it);
    }
  }
  return v;  // ultimo nodo visitado
}
vector<int> diams;
vector<ii> centros;
void diametros() {
  memset(d, -1, sizeof(d));
  memset(d2, -1, sizeof(d2));
  diams.clear(), centros.clear();
  forn(i, n) if (d[i] == -1) {
    int v, c;
    c = v = bfs(bfs(i, d2), d);
    forn(_, d[v] / 2) c = p[c];
    diams.pb(d[v]);
    if (d[v] & 1) centros.pb(ii(c, p[c]));
    else centros.pb(ii(c, c));
  }
}
```

6.16   Euler path or cycle

```cpp
// Be careful with nodes with degree 0 when solving your problem, the
// comments below assume that there are no nodes with degree 0.
//
// Euler [path/cycle] exists in a bidirectional graph iff the graph is
// connected and at most [2/0] nodes have odd degree. The path must
// start from an odd degree vertex when there are 2.
//
// Euler [path/cycle] exists in a directed graph iff the graph is
// [connected when making edges bidirectional / a single SCC], and
// at most [1/0] node have indg - outdg = 1, at most [1/0] node have
// outdg - indg = 1, all the other nodes have indg = outdg. The path
// must start from the node with outdg - indg = 1, when there is one.
//
// Directed version (uncomment commented code for undirected)
struct edge {
  int y;
  // list<edge>::iterator rev;
  edge(int yy) : y(yy) {}
```

```cpp
};
struct EulerPath {
  vector<list<edge>> g;
  EulerPath(int n) : g(n) {}
  void addEdge(int a, int b) {
    g[a].push_front(edge(b));
    // auto ia = g[a].begin();
    // g[b].push_front(edge(a));
    // auto ib = g[b].begin();
    // ia->rev=ib, ib->rev=ia;
  }
  vector<int> p;
  void go(int x) {
    while (sz(g[x])) {
      int y = g[x].front().y;
      // g[y].erase(g[x].front().rev);
      g[x].pop_front();
      go(y);
    }
    p.push_back(x);
  }
  vector<int> getPath(int x) {  // get a path that starts from x
    // you must check that a path exists from x before calling get_path!
    p.clear(), go(x);
    reverse(p.begin(), p.end());
    return p;
  }
};
```

6.17   Dynamic Connectivity

```cpp
struct UnionFind {
  int n, comp;
  vector<int> pre, si, c;
  UnionFind(int n = 0) : n(n), comp(n), pre(n), si(n, 1) {
    forn(i, n) pre[i] = i;
  }
  int find(int u) { return u == pre[u] ? u : find(pre[u]); }
  bool merge(int u, int v) {
    if ((u = find(u)) == (v = find(v))) return false;
    if (si[u] < si[v]) swap(u, v);
    si[u] += si[v], pre[v] = u, comp--, c.pb(v);
    return true;
  }
  int snap() { return sz(c); }
  void rollback(int snap) {
    while (sz(c) > snap) {
      int v = c.back();
      c.pop_back();
      si[pre[v]] -= si[v], pre[v] = v, comp++;
    }
  }
};
enum { ADD, DEL, QUERY };
```

```
24  struct Query {
25    int type, u, v;
26  };
27  struct DynCon {  // bidirectional graphs; create vble as DynCon name(
        cant_nodos)
28    vector<Query> q;
29    UnionFind dsu;
30    vector<int> match, res;
31    // se puede no usar cuando hay identificador para cada arista (mejora poco)
32    map<ii, int> last;
33    DynCon(int n = 0) : dsu(n) {}
34    void add(int u, int v)  // to add an edge
35    {
36      if (u > v) swap(u, v);
37      q.pb((Query){ADD, u, v}), match.pb(-1);
38      last[ii(u, v)] = sz(q) - 1;
39    }
40    void remove(int u, int v)  // to remove an edge
41    {
42      if (u > v) swap(u, v);
43      q.pb((Query){DEL, u, v});
44      int prev = last[ii(u, v)];
45      match[prev] = sz(q) - 1;
46      match.pb(prev);
47    }
48    void query()  // to add a question (query) type of query
49    {
50      q.pb((Query){QUERY, -1, -1}), match.pb(-1);
51    }
52    void process()  // call this to process queries in the order of q
53    {
54      forn(i, sz(q)) if (q[i].type == ADD && match[i] == -1) match[i] = sz(q);
55      go(0, sz(q));
56    }
57    void go(int l, int r) {
58      if (l + 1 == r) {
59        if (q[l].type == QUERY)  // Aqui responder la query usando el dsu!
60          res.pb(dsu.comp);       // aqui query=cantidad de componentes conexas
61        return;
62      }
63      int s = dsu.snap(), m = (l + r) / 2;
64      forr(i, m, r) if (match[i] != -1 && match[i] < l) dsu.merge(q[i].u, q[i].v
          );
65      go(l, m);
66      dsu.rollback(s);
67      s = dsu.snap();
68      forr(i, l, m) if (match[i] != -1 && match[i] >= r)
69          dsu.merge(q[i].u, q[i].v);
70      go(m, r);
71      dsu.rollback(s);
72    }
73  };
```

7   Flow

7.1   Dinic

```
1   struct Edge {
2     int u, v;
3     ll cap, flow;
4     Edge() {}
5     Edge(int uu, int vv, ll c) : u(uu), v(vv), cap(c), flow(0) {}
6   };
7   struct Dinic {
8     int N;
9     vector<Edge> E;
10    vector<vector<int>> g;
11    vector<int> d, pt;
12    Dinic(int n) : N(n), g(n), d(n), pt(n) {}  // clear and init
13    void addEdge(int u, int v, ll cap) {
14      if (u != v) {
15        g[u].pb(sz(E));
16        E.pb({u, v, cap});
17        g[v].pb(sz(E));
18        E.pb({v, u, 0});
19      }
20    }
21    bool BFS(int S, int T) {
22      queue<int> q({S});
23      fill(d.begin(), d.end(), N + 1);
24      d[S] = 0;
25      while (!q.empty()) {
26        int u = q.front();
27        q.pop();
28        if (u == T) break;
29        for (int k : g[u]) {
30          Edge& e = E[k];
31          if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
32            d[e.v] = d[e.u] + 1;
33            q.push(e.v);
34          }
35        }
36      }
37      return d[T] != N + 1;
38    }
39    ll DFS(int u, int T, ll flow = -1) {
40      if (u == T || flow == 0) return flow;
41      for (int& i = pt[u]; i < sz(g[u]); ++i) {
42        Edge& e = E[g[u][i]];
43        Edge& oe = E[g[u][i] ^ 1];
44        if (d[e.v] == d[e.u] + 1) {
45          ll amt = e.cap - e.flow;
46          if (flow != -1 && amt > flow) amt = flow;
47          if (ll pushed = DFS(e.v, T, amt)) {
48            e.flow += pushed;
49            oe.flow -= pushed;
50            return pushed;
51          }
```

```
52          }
53        }
54        return 0;
55      }
56      ll maxFlow(int S, int T) {  // O(V^2*E), unit nets: O(sqrt(V)*E)
57        ll total = 0;
58        while (BFS(S, T)) {
59          fill(pt.begin(), pt.end(), 0);
60          while (ll flow = DFS(S, T)) total += flow;
61        }
62        return total;
63      }
64  };
65  // Dinic wrapper to allow setting demands of min flow on edges
66  // If an edge with a min flow demand is part of a cycle, then the result
67  // is not guaranteed to be correct, it could result in false positives
68  struct DinicWithDemands {
69      int N;
70      vector<pair<Edge, ll>> E;  // (normal dinic edge, min flow)
71      Dinic dinic;
72      DinicWithDemands(int n) : N(n), E(0), dinic(n + 2) {}
73      void addEdge(int u, int v, ll cap, ll minFlow) {
74        assert(minFlow <= cap);
75        if (u != v) E.pb(mp(Edge(u, v, cap), minFlow));
76      }
77      ll maxFlow(int S, int T) {  // Same complexity as normal Dinic
78        int SRC = N, SNK = N + 1;
79        ll minFlowSum = 0;
80        forall(e, E) {  // force the min flow
81          minFlowSum += e->snd;
82          dinic.addEdge(SRC, e->fst.v, e->snd);
83          dinic.addEdge(e->fst.u, SNK, e->snd);
84          dinic.addEdge(e->fst.u, e->fst.v, e->fst.cap - e->snd);
85        }
86        dinic.addEdge(T, S, INF);  // INF >= max possible flow
87        ll flow = dinic.maxFlow(SRC, SNK);
88        if (flow < minFlowSum) return -1;  // no valid flow exists
89        assert(flow == minFlowSum);
90        // Now go back to the original network, to a valid
91        // state where all min flow values are satisfied.
92        forn(i, sz(E)) {
93          forn(j, 4) {
94            assert(j % 2 || dinic.E[6 * i + j].flow == E[i].snd);
95            dinic.E[6 * i + j].cap = dinic.E[6 * i + j].flow = 0;
96          }
97          dinic.E[6 * i + 4].cap += E[i].snd;
98          dinic.E[6 * i + 4].flow += E[i].snd;
99          // don't change edge [6*i+5] to keep forcing the mins
100       }
101       forn(i, 2) dinic.E[6 * sz(E) + i].cap = dinic.E[6 * sz(E) + i].flow = 0;
102       // Just finish the maxFlow now
103       dinic.maxFlow(S, T);
104       flow = 0;  // get the result manually
105       forall(e, dinic.g[S]) flow += dinic.E[*e].flow;
106       return flow;
107     }
```

```
108 };
```

7.2   Min cost - Max flow

```
1   typedef ll tf;
2   typedef ll tc;
3   const tf INF_FLOW = 1e14;
4   const tc INF_COST = 1e14;
5   struct edge {
6     int u, v;
7     tf cap, flow;
8     tc cost;
9     tf rem() { return cap - flow; }
10  };
11  struct MCMF {
12    vector<edge> e;
13    vector<vector<int>> g;
14    vector<tf> vcap;
15    vector<tc> dist;
16    vector<int> pre;
17    tc minCost;
18    tf maxFlow;
19    // tf wantedFlow; // Use it for fixed flow instead of max flow
20    MCMF(int n) : g(n), vcap(n), dist(n), pre(n) {}
21    void addEdge(int u, int v, tf cap, tc cost) {
22      g[u].pb(sz(e)), e.pb({u, v, cap, 0, cost});
23      g[v].pb(sz(e)), e.pb({v, u, 0, 0, -cost});
24    }
25    // O(n*m * min(flow, n*m)), sometimes faster in practice
26    void run(int s, int t) {
27      vector<bool> inq(sz(g));
28      maxFlow = minCost = 0;  // result will be in these variables
29      while (1) {
30        fill(vcap.begin(), vcap.end(), 0), vcap[s] = INF_FLOW;
31        fill(dist.begin(), dist.end(), INF_COST), dist[s] = 0;
32        fill(pre.begin(), pre.end(), -1), pre[s] = 0;
33        queue<int> q;
34        q.push(s), inq[s] = true;
35        while (sz(q)) {  // Fast bellman-ford
36          int u = q.front();
37          q.pop(), inq[u] = false;
38          for (auto eid : g[u]) {
39            edge& E = e[eid];
40            if (E.rem() && dist[E.v] > dist[u] + E.cost) {
41              dist[E.v] = dist[u] + E.cost;
42              pre[E.v] = eid;
43              vcap[E.v] = min(vcap[u], E.rem());
44              if (!inq[E.v]) q.push(E.v), inq[E.v] = true;
45            }
46          }
47        }
48        if (pre[t] == -1) break;
49        tf flow = vcap[t];
50        // flow = min(flow, wantedFlow - maxFlow); //For fixed flow
```

```
51        maxFlow += flow;
52        minCost += flow * dist[t];
53        for (int v = t; v != s; v = e[pre[v]].u) {
54          e[pre[v]].flow += flow;
55          e[pre[v] ^ 1].flow -= flow;
56        }
57        // if(maxFlow == wantedFlow) break; //For fixed flow
58      }
59    }
60 };
```

7.3   Matching - Hopcroft Karp

```
1  struct HopcroftKarp {  // [0,n)->[0,m) (ids independent in each side)
2    int n, m;
3    vector<vector<int>> g;
4    vector<int> mt, mt2, ds;
5    HopcroftKarp(int nn, int mm) : n(nn), m(mm), g(n) {}
6    void add(int a, int b) { g[a].pb(b); }
7    bool bfs() {
8      queue<int> q;
9      ds = vector<int>(n, -1);
10     forn(i, n) if (mt2[i] < 0) ds[i] = 0, q.push(i);
11     bool r = false;
12     while (!q.empty()) {
13       int x = q.front();
14       q.pop();
15       for (int y : g[x]) {
16         if (mt[y] >= 0 && ds[mt[y]] < 0) {
17           ds[mt[y]] = ds[x] + 1, q.push(mt[y]);
18         } else if (mt[y] < 0) r = true;
19       }
20     }
21     return r;
22   }
23   bool dfs(int x) {
24     for (int y : g[x]) {
25       if (mt[y] < 0 || ds[mt[y]] == ds[x] + 1 && dfs(mt[y])) {
26         mt[y] = x, mt2[x] = y;
27         return true;
28       }
29     }
30     ds[x] = 1 << 30;
31     return false;
32   }
33   int mm() {  // O(sqrt(V)*E)
34     int r = 0;
35     mt = vector<int>(m, -1);
36     mt2 = vector<int>(n, -1);
37     while (bfs()) forn(i, n) if (mt2[i] < 0) r += dfs(i);
38     return r;
39   }
40 };
```

7.4   Hungarian

```
1  typedef long double td;
2  typedef vector<int> vi;
3  typedef vector<td> vd;
4  const td INF = 1e100;  // for maximum set INF to 0, and negate costs
5  bool zz(td x) { return abs(x) < 1e-9; }  // change to x==0, for ints/ll
6  struct Hungarian {
7    int n;
8    vector<vd> cs;
9    vi L, R;
10   Hungarian(int N, int M) : n(max(N, M)), cs(n, vd(n)), L(n), R(n) {
11     forn(x, N) forn(y, M) cs[x][y] = INF;
12   }
13   void set(int x, int y, td c) { cs[x][y] = c; }
14   td assign() {  // O(n^3)
15     int mat = 0;
16     vd ds(n), u(n), v(n);
17     vi dad(n), sn(n);
18     forn(i, n) u[i] = *min_element(cs[i].begin(), cs[i].end());
19     forn(j, n) {
20       v[j] = cs[0][j] - u[0];
21       forr(i, 1, n) v[j] = min(v[j], cs[i][j] - u[i]);
22     }
23     L = R = vi(n, -1);
24     forn(i, n) forn(j, n) if (R[j] == -1 && zz(cs[i][j] - u[i] - v[j])) {
25       L[i] = j, R[j] = i, mat++;
26       break;
27     }
28     for (; mat < n; mat++) {
29       int s = 0, j = 0, i;
30       while (L[s] != -1) s++;
31       fill(dad.begin(), dad.end(), -1);
32       fill(sn.begin(), sn.end(), 0);
33       forn(k, n) ds[k] = cs[s][k] - u[s] - v[k];
34       while (1) {
35         j = -1;
36         forn(k, n) if (!sn[k] && (j == -1 || ds[k] < ds[j])) j = k;
37         sn[j] = 1, i = R[j];
38         if (i == -1) break;
39         forn(k, n) if (!sn[k]) {
40           td new_ds = ds[j] + cs[i][k] - u[i] - v[k];
41           if (ds[k] > new_ds) ds[k] = new_ds, dad[k] = j;
42         }
43       }
44       forn(k, n) if (k != j && sn[k]) {
45         td w = ds[k] - ds[j];
46         v[k] += w, u[R[k]] -= w;
47       }
48       u[s] += ds[j];
49       while (dad[j] >= 0) {
50         int d = dad[j];
51         R[j] = R[d], L[R[j]] = j, j = d;
52       }
53       R[j] = s, L[s] = j;
54     }
```

```
55       td ret = 0;
56       forn(i, n) ret += cs[i][L[i]];
57       return ret;
58     }
59 };
```

7.5   Edmond's Karp

```
1  struct EdmondsKarp {
2    int N;
3    vector<unordered_map<int, ll>> g;
4    vector<int> p;
5    ll f;
6    EdmondsKarp(int n) : N(n), g(n), p(n) {}
7    void addEdge(int a, int b, int w) { g[a][b] = w; }
8    void augment(int v, int SRC, ll minE) {
9      if (v == SRC) f = minE;
10     else if (p[v] != -1) {
11       augment(p[v], SRC, min(minE, g[p[v]][v]));
12       g[p[v]][v] -= f, g[v][p[v]] += f;
13     }
14   }
15   ll maxflow(int SRC, int SNK) {  // O(min(VE^2,Mf*E))
16     ll ret = 0;
17     do {
18       queue<int> q;
19       q.push(SRC);
20       fill(p.begin(), p.end(), -1);
21       f = 0;
22       while (sz(q)) {
23         int node = q.front();
24         q.pop();
25         if (node == SNK) break;
26         forall(it, g[node]) if (it->snd > 0 && p[it->fst] == -1) {
27           q.push(it->fst), p[it->fst] = node;
28         }
29       }
30       augment(SNK, SRC, INF);  // INF > max possible flow
31       ret += f;
32     } while (f);
33     return ret;
34   }
35 };
```

7.6   Matching

```
1  vector<int> g[MAXN];   // [0,n)->[0,m)
2  int n, m;
3  int mat[MAXM];
4  bool vis[MAXN];
5  int match(int x) {
6    if (vis[x]) return 0;
7    vis[x] = true;
8    for (int y : g[x])
9      if (mat[y] < 0 || match(mat[y])) {
10       mat[y] = x;
11       return 1;
12     }
13   return 0;
14 }
15 vector<pair<int, int> > max_matching() {  // O(V^2 * E)
16   vector<pair<int, int> > r;
17   memset(mat, -1, sizeof(mat));
18   forn(i, n) memset(vis, false, sizeof(vis)), match(i);
19   forn(i, m) if (mat[i] >= 0) r.pb({mat[i], i});
20   return r;
21 }
```

7.7   Min Cut

```
1  // Suponemos un grafo con el formato definido en Push relabel
2  bitset<MAX_V> type, used;  // reset this
3  void dfs1(int nodo) {
4    type.set(nodo);
5    forall(it, G[nodo]) if (!type[it->fst] && it->snd > 0) dfs1(it->fst);
6  }
7  void dfs2(int nodo) {
8    used.set(nodo);
9    forall(it, G[nodo]) {
10     if (!type[it->fst]) {
11       // edge nodo -> (it->fst) pertenece al min_cut
12       // y su peso original era: it->snd + G[it->fst][nodo]
13       // si no existia arista original al reves
14     } else if (!used[it->fst]) dfs2(it->fst);
15   }
16 }
17 void minCut()  // antes correr algun maxflow()
18 {
19   dfs1(SRC);
20   dfs2(SRC);
21   return;
22 }
```

7.8   Push Relabel

```cpp
#define MAX_V 1000
int N;  // valid nodes are [0...N-1]
#define INF 1e9
// special nodes
#define SRC 0
#define SNK 1
map<int, int> G[MAX_V];  // limpiar esto -- unordered_map mejora
// To add an edge use
#define add(a, b, w) G[a][b] = w
ll excess[MAX_V];
int height[MAX_V], active[MAX_V], cuenta[2 * MAX_V + 1];
queue<int> Q;

void enqueue(int v) {
  if (!active[v] && excess[v] > 0) active[v] = true, Q.push(v);
}
void push(int a, int b) {
  int amt = min(excess[a], ll(G[a][b]));
  if (height[a] <= height[b] || amt == 0) return;
  G[a][b] -= amt, G[b][a] += amt;
  excess[b] += amt, excess[a] -= amt;
  enqueue(b);
}
void gap(int k) {
  forn(v, N) {
    if (height[v] < k) continue;
    cuenta[height[v]]--;
    height[v] = max(height[v], N + 1);
    cuenta[height[v]]++;
    enqueue(v);
  }
}
void relabel(int v) {
  cuenta[height[v]]--;
  height[v] = 2 * N;
  forall(it, G[v]) if (it->snd) height[v] = min(height[v], height[it->fst] +
      1);
  cuenta[height[v]]++;
  enqueue(v);
}
ll maxflow()  // O(V^3)
{
  zero(height), zero(active), zero(cuenta), zero(excess);
  cuenta[0] = N - 1;
  cuenta[N] = 1;
  height[SRC] = N;
  active[SRC] = active[SNK] = true;
  forall(it, G[SRC]) {
    excess[SRC] += it->snd;
    push(SRC, it->fst);
  }
  while (sz(Q)) {
    int v = Q.front();
    Q.pop();
    active[v] = false;
    forall(it, G[v]) push(v, it->fst);
    if (excess[v] > 0) cuenta[height[v]] == 1 ? gap(height[v]) : relabel(v);
  }
  ll mf = 0;
  forall(it, G[SRC]) mf += G[it->fst][SRC];
  return mf;
}
```

8   Other algorithms

8.1   Longest Increasing Subsecuence

```cpp
// Change comparisons and binary search for non-increasing
// Given an array, paint it in the least number of colors so that each
// color turns to a non-increasing subsequence. Solution: Min number of
// colors=Length of the longest increasing subsequence
struct lis {
  T INF;
  int n; vector<T> a; // secuencia y su longitud
  vector<pair<T,int>> d; // d[i]=ultimo valor de la subsecuencia de tamanio i
  vector<int> p; // padres
  vector<T> ret; // respuesta

  lis(T INF_, vector<T> &a_) {
    n = sz(a_);
    INF = INF_;
    a = a_;
    d.resize(n+1);
    p.resize(n+1);
  }
  void rec(int i) {
    if(i == -1) return;
    ret.push_back(a[i]);
    rec(p[i]);
  }
  int run() {
    d[0] = {-INF,-1};
    forn(i,n) d[i+1] = {INF, -1};
    forn(i,n) {
      int j = int(upper_bound(d.begin(), d.end(), mp(a[i], n))-d.begin());
      if(d[j-1].fst<a[i] && a[i]<d[j].fst) {
        p[i] = d[j-1].snd;
        d[j] = {a[i], i};
      }
    }
    ret.clear();
    dforn(i, n+1) if(d[i].fst!=INF) {
      rec(d[i].snd);  // reconstruir
      reverse(ret.begin(), ret.end());
      return i;  // longitud
    }
    return 0;
  }
};
```

8.2   Mo's

```cpp
// Commented code should be used if updates are needed
int n, sq, nq;  // array size, sqrt(array size), #queries
struct Qu {       //[l, r)
  int l, r, id;
```

```cpp
  // int upds; // # of updates before this query
};
Qu qs[MAXN];
ll ans[MAXN];  // ans[i] = answer to ith query
// struct Upd{
//   int p, v, prev; // pos, new_val, prev_val
// };
// Upd vupd[MAXN];

// Without updates
bool qcomp(const Qu& a, const Qu& b) {
  if (a.l / sq != b.l / sq) return a.l < b.l;
  return (a.l / sq) & 1 ? a.r < b.r : a.r > b.r;
}

// With updates
// bool qcomp(const Qu &a, const Qu &b){
//   if(a.l/sq != b.l/sq) return a.l<b.l;
//   if(a.r/sq != b.r/sq) return a.r<b.r;
//   return a.upds < b.upds;
// }

// Without updates: O(n^2/sq + q*sq)
// with sq = sqrt(n): O(n*sqrt(n) + q*sqrt(n))
// with sq = n/sqrt(q): O(n*sqrt(q))
//
// With updates: O(sq*q + q*n^2/sq^2)
// with sq = n^(2/3): O(q*n^(2/3))
// with sq = (2*n^2)^(1/3) may improve a bit
void mos() {
  forn(i, nq) qs[i].id = i;
  sq = sqrt(n) + .5;  // without updates
  // sq=pow(n, 2/3.0)+.5; // with updates
  sort(qs, qs + nq, qcomp);
  int l = 0, r = 0;
  init();
  forn(i, nq) {
    Qu q = qs[i];
    while (l > q.l) add(--l);
    while (r < q.r) add(r++);
    while (l < q.l) remove(l++);
    while (r > q.r) remove(--r);
    // while(upds<q.upds){
    //   if(vupd[upds].p >= l && vupd[upds].p < r) remove(vupd[upds].p);
    //   v[vupd[upds].p] = vupd[upds].v; // do update
    //   if(vupd[upds].p >= l && vupd[upds].p < r) add(vupd[upds].p);
    //   upds++;
    // }
    // while(upds>q.upds){
    //   upds--;
    //   if(vupd[upds].p >= l && vupd[upds].p < r) remove(vupd[upds].p);
    //   v[vupd[upds].p] = vupd[upds].prev; // undo update
    //   if(vupd[upds].p >= l && vupd[upds].p < r) add(vupd[upds].p);
    // }
    ans[q.id] = get_ans();
  }
```

```
61 }
```

9   Juegos

9.1   Nim Game

Juego en el que hay N pilas, con objetos. Cada jugador debe sacar al menos un objeto de una pila. GANA el jugador que saca el último objeto.

$$P_0 \oplus P_1 \oplus ... \oplus P_n = R$$

Si $R \neq 0$ gana el jugador 1.

9.1.1   Misere Game

Es un juego con las mismas reglas que Nim, pero PIERDE el que saca el último objeto. Entonces teniendo el resultado de la suma R, y si todas las pilas tienen 1 solo objeto $todos1=true$, podemos decir que el jugador2 GANA si:

$$(R=0)\&\neg todos1 \| (R \neq 0)\& todos1$$

9.2   Ajedrez

9.2.1   Non-Attacking N Queen

Utiliza: `<algorithm>`

Notas: todo es $O(!N \cdot N^2)$.

```cpp
#define NQUEEN 8
#define abs(x)  ((x)<0?(-(x)):(x))

int board[NQUEEN];
void inline init(){for(int i=0;i<NQUEEN;++i)board[i]=i;}
bool check(){
    for(int i=0;i<NQUEEN;++i)
        for(int j=i+1;i<NQUEEN;++j)
            if(abs(i-j)==abs(board[i]-board[j]))
                return false;
    return true;
}
//en main
init();
do{
    if(check()){
        //process solution
    }
}while(next_permutation(board,board+NQUEEN));
```

9.3   Green Hackenbush

```cpp
// A two-player game played on an undirected graph where some nodes
// are connected to the ground. On each turn, a player removes an edge.
// If this removal splits the graph into two components, any component
// that is not connected to the ground is removed. A player loses the game
// if it's impossible to make a move.
struct green_hackenbush {
  vector<vector<int>> g;
  vector<int> tin, low, gr;
  int t, root, ans;
  green_hackenbush(int n) {
    t = 0, root = -1, ans = 0;
    g.resize(n); gr.resize(n);
    tin.resize(n); low.resize(n);
  }
  // make u a node in the ground
  void ground(int u) {
    gr[u] = 1; if(root == -1) root = u;
  }
  // call first ground() if u or v are in the ground
  void add_edge(int u, int v) {
    if(gr[u]) u = root;
    if(gr[v]) v = root;
    if(u == v) { ans ^= 1; return; }
    g[u].pb(v); g[v].pb(u);
  }
  int solve(int u, int d) {
    tin[u] = low[u] = ++t;
    int ret = 0;
    forn(i,sz(g[u])) {
      int v = g[u][i];
      if(v == d) continue;
      if(tin[v] == 0) {
        int retv = solve(v,u);
        low[u] = min(low[u],low[v]);
        if(low[v] >  tin[u]) ret ^= (1+retv)^1;
        else ret ^= retv;
      }else low[u] = min(low[u], tin[v]);
    }
    forn(i,sz(g[u])) {
      int v = g[u][i];
      if(v != d && tin[u] <= tin[v]) ret ^= 1;
    }
    return ret;
  }
  int solve() {
    return root == -1? 0 : ans^solve(root,-1);
  }
};
```

10   Utils

10.1   Compile C++20 with g++

```
g++ -std=c++20 {file} -o {filename}
Para Geany:
compile: g++ -DANARAP -std=c++20 -g -O2 -Wconversion -Wshadow -Wall -Wextra -c
    "%f"
build:   g++ -DANARAP -std=c++20 -g -O2 -Wconversion -Wshadow -Wall -Wextra -o
    "%e" "%f"
```

10.2   C++ utils mix

```cpp
// 1- Random number generator (mt19937_64 for 64-bits version)
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
// usage
rng()%k // random value [0,k)
shuffle(v.begin(), v.end(), rng); // vector random shuffle

// 2- Pragma
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")

// 3- Custom comparator for set/map
struct comp {
  bool operator()(const double& a, const double& b) const {
    return a+EPS<b;}
};
set<double,comp> w; // or map<double,int,comp>

// 4- Iterate over non empty subsets of bitmask
for(int s=m;s;s=(s-1)&m) // Decreasing order
for (int s=0;s=s-m&m;)   // Increasing order

// 5- Other bits operations
// Return the numbers the numbers of 1-bit in x
int __builtin_popcount (unsigned int x)
// Returns the number of trailing 0-bits in x. x=0 is undefined.
int __builtin_ctz (unsigned int x)
// Returns the number of leading 0-bits in x. x=0 is undefined.
int __builtin_clz (unsigned int x)
// x of type long long just add 'll' at the end of the function.
int __builtin_popcountll (unsigned long long x)
// Get the value of the least significant bit that is one.
v=(x&(-x))

// 6- Comparing floats
const double EPS = 1e-9 // usually correct, but not always
abs(x-y) < EPS // use this instead of x == y
x > y + EPS // use this instead of x > y (EPS "against" comparison)
x > y - EPS // use this instead of x >= y (EPS "in favor" of comparison)

// 7- string stream, convert types easily
```

```cpp
string int_to_str(int x) {
  stringstream ss;
  ss << x;
  string ret;
  ss >> ret;
  return ret;
}

// 8- Output
cout << setprecision(2) << fixed; // print floats with 2 decimal digits
cout << setfill(' ') << setw(3) << 2 << endl; // add spaces to the left

// 9- Input
string line;
getline(cin, line); // read whole line

cin >> noskipws; // make cin stop skipping white spaces
cin >> skipws; // make cin start skipping white spaces (on by default)

inline void Scanf(int& a) { // sometimes faster, only for positive integers
  char c = 0;
  while(c<33)  c = getc(stdin);
  a = 0;
  while(c>33)  a = a*10 + c - '0', c = getc(stdin);
}

// 10- Type limits
#include <limits>
numeric_limits<T>
  ::max()
  ::min()
  ::epsilon()
// double inf
const double DINF=numeric_limits<double>::infinity();

// 11- Bitset trick
bs._Find_next(idx) // This function returns first set bit after index idx. Ex:
      bitset<10> bs, bs[1] = 1, bs[3] = 1; bs.Find_next(0) => 1, bs._Find_next
      (1) => 3
```

10.3   Python example

```python
import sys, math
input = sys.stdin.readline

############ ---- Input Functions ---- ############
def inp():
    return(int(input()))
def inlt():
    return(list(map(int,input().split())))
def insr():
    s = input()
    return(list(s[:len(s) - 1]))
def invr():
    return(map(float,input().split()))


n, k = inlt()
intpart = 0
while intpart*intpart <= n:
  intpart += 1
intpart -= 1

if(k == 0):
  print(intpart)
else:
  L = 0
  R = 10**k-1
  aux = 10**k

  while(L < R):
    M = (L+R+1)//2
    if intpart**2 * aux**2 + M**2 + 2*intpart*M*aux <= n * aux**2:
      L = M
    else:
      R = M-1

  decpart = str(L)
  while(len(decpart) < k):
    decpart = '0'+decpart
  print(f"{intpart}.{decpart}")
```

10.4   Test generator

```python
# usage: (note that test_generator.py refers to this file)
# 1. Modify the code below to generate the tests you want to use
# 2. Compile the 2 solutions to compare (e.g. A.cpp B.cpp into A B)
# 3. run: python3 test_generator.py A B
# Note that 'test_generator.py', 'A' and 'B' must be in the SAME FOLDER
# Note that A and B must READ FROM STANDARD INPUT, not from file,
# be careful with the usual freopen("input.in", "r", stdin) in them
import sys, subprocess
from datetime import datetime
from random import randint, seed

def buildTestCase(): # example of trivial "a+b" problem
  a = randint(1,100)
  b = randint(1,100)
  return f"{a} {b}\n"

seed(datetime.now().timestamp())
ntests = 100 # change as wanted
sol1 = sys.argv[1]
sol2 = sys.argv[2]
# Sometimes it's a good idea to use extra arguments that could then be
# passed to 'buildTestCase' and help you "shape" your tests
for curtest in range(ntests):
  test_case = buildTestCase()
  # Here the test is executed and outputs are compared
  print("running... ", end='')
  ans1 = subprocess.check_output(f"./{sol1}",
      input=test_case.encode('utf-8')).decode('utf-8')
  ans2 = subprocess.check_output(f"./{sol2}",
      input=test_case.encode('utf-8')).decode('utf-8')
  if ans1 == ans2:
    assert ans1 != "", 'ERROR?? ans1 = ans2 = empty string ("")'
    print("OK")
  else:
    print("FAILED!")
    print(test_case)
    print(f"ans from {sol1}:\n{ans1}")
    print(f"ans from {sol2}:\n{ans2}")
    break
```

10.5   Funciones Utiles

Algo	Params	Función
fill, fill_n	f, l / n, elem	*void* llena [f, l) o [f,f+n) con elem
lower_bound, upper_bound	f, l, elem	*it* al primer ultimo donde se puede insertar elem para que quede ordenada
copy	f, l, resul	hace resul+i=f+i $\forall i$
find, find_if, find_first_of	f, l, elem / pred / f2, l2	*it* encuentra i $\in$[f,l) tq. i=elem, pred(i), i$\in$[f2,l2)
count, count_if	f, l, elem/pred	cuenta elem, pred(i)
search	f, l, f2, l2	busca [f2,l2) $\in$ [f,l)
replace, replace_if	f, l, old / pred, new	cambia old / pred(i) por new
lexicographical_compare	f1,l1,f2,l2	*bool* con [f1,l1)¡[f2,l2)
accumulate	f,l,i,[op]	$T = \sum$/oper de [f,l)
inner_product	f1, l1, f2, i	$T = i + [f1, l1) \cdot [f2, \dots)$
partial_sum	f, l, r, [op]	r+i = $\sum$/oper de [f,f+i] $\forall i \in$[f,l)
__builtin_ffs	unsigned int	Pos. del primer 1 desde la derecha
__builtin_clz	unsigned int	Cant. de ceros desde la izquierda.
__builtin_ctz	unsigned int	Cant. de ceros desde la derecha.
__builtin_popcount	unsigned int	Cant. de 1's en x.
__builtin_parity	unsigned int	1 si x es par, 0 si es impar.
__builtin_XXXXXXll	unsigned ll	= pero para long long's.