Team notebook

Make PersueychUN Great Again - Universidad Nacional de Colombia, Bogota

March 19, 2019

Contents

1 Data structures

1.1 Centroid decomposition

```
int cnt[N], depth[N], f[N];
int dfs (int u, int p = -1) {
  cnt[u] = 1;
  for (int v : g[u])
    if (!depth[v] && v != p)
      cnt[u] += dfs(v, u);
    return cnt[u];
}
int get_centroid (int u, int r, int p = -1) {
  for (int v : g[u])
    if (!depth[v] && v != p && cnt[v] > r)
      return get_centroid(v, r, u);
  return u;
}
int decompose (int u, int d = 1) {
  int centroid = get_centroid(u, dfs(u) >> 1);
  depth[centroid] = d;
  /// magic function
  for (int v : g[centroid])
    if (!depth[v])
      f[decompose(v, d + 1)] = centroid;
  return centroid;
}
int lca (int u, int v) {
  for (; u != v; u = f[u])
    if (depth[v] > depth[u])
      swap(u, v);
  return u;
}
```

1.2 Fenwick tree

```
int lower_find(int val) { /// last value < or <= to val
  int idx = 0;
  for(int i = 31-__builtin_clz(n); i >= 0; --i) {
    int nidx = idx | (1 << i);
    if(nidx <= n && bit[nidx] <= val) { /// change <= to <
      val -= bit[nidx];
      idx = nidx;
    }
  }
  return idx;
}
```

1.3 Heavy light decomposition

```
int idx;
vector<int> len, hld_child, hld_index, hld_root, up;
void dfs( int u, int p = 0 ) {
  len[u] = 1;
  up[u] = p;
  for( auto& v : g[u] ) {
    if( v == p ) continue;
    depth[v] = depth[u]+1;
    dfs(v, u);
    len[u] += len[v];
    if( hld_child[u] == -1 || len[hld_child[u]] < len[v] )
      hld_child[u] = v;
  }
}
void build_hld( int u, int p = 0 ) {
  hld_index[u] = idx++;
  narr[hld_index[u]] = arr[u]; /// to initialize the segment tree
  if( hld_root[u] == -1 ) hld_root[u] = u;
  if( hld_child[u] != -1 ) {
    hld_root[hld_child[u]] = hld_root[u];
    build_hld(hld_child[u], u);
  }
  for( auto& v : g[u] ) {
    if( v == p || v == hld_child[u] ) continue;
    build_hld(v, u);
  }
}
void update_hld( int u, int val ) {
  update_tree(hld_index[u], val);
}
data query_hld( int u, int v ) {
  data val = NULL_DATA;
  while( hld_root[u] != hld_root[v] ) {
    if( depth[hld_root[u]] < depth[hld_root[v]] ) swap(u, v);
    val = val+query_tree(hld_index[hld_root[u]], hld_index[u]);
    u = up[hld_root[u]];
```

```
  }
  if( depth[u] > depth[v] ) swap(u, v);
  val = val+query_tree(hld_index[u], hld_index[v]);
  return val;
/// when updates are on edges use:
///   if (depth[u] == depth[v]) return val;
///   val = val+query_tree(hld_index[u] + 1, hld_index[v]);
}
void build(int n, int root) {
  len = hld_index = up = depth = vector<int>(n+1);
  hld_child = hld_root = vector<int>(n+1, -1);
  idx = 1; /// segtree index [1, n]
  dfs(root, root); build_hld(root, root);
  /// initialize data structure
}
```

1.4 Mo's

```
struct query { /// Complexity: O(|N+Q|*sqrt(|N|)*|ADD/DEL|)
  int l, r, idx;
  query (int l, int r, int idx) : l(l), r(r), idx(idx) {}
};
int S; // s = sqrt(n)
bool cmp (query a, query b) {
  if (a.l/S != b.l/S) return a.l/S < b.l/S;
  return a.r > b.r;
}
S = sqrt(n); // n = size of array
sort(q.begin(), q.end(), cmp);
int l = 0, r = -1;
for (int i = 0; i < q.size(); ++i) {
  while (r < q[i].r) add(++r);
  while (l > q[i].l) add(--l);
  while (r > q[i].r) del(r--);
  while (l < q[i].l) del(l++);
  ans[q[i].idx] = get_ans();
}
```

1.5 Order statistics

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;
```

1.6 Rmq

```
struct rmq {
  vector<vector<int> > table;
  rmq(vector<int> &v) : table(v.size() + 1, vector<int>(20)) {
    int n = v.size()+1;
    for (int i = 0; i < n; i++) table[i][0] = v[i];
    for (int j = 1; (1<<j) <= n; j++)
      for (int i = 0; i + (1<<j-1) < n; i++)
        table[i][j] = max(table[i][j-1], table[i + (1<<j-1)][j-1]);
  }
  int query(int a, int b) {
    int j = 31 - __builtin_clz(b-a+1);
    return max(table[a][j], table[b-(1<<j)+1][j]);
  }
};
```

1.7 Sack

```
int dfs(int u, int p = -1) {
  who[t] = u; fr[u] = t++;
  pii best = {0, -1};
  int sz = 1;
  for(auto v : g[u])
    if(v != p) {
      int cur_sz = dfs(v, u);
      sz += cur_sz;
      best = max(best, {cur_sz, v});
    }
  to[u] = t-1;
  big[u] = best.second;
  return sz;
}
void add(int u, int x) { /// x == 1 add, x == -1 delete
  cnt[u] += x;
}
void go(int u, int p = -1, bool keep = true) {
  for(auto v : g[u])
    if(v != p && v != big[u])
      go(v, u, 0);
  if(big[u] != -1) go(big[u], u, 1);
  for(auto v : g[u]) /// add all small
    if(v != p && v != big[u])
      for(int i = fr[v]; i <= to[v]; i++)
        add(who[i], 1);
  add(u, 1);
  ans[u] = get(u);
  if(!keep)
    for(int i = fr[u]; i <= to[u]; i++)
      add(who[i], -1);
}
void solve(int root) {
  t = 0;
  dfs(root);
```

```
  go(root);
}
```

1.8 Treap

```
mt19937_64 rng64(chrono::steady_clock::now().time_since_epoch().count());
uniform_int_distribution<ll> dis64(0, 1ll<<60);
template <typename T>
class treap {
private:
  struct item;
  typedef struct item * pitem;
  pitem root = NULL;
  struct item {
    ll prior; int cnt, rev;
    T key, add, fsum;
    pitem l, r;
    item(T x, ll p) {
      add = 0*x; key = fsum = x;
      cnt = 1; rev = 0;
      l = r = NULL; prior = p;
    }
  };
  int cnt(pitem it) { return it ? it->cnt : 0; }
  void upd_cnt(pitem it) {
    if(it) it->cnt = cnt(it->l) + cnt(it->r) + 1;
  }
  void upd_sum(pitem it) {
    if(it) {
      it->fsum = it->key;
      if(it->l) it->fsum += it->l->fsum;
      if(it->r) it->fsum += it->r->fsum;
    }
  }
  void update(pitem t, T add, int rev) {
    if(!t) return;
    t->add = t->add + add;
    t->rev = t->rev ^ rev;
    t->key = t->key + add;
    t->fsum = t->fsum + cnt(t) * add;
  }
  void push(pitem t) {
     if(!t || (t->add == 0*T() && t->rev == 0)) return;
    update(t->l, t->add, t->rev);
    update(t->r, t->add, t->rev);
    if(t->rev) swap(t->l,t->r);
    t->add = 0*T(); t->rev = 0;
  }
  void merge(pitem & t, pitem l, pitem r) {
    push(l); push(r);
    if(!l || !r) t = l ? l : r;
    else if(l->prior > r->prior) merge(l->r, l->r, r), t = l;
    else merge(r->l, l, r->l), t = r;
    upd_cnt(t); upd_sum(t);
  }
  void split(pitem t, pitem & l, pitem & r, int index) { // split index = how
      many elements
    if(!t) return void(l = r = 0);
    push(t);
    if(index <= cnt(t->l)) split(t->l, l, t->l, index), r = t;
    else split(t->r, t->r, r, index - 1 - cnt(t->l)), l = t;
    upd_cnt(t); upd_sum(t);
  }
  void insert(pitem & t, pitem it, int index) { // insert at position
    push(t);
    if(!t) t = it;
    else if(it->prior > t->prior) split(t, it->l, it->r, index), t = it;
    else if(index <= cnt(t->l)) insert(t->l, it, index);
    else insert(t->r, it, index-cnt(t->l)-1);
    upd_cnt(t); upd_sum(t);
  }
  void erase(pitem & t, int index) {
    push(t);
    if(cnt(t->l) == index) merge(t, t->l, t->r);
    else if(index < cnt(t->l)) erase(t->l, index);
    else erase(t->r, index - cnt(t->l) - 1);
    upd_cnt(t); upd_sum(t);
  }
  T get(pitem t, int index) {
    push(t);
    if(index < cnt(t->l)) return get(t->l, index);
    else if(index > cnt(t->l)) return get(t->r, index - cnt(t->l) - 1);
    return t->key;
  }
public:
  int size() { return cnt(root); }
  void insert(int pos, T x) {
    if(pos > size()) return;
    pitem it = new item(x, dis64(rng64));
    insert(root, it, pos);
  }
  void erase(int pos) {
    if(pos >= size()) return;
    erase(root, pos);
  }
  T operator[](int index) { return get(root, index); }
};
```

2 Dp optimization

2.1 Convex hull trick dynamic

```
typedef ll T;
const T is_query = -(1LL<<62); // special value for query
```

```
struct line {
  T m, b;
  mutable multiset<line>::iterator it, end;
  const line* succ(multiset<line>::iterator it) const {
    return (++it == end ? nullptr : &*it);
  }
  bool operator < (const line& rhs) const {
    if(rhs.b != is_query) return m < rhs.m;
    const line *s = succ(it);
    if(!s) return 0;
    return b-s->b < (s->m-m)*rhs.m;
  }
};
struct hull_dynamic : public multiset<line> { // for maximum
  bool bad(iterator y) {
    iterator z = next(y);
    if(y == begin()){
      if(z == end()) return false;
      return y->m == z->m && y->b <= z->b;
    }
    iterator x = prev(y);
    if(z == end()) return y->m == x->m && y->b <= x->b;
    return (x->b - y->b)*(z->m - y->m) >=
          (y->b - z->b)*(y->m - x->m);
  }
  iterator next(iterator y){ return ++y; }
  iterator prev(iterator y){ return --y; }
  void add(T m, T b){
    iterator y = insert((line){m, b});
    y->it = y; y->end = end();
    if(bad(y)){ erase(y); return; }
    while(next(y) != end() && bad(next(y))) erase(next(y));
    while(y != begin() && bad(prev(y))) erase(prev(y));
  }
  T eval(T x){
    line l = *lower_bound((line){x, is_query});
    return l.m*x+l.b;
  }
};
```

2.2 Convex hull trick

```
struct line {
  ll m, b;
  line (ll m, ll b) : m(m), b(b) {}
  ll eval (ll x) { return m*x + b; }
};
vector<line> lines[MAXN];
vector<lf> inter[MAXN];
lf get_inter (line &a, line &b) { // be careful with same slope !!!
  return lf(b.b - a.b) / lf(a.m - b.m);
}
//works for
//dp[i] = min(b[j] * a[i] + dp[j]) with j < i and b[i] > b[i + 1]
//dp[i] = max(b[j] * a[i] + dp[j]) with j < i and b[i] < b[i + 1]
void add (line l, int u) { // lines must be added in slope order
  while (lines[u].size() >= 2 && get_inter(lines[u][lines[u].size()-2], l) <=
      inter[u][lines[u].size()-2]) {
    lines[u].pop_back();
    inter[u].pop_back();
  }
  int len = lines[u].size();
  lines[u].push_back(l);
  if (lines[u].size()-1 > 0) inter[u].push_back(get_inter(lines[u][len],
      lines[u][len-1]));
}
ll get_min (lf x, int u) {
  if(lines[u].size() == 0) return INF;
  if (lines[u].size() == 1) return lines[u][0].eval(x);
  int pos = lower_bound(inter[u].begin(), inter[u].end(), x) - inter[u].begin();
  return lines[u][pos].eval(x);
}
```

2.3 Divide and conquer

```
void go(int k, int l, int r, int opl, int opr) {
  if(l > r) return;
  int mid = (l + r) / 2, op = -1;
  ll &best = dp[mid][k];
  best = INF;
  for(int i = min(opr, mid); i >= opl; i--) {
    ll cur = dp[i][k-1] + cost(i+1, mid);
    if(best > cur) {
      best = cur;
      op = i;
    }
  }
  go(k, l, mid-1, opl, op);
  go(k, mid+1, r, op, opr);
}
```

3 Formulas

3.1 2-SAT rules

- $p \to q \equiv \neg p \lor q$
- $p \to q \equiv \neg q \to \neg p$
- $p \lor q \equiv \neg p \to q$
- $p \land q \equiv \neg(p \to \neg q)$

- $\neg(p \to q) \equiv p \land \neg q$
- $(p \to q) \land (p \to r) \equiv p \to (q \land r)$
- $(p \to q) \lor (p \to r) \equiv p \to (q \lor r)$
- $(p \to r) \land (q \to r) \equiv (p \land q) \to r$
- $(p \to r) \lor (q \to r) \equiv (p \lor q) \to r$
- $(p \land q) \lor (r \land s) \equiv (p \lor r) \land (p \lor s) \land (q \lor r) \land (q \lor s)$

3.2 Burnside's lemma

$\#orbitas = \frac{1}{|G|}\sum_{g \in G} |fix(g)|$

1. **G**: Las acciones que se pueden aplicar sobre un elemento, incluyendo la identidad, eg. Shift 0 veces, Shift 1 veces...
2. **Fix(g)**: Es el número de elementos que al aplicar g vuelven a ser ser ellos mismos
3. **Órbita**: El conjunto de elementos que pueden ser iguales entre si al aplicar alguna de las acciones de G

3.3 Catalan Numbers

- $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$ con $n \geq 0$, $C_0 = 1$ y $C_{n+1} = \frac{2(2n+1)}{n+2}C_n$
- 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670

3.4 Combinatorics

- Distribute N objects among K people
  $\left(\!\binom{n}{k}\!\right) = \binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!}$
- Hockey-stick identity
  $\sum_{i=r}^{n}\binom{i}{r} = \binom{n+1}{r+1}$

3.5 Compound Interest

- N is the initial population, it grows at a rate of R. So, after X years the popularion will be $N \times (1+R)^X$

3.6 DP optimization theory

Name	Original Recurrence	Sufficient Condition		
CH 1	$dp[i] = min_{j<i}\{dp[j]+b[j]*a[i]\}$	$b[j] \geq b[j+1]$Optionally $a[i] \leq a[i+1]$	$O(n^2)$	$O(n)$
CH 2	$dp[i][j] = min_{k<j}\{dp[i-1][k] + b[k]*a[j]\}$	$b[k] \geq b[k+1]$ Optionally $a[j] \leq a[j+1]$	$O(kn^2)$	$O(kn)$
D&Q	$dp[i][j] = min_{k<j}\{dp[i-1][k] + C[k][j]\}$	$A[i][j] \leq A[i][j+1]$	$O(kn^2)$	$O(kn\log n)$
Knuth	$dp[i][j] = min_{i<k<j}\{dp[i][k] + dp[k][j]\} + C[i][j]$	$A[i,j-1] \leq A[i,j] \leq A[i+1,j]$	$O(n^3)$	$O(n^2)$

Notes:

- $A[i][j]$ - the smallest k that gives the optimal answer, for example in $dp[i][j] = dp[i-1][k] + C[k][j]$
- $C[i][j]$ - some given cost function
- We can generalize a bit in the following way $dp[i] = \min_{j<i}\{F[j] + b[j]*a[i]\}$, where $F[j]$ is computed from $dp[j]$ in constant time

3.7 Euler Totient properties

- $\phi(p) = p - 1$
- $\phi(p^e) = p^e(1 - \frac{1}{p})$
- $\phi(n*m) = \phi(n)*\phi(m)$ si $gcd(n,m) = 1$
- $\phi(n) = n(1-\frac{1}{p_1})(1-\frac{1}{p_2})...(1-\frac{1}{p_k})$ donde p_i es primo y divide a n

3.8 Fermat's theorem

Let m be a prime and x and m coprimes, then:

- $x^{m-1} \mod m = 1$
- $x^k \mod m = x^{k \mod (m-1)} \mod m$
- $x^{\phi(m)} \mod m = 1$

3.9 Great circle distance or geographical distance

Great circle distance or geographical distance

- d = great distance, ϕ = latitude, λ = longitude, Δ = difference (all the values in radians)
- σ = central angle, angle form for the two vector
- $d = r*\sigma$, $\sigma = 2*\arcsin(\sqrt{\sin^2(\frac{\Delta\phi}{2}) + \cos(\phi_1)\cos(\phi_2)\sin^2(\frac{\Delta\lambda}{2})})$

3.10 Heron's Formula

- $s = \frac{a+b+c}{2}$
- $Area = \sqrt{s(s-a)(s-b)(s-c)}$
- a, b, c there are the lenghts of the sides

3.11 Interesting theorems

- $a^d \equiv a^{d \mod \phi(n)} \mod n$
  if $a \in Z^{n_*}$ or $a \notin Z^{n_*}$ and $d \mod \phi(n) \neq 0$
- $a^d \equiv a^{\phi(n)} \mod n$
  if $a \notin Z^{n_*}$ and $d \mod \phi(n) = 0$
- thus, for all a, n and d (with $d \geq \log_2(n)$)
  $a^d \equiv a^{\phi(n)+d \mod \phi(n)} \mod n$

3.12 Law of sines and cosines

- a, b, c: lenghts, A, B, C: opposite angles, d: circumcircle
- $\frac{a}{sin(A)} = \frac{b}{sin(B)} = \frac{c}{sin(C)} = d$
- $c^2 = a^2 + b^2 - 2ab\cos(C)$

3.13 Number of divisors

- $\tau(n) = \prod_{i=1}^{k}(\alpha_i + 1)$

3.14 Product of divisors of a number

$\mu(n) = n^{\frac{\tau(n)}{2}}$

- if p is a prime, then: $\mu(p^k) = p^{\frac{k(k+2)}{2}}$
- if a and b are coprimes, then: $\mu(ab) = \mu(a)^{\tau(b)}\mu(b)^{\tau(a)}$

3.15 Pythagorean triples ($a^2 + b^2 = c^2$)

- Given an arbitrary pair of integers m and n with $m > n > 0$:
  $a = m^2 - n^2$, $b = 2mn$, $c = m^2 + n^2$
- The triple generated by Euclid's formula is primitive if and only if m and n are coprime and not both odd.
- To generate all Pythagorean triples uniquely:
  $a = k(m^2 - n^2)$, $b = k(2mn)$, $c = k(m^2 + n^2)$
- If m and n are two odd integer such that $m > n$, then:
  $a = mn$, $b = \frac{m^2-n^2}{2}$, $c = \frac{m^2+n^2}{2}$
- If $n = 1$ or 2 there are no solutions. Otherwise
  n is even: $((\frac{n^2}{4} - 1)^2 + n^2 = (\frac{n^2}{4} + 1)^2)$
  n is odd: $((\frac{n^2-1}{2})^2 + n^2 = (\frac{n^2+1}{2})^2)$

3.16 Simplex Rules

The simplex algorithm operated on linear programs in standard form:
Maximixe : $c^T \cdot x$
Subject to : $Ax \leq b, x_i \geq 0$

- $x = (x_1, ..., x_n)$ the variables of the problem
- $c = (c_1, ..., c_n)$ are the coefficients of the objective function
- A is a $p \times n$ matrix and $b = (b_1, ..., b_p)$ constants with $b_j \geq 0$

3.17 Sum of divisors of a number

- $\sigma(n) = \prod_{i=1}^{k}(1 + p_i + ... + p_i^{\alpha_i}) = \prod_{i=1}^{k} \frac{p_i^{\alpha_i+1}-1}{p_i-1}$

3.18 Summations

- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{i=1}^{n} i^3 = (\frac{n(n+1)}{2})^2$
- $\sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$
- $\sum_{i=1}^{n} i^5 = \frac{(n(n+1))^2(2n^2+2n-1)}{12}$
- $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$ para $x \neq 1$

3.19 Theorems

- There is always a prime between numbers n^2 and $(n+1)^2$, where n is any positive integer
- There is an infinite number of pairs of the from $\{p, p+2\}$ where both p and $p+2$ are primes.
- Every even integer greater than 2 can be expressed as the sum of two primes.
- Every integer greater than 2 can be written as the sum of three primes.

4 Geometry

4.1 3D

```
typedef double T;
struct p3 {
  T x, y, z;
  // Basic vector operations
  p3 operator + (p3 p) { return {x+p.x, y+p.y, z+p.z }; }
  p3 operator - (p3 p) { return {x - p.x, y - p.y, z - p.z}; }
  p3 operator * (T d) { return {x*d, y*d, z*d}; }
  p3 operator / (T d) { return {x / d, y / d, z / d}; } // only for floating point
  // Some comparators
  bool operator == (p3 p) { return tie(x, y, z) == tie(p.x, p.y, p.z); }
  bool operator != (p3 p) { return !operator == (p); }
};
p3 zero {0, 0, 0 };
T operator | (p3 v, p3 w) { /// dot
  return v.x*w.x + v.y*w.y + v.z*w.z;
}
p3 operator * (p3 v, p3 w) { /// cross
  return { v.y*w.z - v.z*w.y, v.z*w.x - v.x*w.z, v.x*w.y - v.y*w.x };
}
T sq(p3 v) { return v | v; }
double abs(p3 v) { return sqrt(sq(v)); }
p3 unit(p3 v) { return v / abs(v); }
double angle(p3 v, p3 w) {
  double cos_theta = (v | w) / abs(v) / abs(w);
  return acos(max(-1.0, min(1.0, cos_theta)));
}
T orient(p3 p, p3 q, p3 r, p3 s) { /// orient s, pqr form a triangle
  return (q - p) * (r - p) | (s - p);
}
T orient_by_normal(p3 p, p3 q, p3 r, p3 n) { /// same as 2D but in n-normal
    direction
  return (q - p) * (r - p) | n;
}
struct plane {
  p3 n; T d;
  /// From normal n and offset d
  plane(p3 n, T d): n(n), d(d) {}
  /// From normal n and point P
  plane(p3 n, p3 p): n(n), d(n | p) {}
  /// From three non-collinear points P,Q,R
  plane(p3 p, p3 q, p3 r): plane((q - p) * (r - p), p) {}
  /// - these work with T = int
  T side(p3 p) { return (n | p) - d; }
  double dist(p3 p) { return abs(side(p)) / abs(n); }
  plane translate(p3 t) {return {n, d + (n | t)}; }
  /// - these require T = double
  plane shift_up(double dist) { return {n, d + dist * abs(n)}; }
  p3 proj(p3 p) { return p - n * side(p) / sq(n); }
  p3 refl(p3 p) { return p - n * 2 * side(p) / sq(n); }
};
struct line3d {
  p3 d, o;
  /// From two points P, Q
  line3d(p3 p, p3 q): d(q - p), o(p) {}
  /// From two planes p1, p2 (requires T = double)
  line3d(plane p1, plane p2) {
    d = p1.n * p2.n;
    o = (p2.n * p1.d - p1.n * p2.d) * d / sq(d);
  }
  /// - these work with T = int
  double sq_dist(p3 p) { return sq(d * (p - o)) / sq(d); }
  double dist(p3 p) { return sqrt(sq_dist(p)); }
  bool cmp_proj(p3 p, p3 q) { return (d | p) < (d | q); }
  /// - these require T = double
  p3 proj(p3 p) { return o + d * (d | (p - o)) / sq(d); }
  p3 refl(p3 p) { return proj(p) * 2 - p; }
  p3 inter(plane p) { return o - d * p.side(o) / (p.n | d); }
};

double dist(line3d l1, line3d l2) {
  p3 n = l1.d * l2.d;
  if(n == zero) // parallel
    return l1.dist(l2.o);
  return abs((l2.o - l1.o) | n) / abs(n);
}
p3 closest_on_line1(line3d l1, line3d l2) { /// closest point on l1 to l2
  p3 n2 = l2.d * (l1.d * l2.d);
  return l1.o + l1.d * ((l2.o - l1.o) | n2) / (l1.d | n2);
}
double small_angle(p3 v, p3 w) { return acos(min(abs(v | w) / abs(v) / abs(w),
     1.0)); }
double angle(plane p1, plane p2) { return small_angle(p1.n, p2.n); }
bool is_parallel(plane p1, plane p2) { return p1.n * p2.n == zero; }
bool is_perpendicular(plane p1, plane p2) { return (p1.n | p2.n) == 0; }
double angle(line3d l1, line3d l2) { return small_angle(l1.d, l2.d); }
bool is_parallel(line3d l1, line3d l2) { return l1.d * l2.d == zero; }
bool is_perpendicular(line3d l1, line3d l2) { return (l1.d | l2.d) == 0; }
double angle(plane p, line3d l) { return _pI / 2 - small_angle(p.n, l.d); }
bool is_parallel(plane p, line3d l) { return (p.n | l.d) == 0; }
bool is_perpendicular(plane p, line3d l) { return p.n * l.d == zero; }
line3d perp_through(plane p, p3 o) { return line(o, o + p.n); }
plane perp_through(line3d l, p3 o) { return plane(l.d, o); }
```

4.2 General

```
const lf eps = 1e-9;
typedef double T;
struct pt {
  T x, y;
  pt operator + (pt p) { return {x+p.x, y+p.y}; }
  pt operator - (pt p) { return {x-p.x, y-p.y}; }
```

```
  pt operator * (pt p) { return {x*p.x-y*p.y, x*p.y+y*p.x}; }
  pt operator * (T d) { return {x*d, y*d}; }
  pt operator / (lf d) { return {x/d, y/d}; } /// only for floating point
  bool operator == (pt b) { return x == b.x && y == b.y; }
  bool operator != (pt b) { return !(*this == b); }
  bool operator < (const pt &o) const { return y < o.y || (y == o.y && x < o.x); }
  bool operator > (const pt &o) const { return y > o.y || (y == o.y && x > o.x); }
};
int cmp (lf a, lf b) { return (a + eps < b ? -1 :(b + eps < a ? 1 : 0)); }
/** Already in complex **/
T norm(pt a) { return a.x*a.x + a.y*a.y; }
lf abs(pt a) { return sqrt(norm(a)); }
lf arg(pt a) { return atan2(a.y, a.x); }
ostream& operator << (ostream& os, pt &p) {
  return os << "("<< p.x << "," << p.y << ")";
}
/***/
istream &operator >> (istream &in, pt &p) {
    T x, y; in >> x >> y;
    p = {x, y};
    return in;
}
T dot(pt a, pt b) { return a.x*b.x + a.y*b.y; }
T cross(pt a, pt b) { return a.x*b.y - a.y*b.x; }
T orient(pt a, pt b, pt c) { return cross(b-a,c-a); }
//pt rot(pt p, lf a) { return {p.x*cos(a) - p.y*sin(a), p.x*sin(a) + p.y*cos(a)};
     }
//pt rot(pt p, double a) { return p * polar(1.0, a); } /// for complex
//pt rotate_to_b(pt a, pt b, lf ang) { return rot(a-b, ang)+b; }
pt rot90ccw(pt p) { return {-p.y, p.x}; }
pt rot90cw(pt p) { return {p.y, -p.x}; }
pt translate(pt p, pt v) { return p+v; }
pt scale(pt p, double f, pt c) { return c + (p-c)*f; }
bool are_perp(pt v, pt w) { return dot(v,w) == 0; }
int sign(T x) { return (T(0) < x) - (x < T(0)); }
pt unit(pt a) { return a/abs(a); }

bool in_angle(pt a, pt b, pt c, pt x) {
  assert(orient(a,b,c) != 0);
  if (orient(a,b,c) < 0) swap(b,c);
  return orient(a,b,x) >= 0 && orient(a,c,x) <= 0;
}

//lf angle(pt a, pt b) { return acos(max(-1.0, min(1.0,
     dot(a,b)/abs(a)/abs(b)))); }
//lf angle(pt a, pt b) { return atan2(cross(a, b), dot(a, b)); }
/// returns vector to transform points
pt get_linear_transformation(pt p, pt q, pt r, pt fp, pt fq) {
  pt pq = q-p, num{cross(pq, fq-fp), dot(pq, fq-fp)};
  return fp + pt{cross(r-p, num), dot(r-p, num)} / norm(pq);
}

bool half(pt p) { /// true if is in (0, 180]
  assert(p.x != 0 || p.y != 0); /// the argument of (0,0) is undefined
  return p.y > 0 || (p.y == 0 && p.x < 0);
}
bool half_from(pt p, pt v = {1, 0}) {
  return cross(v,p) < 0 || (cross(v,p) == 0 && dot(v,p) < 0);
}
bool polar_cmp(const pt &a, const pt &b) {
  return make_tuple(half(a), 0) < make_tuple(half(b), cross(a,b));
}

struct line {
  pt v; T c;
  line(pt v, T c) : v(v), c(c) {}
  line(T a, T b, T c) : v({b,-a}), c(c) {}
  line(pt p, pt q) : v(q-p), c(cross(v,p)) {}
  T side(pt p) { return cross(v,p)-c; }
  lf dist(pt p) { return abs(side(p)) / abs(v); }
  lf sq_dist(pt p) { return side(p)*side(p) / (lf)norm(v); }
  line perp_through(pt p) { return {p, p + rot90ccw(v)}; }
  bool cmp_proj(pt p, pt q) { return dot(v,p) < dot(v,q); }
  line translate(pt t) { return {v, c + cross(v,t)}; }
  line shift_left(double d) { return {v, c + d*abs(v)}; }
  pt proj(pt p) { return p - rot90ccw(v)*side(p)/norm(v); }
  pt refl(pt p) { return p - rot90ccw(v)*2*side(p)/norm(v); }
};

bool inter_ll(line l1, line l2, pt &out) {
  T d = cross(l1.v, l2.v);
  if (d == 0) return false;
  out = (l2.v*l1.c - l1.v*l2.c) / d;
  return true;
}
line bisector(line l1, line l2, bool interior) {
  assert(cross(l1.v, l2.v) != 0); /// l1 and l2 cannot be parallel!
  lf sign = interior ? 1 : -1;
  return {l2.v/abs(l2.v) + l1.v/abs(l1.v) * sign,
          l2.c/abs(l2.v) + l1.c/abs(l1.v) * sign};
}

bool in_disk(pt a, pt b, pt p) {
  return dot(a-p, b-p) <= 0;
}
bool on_segment(pt a, pt b, pt p) {
  return orient(a,b,p) == 0 && in_disk(a,b,p);
}
bool proper_inter(pt a, pt b, pt c, pt d, pt &out) {
  T oa = orient(c,d,a),
  ob = orient(c,d,b),
  oc = orient(a,b,c),
  od = orient(a,b,d);
  /// Proper intersection exists iff opposite signs
  if (oa*ob < 0 && oc*od < 0) {
    out = (a*ob - b*oa) / (ob-oa);
    return true;
  }
  return false;
}
```

```cpp
set<pt> inter_ss(pt a, pt b, pt c, pt d) {
  pt out;
  if (proper_inter(a,b,c,d,out)) return {out};
  set<pt> s;
  if (on_segment(c,d,a)) s.insert(a);
  if (on_segment(c,d,b)) s.insert(b);
  if (on_segment(a,b,c)) s.insert(c);
  if (on_segment(a,b,d)) s.insert(d);
  return s;
}
lf pt_to_seg(pt a, pt b, pt p) {
  if(a != b) {
    line l(a,b);
    if (l.cmp_proj(a,p) && l.cmp_proj(p,b)) /// if closest to projection
      return l.dist(p); /// output distance to line
  }
  return min(abs(p-a), abs(p-b)); /// otherwise distance to A or B
}
lf seg_to_seg(pt a, pt b, pt c, pt d) {
  pt dummy;
  if (proper_inter(a,b,c,d,dummy)) return 0;
  return min({pt_to_seg(a,b,c), pt_to_seg(a,b,d),
            pt_to_seg(c,d,a), pt_to_seg(c,d,b)});
}

enum {IN, OUT, ON};
struct polygon {
  vector<pt> p;
  polygon(int n) : p(n) {}
  int top = -1, bottom = -1;
  void delete_repetead() {
    vector<pt> aux;
    sort(p.begin(), p.end());
    for(pt &i : p)
      if(aux.empty() || aux.back() != i)
        aux.push_back(i);
    p.swap(aux);
  }
  bool is_convex() {
    bool pos = 0, neg = 0;
    for (int i = 0, n = p.size(); i < n; i++) {
      int o = orient(p[i], p[(i+1)%n], p[(i+2)%n]);
      if (o > 0) pos = 1;
      if (o < 0) neg = 1;
    }
    return !(pos && neg);
  }
  lf area(bool s = false) {
    lf ans = 0;
    for (int i = 0, n = p.size(); i < n; i++)
      ans += cross(p[i], p[(i+1)%n]);
    ans /= 2;
    return s ? ans : abs(ans);
  }
  lf perimeter() {
    lf per = 0;
    for(int i = 0, n = p.size(); i < n; i++)
      per += abs(p[i] - p[(i+1)%n]);
    return per;
  }
  bool above(pt a, pt p) { return p.y >= a.y; }
  bool crosses_ray(pt a, pt p, pt q) {
    return (above(a,q)-above(a,p))*orient(a,p,q) > 0;
  }
  int in_polygon(pt a) {
    int crosses = 0;
    for(int i = 0, n = p.size(); i < n; i++) {
      if(on_segment(p[i], p[(i+1)%n], a)) return ON;
      crosses += crosses_ray(a, p[i], p[(i+1)%n]);
    }
    return (crosses&1 ? IN : OUT);
  }
  void normalize() { /// polygon is CCW
    bottom = min_element(p.begin(), p.end()) - p.begin();
    vector<pt> tmp(p.begin()+bottom, p.end());
    tmp.insert(tmp.end(), p.begin(), p.begin()+bottom);
    p.swap(tmp);
    bottom = 0;
    top = max_element(p.begin(), p.end()) - p.begin();
  }
  int in_convex(pt a) {
    assert(bottom == 0 && top != -1);
    if(a < p[0] || a > p[top]) return OUT;
    T orientation = orient(p[0], p[top], a);
    if(orientation == 0) {
      if(a == p[0] || a == p[top]) return ON;
      return top == 1 || top + 1 == p.size() ? ON : IN;
    } else if (orientation < 0) {
      auto it = lower_bound(p.begin()+1, p.begin()+top, a);
      T d = orient(*prev(it), a, *it);
      return d < 0 ? IN : (d > 0 ? OUT: ON);
    }
    else {
      auto it = upper_bound(p.rbegin(), p.rend()-top-1, a);
      T d = orient(*it, a, it == p.rbegin() ? p[0] : *prev(it));
      return d < 0 ? IN : (d > 0 ? OUT: ON);
    }
  }
  polygon cut(pt a, pt b) {
    line l(a, b);
    polygon new_polygon(0);
    for(int i = 0, n = p.size(); i < n; ++i) {
      pt c = p[i], d = p[(i+1)%n];
      lf abc = cross(b-a, c-a), abd = cross(b-a, d-a);
      if(abc >= 0) new_polygon.p.push_back(c);
      if(abc*abd < 0) {
        pt out; inter_ll(l, line(c, d), out);
        new_polygon.p.push_back(out);
      }
    }
```

```
    return new_polygon;
  }
  void convex_hull() {
    sort(p.begin(), p.end());
    vector<pt> ch;
    ch.reserve(p.size()+1);
    for(int it = 0; it < 2; it++) {
      int start = ch.size();
      for(auto &a : p) {
        /// if colineal are needed, use < and remove repeated points
        while(ch.size() >= start+2 && orient(ch[ch.size()-2], ch.back(), a) <= 0)
          ch.pop_back();
        ch.push_back(a);
      }
      ch.pop_back();
      reverse(p.begin(), p.end());
    }
    if(ch.size() == 2 && ch[0] == ch[1]) ch.pop_back();
    /// be careful with CH of size < 3
    p.swap(ch);
  }
  vector<pii> antipodal() {
    vector<pii> ans;
    int n = p.size();
    if(n == 2) ans.push_back({0, 1});
    if(n < 3) return ans;
    auto nxt = [&](int x) { return (x+1 == n ? 0 : x+1); };
    auto area2 = [&](pt a, pt b, pt c) { return cross(b-a, c-a); };
    int b0 = 0;
    while(abs(area2(p[n - 1], p[0], p[nxt(b0)])) >
         abs(area2(p[n - 1], p[0], p[b0])))
      ++b0;
    for(int b = b0, a = 0; b != 0 && a <= b0; ++a) {
      ans.push_back({a, b});
      while (abs(area2(p[a], p[nxt(a)], p[nxt(b)])) >
            abs(area2(p[a], p[nxt(a)], p[b]))) {
        b = nxt(b);
        if(a != b0 || b != 0) ans.push_back({ a, b });
        else return ans;
      }
      if(abs(area2(p[a], p[nxt(a)], p[nxt(b)])) ==
         abs(area2(p[a], p[nxt(a)], p[b]))) {
        if(a != b0 || b != n-1) ans.push_back({ a, nxt(b) });
        else ans.push_back({ nxt(a), b });
      }
    }
    return ans;
  }
  pt centroid() {
    pt c{0, 0};
    lf scale = 6. * area(true);
    for(int i = 0, n = p.size(); i < n; ++i) {
      int j = (i+1 == n ? 0 : i+1);
      c = c + (p[i] + p[j]) * cross(p[i], p[j]);
    }
    return c / scale;
  }
  ll pick() {
    ll boundary = 0;
    for(int i = 0, n = p.size(); i < n; i++) {
      int j = (i+1 == n ? 0 : i+1);
      boundary += __gcd((ll)abs(p[i].x - p[j].x), (ll)abs(p[i].y - p[j].y));
    }
    return area() + 1 - boundary/2;
  }
  pt& operator[] (int i){ return p[i]; }
};

struct circle {
  pt c; T r;
};

circle center(pt a, pt b, pt c) {
  b = b-a, c = c-a;
  assert(cross(b,c) != 0); /// no circumcircle if A,B,C aligned
  pt cen = a + rot90ccw(b*norm(c) - c*norm(b))/cross(b,c)/2;
  return {cen, abs(a-cen)};
}
int inter_cl(circle c, line l, pair<pt, pt> &out) {
  lf h2 = c.r*c.r - l.sq_dist(c.c);
  if(h2 >= 0) {
    pt p = l.proj(c.c);
    pt h = l.v*sqrt(h2)/abs(l.v);
    out = {p-h, p+h};
  }
  return 1 + sign(h2);
}
int inter_cc(circle c1, circle c2, pair<pt, pt> &out) {
  pt d=c2.c-c1.c; double d2=norm(d);
  if(d2 == 0) { assert(c1.r != c2.r); return 0; } // concentric circles
  double pd = (d2 + c1.r*c1.r - c2.r*c2.r)/2; // = |O_1P| * d
  double h2 = c1.r*c1.r - pd*pd/d2; // = h2
  if(h2 >= 0) {
    pt p = c1.c + d*pd/d2, h = rot90ccw(d)*sqrt(h2/d2);
    out = {p-h, p+h};
  }
  return 1 + sign(h2);
}

int tangents(circle c1, circle c2, bool inner, vector<pair<pt,pt>> &out) {
  if(inner) c2.r = -c2.r;
  pt d = c2.c-c1.c;
  double dr = c1.r-c2.r, d2 = norm(d), h2 = d2-dr*dr;
  if(d2 == 0 || h2 < 0) { assert(h2 != 0); return 0; }
  for(double s : {-1,1}) {
    pt v = (d*dr + rot90ccw(d)*sqrt(h2)*s)/d2;
    out.push_back({c1.c + v*c1.r, c2.c + v*c2.r});
  }
  return 1 + (h2 > 0);
}
```

```
int tangent_through_pt(pt p, circle c, pair<pt, pt> &out) {
  double d = abs(p - c.c);
        if(d < c.r) return 0;
  pt base = c.c-p;
  double w = sqrt(norm(base) - c.r*c.r);
  pt a = {w, c.r}, b = {w, -c.r};
  pt s = p + base*a/norm(base)*w;
  pt t = p + base*b/norm(base)*w;
  out = {s, t};
  return 1 + (abs(c.c-p) == c.r);
}
```

5 Graphs

5.1 2-satisfiability

```
struct sat2 {
  int n;
  vector<vector<vector<int>>> g;
  vector<int> tag;
  vector<bool> seen, value;
  stack<int> st;
  sat2(int n) : n(n), g(2, vector<vector<int>>(2*n)), tag(2*n), seen(2*n),
       value(2*n) { }
  int neg(int x) { return 2*n-x-1; }
  void add_or(int u, int v) { implication(neg(u), v); }
  void make_true(int u) { add_edge(neg(u), u); }
  void make_false(int u) { make_true(neg(u)); }
  void eq(int u, int v) {
    implication(u, v);
    implication(v, u);
  }
  void diff(int u, int v) { eq(u, neg(v)); }
  void implication(int u, int v) {
    add_edge(u, v);
    add_edge(neg(v), neg(u));
  }
  void add_edge(int u, int v) {
    g[0][u].push_back(v);
    g[1][v].push_back(u);
  }
  void dfs(int id, int u, int t = 0) {
    seen[u] = true;
    for(auto& v : g[id][u])
      if(!seen[v])
        dfs(id, v, t);
    if(id == 0) st.push(u);
    else tag[u] = t;
  }
  void kosaraju() {
    for(int u = 0; u < n; u++) {
      if(!seen[u]) dfs(0, u);
      if(!seen[neg(u)]) dfs(0, neg(u));
    }
    fill(seen.begin(), seen.end(), false);
    int t = 0;
    while(!st.empty()) {
      int u = st.top(); st.pop();
      if(!seen[u]) dfs(1, u, t++);
    }
  }
  bool satisfiable() {
    kosaraju();
    for(int i = 0; i < n; i++) {
      if(tag[i] == tag[neg(i)]) return false;
      value[i] = tag[i] > tag[neg(i)];
    }
    return true;
  }
};
```

5.2 Eulerian path

```
bool eulerian(vector<int> &tour) { /// directed graph
  int one_in = 0, one_out = 0, start = -1;
  bool ok = true;
  for (int i = 0; i < n; i++) {
    if(out[i] && start == -1) start = i;
    if(out[i] - in[i] == 1) one_out++, start = i;
    else if(in[i] - out[i] == 1) one_in++;
    else ok &= in[i] == out[i];
  }
  ok &= one_in == one_out && one_in <= 1;
  if (ok) {
    function<void(int)> go = [&](int u) {
      while(g[u].size()) {
        int v = g[u].back();
        g[u].pop_back();
        go(v);
      }
      tour.push_back(u);
    };
    go(start);
    reverse(tour.begin(), tour.end());
    if(tour.size() == edges + 1) return true;
  }
  return false;
}
```

5.3 Lowest common ancestor

```
int lca(int a, int b) {
  if(depth[a] < depth[b]) swap(a, b);
  //int ans = INT_MAX;
  for(int i = LOG2-1; i >= 0; --i)
    if(depth[ dp[a][i] ] >= depth[b]) {
      //ans = min(ans, mn[a][i]);
      a = dp[a][i];
    }
  //if (a == b) return ans;
  if(a == b) return a;
  for(int i = LOG2-1; i >= 0; --i)
    if(dp[a][i] != dp[b][i]) {
      //ans = min(ans, mn[a][i]);
      //ans = min(ans, mn[b][i]);
      a = dp[a][i],
      b = dp[b][i];
    }
  //ans = min(ans, mn[a][0]);
  //ans = min(ans, mn[b][0]);
  //return ans;
  return dp[a][0];
}
void dfs(int u, int d = 1, int p = -1) {
  depth[u] = d;
  for(auto v : g[u]) {
    //int v = x.first;
    //int w = x.second;
    if(v != p) {
      dfs(v, d + 1, u);
      dp[v][0] = u;
      //mn[v][0] = w;
    }
  }
}
void build(int n) {
  for(int i = 0; i <= n; i++) dp[i][0] = -1;
  for(int i = 0; i < n, i++) {
    if(dp[i][0] == -1) {
      dp[i][0] = i;
      //mn[i][0] = INT_MAX;
      dfs(i);
    }
  }
  for(int j = 0; j < LOG2-1; ++j)
    for(int i = 0; i <= n; ++i) { // nodes
      dp[i][j+1] = dp[ dp[i][j] ][j];
      //mn[i][j+1] = min(mn[ dp[i][j] ][j], mn[i][j]);
    }
}
```

5.4 Scc

```
int scc(int n) {
  vector<int> dfn(n+1), low(n+1), in_stack(n+1);
  stack<int> st;
  int tag = 0;
  function<void(int, int&)> dfs = [&](int u, int &t) {
    dfn[u] = low[u] = ++t;
    st.push(u);
    in_stack[u] = true;
    for(auto &v : g[u]) {
      if(!dfn[v]) {
        dfs(v, t);
        low[u] = min(low[u], low[v]);
      } else if(in_stack[v])
        low[u] = min(low[u], dfn[v]);
    }
    if (low[u] == dfn[u]) {
      int v;
      do {
        v = st.top(); st.pop();
//        id[v] = tag;
        in_stack[v] = false;
      } while (v != u);
      tag++;
    }
  };
  for(int u = 1, t; u <= n; ++u) {
    if(!dfn[u]) dfs(u, t = 0);
  }
  return tag;
}
```

5.5 Tarjan tree

```
struct tarjan_tree {
  int n;
  vector<vector<int>> g, comps;
  vector<pii> bridge;
  vector<int> id, art;
  tarjan_tree(int n) : n(n), g(n+1), id(n+1), art(n+1) {}
  void add_edge(vector<vector<int>> &g, int u, int v) { /// nodes from [1, n]
    g[u].push_back(v);
    g[v].push_back(u);
  }
  void add_edge(int u, int v) { add_edge(g, u, v); }
  void tarjan(bool with_bridge) {
    vector<int> dfn(n+1), low(n+1);
    stack<int> st;
    comps.clear();
    function<void(int, int, int&)> dfs = [&](int u, int p, int &t) {
      dfn[u] = low[u] = ++t;
      st.push(u);
      int cntp = 0;
```

```
      for(int v : g[u]) {
        cntp += v == p;
        if(!dfn[v])   {
          dfs(v, u, t);
          low[u] = min(low[u], low[v]);
          if(with_bridge && low[v] > dfn[u]) {
            bridge.push_back({min(u,v), max(u,v)});
            comps.push_back({});
            for(int w = -1; w != v; )
              comps.back().push_back(w = st.top()), st.pop();
          }
          if(!with_bridge && low[v] >= dfn[u]) {
            art[u] = (dfn[u] > 1 || dfn[v] > 2);
            comps.push_back({u});
            for(int w = -1; w != v; )
              comps.back().push_back(w = st.top()), st.pop();
          }
        }
        else if(v != p || cntp > 1) low[u] = min(low[u], dfn[v]);
      }
      if(p == -1 && ( with_bridge || g[u].size() == 0 )) {
        comps.push_back({});
        for(int w = -1; w != u; )
          comps.back().push_back(w = st.top()), st.pop();
      }
    };
    for(int u = 1, t; u <= n; ++u)
      if(!dfn[u]) dfs(u, -1, t = 0);
  }
  vector<vector<int>> build_block_cut_tree() {
    tarjan(false);
    int t = 0;
    for(int u = 1; u <= n; ++u)
      if(art[u]) id[u] = t++;
    vector<vector<int>> tree(t+comps.size());
    for(int i = 0; i < comps.size(); ++i)
      for(int u : comps[i]) {
        if(!art[u]) id[u] = i+t;
        else add_edge(tree, i+t, id[u]);
      }
    return tree;
  }
  vector<vector<int>> build_bridge_tree() {
    tarjan(true);
    vector<vector<int>> tree(comps.size());
    for(int i = 0; i < comps.size(); ++i)
      for(int u : comps[i]) id[u] = i;
    for(auto &b : bridge)
      add_edge(tree, id[b.first], id[b.second]);
    return tree;
  }
};
```

6 Math

6.1 Chinese remainder theorem

```
/// finds a suitable x that meets: x is congruent to a_i mod n_i
/** Works for non-coprime moduli.
 Returns {-1,-1} if solution does not exist or input is invalid.
 Otherwise, returns {x,L}, where x is the solution unique to mod L = LCM of mods
*/
pair<int, int> chinese_remainder_theorem( vector<int> A, vector<int> M ) {
  int n = A.size(), a1 = A[0], m1 = M[0];
  for(int i = 1; i < n; i++) {
    int a2 = A[i], m2 = M[i];
    int g = __gcd(m1, m2);
    if( a1 % g != a2 % g ) return {-1,-1};
    int p, q;
    eea(m1/g, m2/g, &p, &q);
    int mod = m1 / g * m2;
    q %= mod; p %= mod;
    int x = ((1ll*(a1%mod)*(m2/g))%mod*q + (1ll*(a2%mod)*(m1/g))%mod*p) % mod; //
         if WA there is overflow
    a1 = x;
    if (a1 < 0) a1 += mod;
    m1 = mod;
  }
  return {a1, m1};
}
```

6.2 Constant modular inverse

```
inv[1] = 1;
for(int i = 2; i < p; ++i)
        inv[i] = (p - (p / i) * inv[p % i] % p) % p;
```

6.3 Extended euclides

```
ll eea(ll a, ll b, ll& x, ll& y) {
  ll xx = y = 0; ll yy = x = 1;
  while (b) {
    ll q = a / b; ll t = b; b = a % b; a = t;
    t = xx; xx = x - q * xx; x = t;
    t = yy; yy = y - q * yy; y = t;
  }
  return a;
}
ll inverse(ll a, ll n) {
  ll x, y;
  ll g = eea(a, n, x, y);
  if(g > 1)
```

```
    return -1;
  return (x % n + n) % n;
}
```

6.4 Fast Fourier transform module

```
const int mod = 7340033; /// mod = c*2^k+1
/// find g = primitive root of mod.
const int root = 2187; /// (g^c)%mod
const int root_1 = 4665133; /// inverse of root
const int root_pw = 1 << 19; /// 2^k

pii find_c_k(int mod) {
  pii ans;
  for(int k = 1; (1<<k) < mod; k++) {
    int pot = 1<<k;
    if((mod - 1) % pot == 0)
      ans = {(mod-1) / pot, k};
  }
  return ans;
}

int find_primitive_root(int mod) {
  vector<bool> seen(mod);
  for(int r = 2; ; r++) {
    fill(seen.begin(), seen.end(), 0);
    int cur = 1, can = 1;
    for(int i = 0; i <= mod-2 && can; i++) {
      if(seen[cur]) can = 0;
      seen[cur] = 1;
      cur = (1ll*cur*r) % mod;
    }
    if(can)
      return r;
  }
  assert(false);
}

void fft(vector<int> &a, bool inv = 0) {
  int n = a.size();
  for(int i = 1, j = 0; i < n; i++) {
    int c = n >> 1;
    for (; j >= c; c >>= 1) j -= c;
    j += c;
    if(i < j) swap(a[i], a[j]);
  }
  for (int len = 2; len <= n; len <<= 1) {
    int wlen = inv ? root_1 : root;
    for(int i = len; i < root_pw; i <<= 1) wlen = (1 LL * wlen * wlen) % mod;
    for(int i = 0; i < n; i += len) {
      int w = 1;
      for(int j = 0; j < (len >> 1); j++) {
        int u = a[i + j], v = (a[i + j + (len >> 1)] * 1 LL * w) % mod;
        a[i + j] = u + v < mod ? u + v : u + v - mod;
        a[i + j + (len >> 1)] = u - v >= 0 ? u - v : u - v + mod;
        w = (w * 1 LL * wlen) % mod;
      }
    }
  }
  if (inv) {
    int nrev = pow(n);
    for(int i = 0; i < n; i++) a[i] = (a[i] * 1 LL * nrev) % mod;
  }
}
vector<int> mul(const vector <int> a, const vector <int> b) {
  vector<int> fa(a.begin(), a.end()), fb(b.begin(), b.end());
  int n = 1;
  while (n < max(a.size(), b.size())) n <<= 1;
  n <<= 1;
  fa.resize(n); fb.resize(n);
  fft(fa); fft(fb);
  for (int i = 0; i < n; i++) fa[i] = (1ll * fa[i] * fb[i]) % mod;
  fft(fa, 1);
  return fa;
}
```

6.5 Fast fourier transform

```
namespace fft {
  typedef long long ll;
  const double PI = acos(-1.0);
  vector<int> rev;
  struct pt {
    double r, i;
    pt(double r = 0.0, double i = 0.0) : r(r), i(i) {}
    pt operator + (const pt & b) { return pt(r + b.r, i + b.i); }
    pt operator - (const pt & b) { return pt(r - b.r, i - b.i); }
    pt operator * (const pt & b) { return pt(r * b.r - i * b.i, r * b.i + i *
        b.r); }
  };
  void fft(vector<pt> &y, int on) {
    int n = y.size();
    for(int i = 1; i < n; i++) if(i < rev[i]) swap(y[i], y[rev[i]]);
    for(int m = 2; m <= n; m <<= 1) {
      pt wm(cos(-on * 2 * PI / m), sin(-on * 2 * PI / m));
      for(int k = 0; k < n; k += m) {
        pt w(1, 0);
        for(int j = 0; j < m / 2; j++) {
          pt u = y[k + j];
          pt t = w * y[k + j + m / 2];
          y[k + j] = u + t;
          y[k + j + m / 2] = u - t;
          w = w * wm;
        }
```

```
      }
    }
    if(on == -1)
      for(int i = 0; i < n; i++) y[i].r /= n;
  }
  vector<ll> mul(vector<ll> &a, vector<ll> &b) {
    int n = 1, la = a.size(), lb = b.size(), t;
    for(n = 1, t = 0; n <= (la+lb+1); n <<= 1, t++); t = 1<<(t-1);
    vector<pt> x1(n), x2(n);
    rev.assign(n, 0);
    for(int i = 0; i < n; i++) rev[i] = rev[i >> 1] >> 1 |(i & 1 ? t : 0);
    for(int i = 0; i < la; i++) x1[i] = pt(a[i], 0);
    for(int i = 0; i < lb; i++) x2[i] = pt(b[i], 0);
    fft(x1, 1); fft(x2, 1);
    for(int i = 0; i < n; i++) x1[i] = x1[i] * x2[i];
    fft(x1, -1);
    vector<ll> sol(n);
    for(int i = 0; i < n; i++) sol[i] = x1[i].r + 0.5;
    return sol;
  }
}
```

6.6 Gauss jordan

```
void gauss_jordan(vector<vector<double>> &a, vector<double> &x) {
  for(int i = 0; i < n; ++i) {
    int maxs = i;
    for(int j = i+1; j < n; ++j)
      if(abs(a[j][i]) > abs(a[maxs][i]))
        maxs = j;
    if(maxs != i)
      for(int j = 0; j <= n; ++j)
        swap(a[i][j], a[maxs][j]);
    for(int j = i + 1; j < n; ++j) {
      lf r = a[j][i]/a[i][i];
      for(int k = 0; k <= n; ++k)
        a[j][k] -= r*a[i][k];
    }
  }
  for(int i = n-1; i >= 0; --i) {
    x[i] = a[i][n]/a[i][i];
    for(int j = i-1; j >= 0; --j)
      a[j][n] -= a[j][i]*x[i];
  }
}
```

6.7 Integral

- Simpsons rule: $\int_a^b f(x)dx \approx \frac{b-a}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)]$
- Arc length: $s = \int_a^b \sqrt{1 + [f'(x)]^2}dx$
- Area of a surface of revolution: $A = 2\pi \int_a^b f(x)\sqrt{1 + [f'(x)]^2}dx$
- Volume of a solid of revolution: $V = \pi \int_a^b f(x)^2 dx$
- Note: In case of multiple functions such as *g(x) h(x)* for a solid of revolution then $f(x) = g(x) - h(x)$
- $f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$
- $f'(x) \approx \frac{f(x-2h)-8f(x-h)+8f(x+h)-f(x+2h)}{12h}$
- $f''(x) \approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2}$

6.8 Linear diaphontine

```
bool diophantine(ll a, ll b, ll c, ll &x, ll &y, ll &g) {
  x = y = 0;
  if(a == 0 && b == 0) return c == 0;
  if(b == 0) swap(a, b), swap(x, y);
  g = eea(abs(a), abs(b), x, y);
  if(c % g) return false;
  a /= g; b /= g; c /= g;
  if(a < 0) x *= -1;
  x = (x % b) * (c % b) % b;
  if(x < 0) x += b;
  y = (c - a*x) / b;
  return true;
}
///finds the first k | x + b * k / gcd(a, b) >= val
ll greater_or_equal_than(ll a, ll b, ll x, ll val, ll g) {
  return ceil(1.0 * (val - x) * g / b);
}
ll less_or_equal_than(ll a, ll b, ll x, ll val, ll g) {
  return floor(1.0 * (val - x) * g / b);
}
void get_xy (ll a, ll b, ll &x, ll &y, ll k, ll g) {
  x = x + b / g * k;
  y = y - a / g * k;
}
```

6.9 Matrix multiplication

```
const int MOD = 1e9+7;
struct matrix {
  const int N = 2;
  int m[N][N], r, c;
  matrix(int r = N, int c = N, bool iden = false) : r(r), c(c) {
    memset(m, 0, sizeof m);
    if(iden)
      for(int i = 0; i < r; i++) m[i][i] = 1;
```

```
  }
  matrix operator * (const matrix &o) const {
    matrix ret(r, o.c);
    for(int i = 0; i < r; ++i)
      for(int j = 0; j < o.c; ++j) {
        ll &r = ret.m[i][j];
        for(int k = 0; k < c; ++k)
          r = (r + 1ll*m[i][k]*o.m[k][j]) % MOD;
      }
    return ret;
  }
};
```

6.10 Miller rabin

```
ll mul (ll a, ll b, ll mod) {
  ll ret = 0;
  for(a %= mod, b %= mod; b != 0;
    b >>= 1, a <<= 1, a = a >= mod ? a - mod : a) {
    if (b & 1) {
      ret += a;
      if (ret >= mod) ret -= mod;
    }
  }
  return ret;
}
ll fpow (ll a, ll b, ll mod) {
  ll ans = 1;
  for (; b; b >>= 1, a = mul(a, a, mod))
    if (b & 1)
      ans = mul(ans, a, mod);
  return ans;
}
bool witness (ll a, ll s, ll d, ll n) {
  ll x = fpow(a, d, n);
  if (x == 1 || x == n - 1) return false;
  for (int i = 0; i < s - 1; i++) {
    x = mul(x, x, n);
    if (x == 1) return true;
    if (x == n - 1) return false;
  }
  return true;
}
ll test[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 0};
bool is_prime (ll n) {
  if (n < 2) return false;
  if (n == 2) return true;
  if (n % 2 == 0) return false;
  ll d = n - 1, s = 0;
  while (d % 2 == 0) ++s, d /= 2;
  for (int i = 0; test[i] && test[i] < n; ++i)
    if (witness(test[i], s, d, n))
      return false;
  return true;
}
```

6.11 Pollard's rho

```
ll pollard_rho(ll n, ll c) {
  ll x = 2, y = 2, i = 1, k = 2, d;
  while (true) {
    x = (mul(x, x, n) + c);
    if (x >= n) x -= n;
    d = __gcd(x - y, n);
    if (d > 1) return d;
    if (++i == k) y = x, k <<= 1;
  }
  return n;
}
void factorize(ll n, vector<ll> &f) {
  if (n == 1) return;
  if (is_prime(n)) {
    f.push_back(n);
    return;
  }
  ll d = n;
  for (int i = 2; d == n; i++)
    d = pollard_rho(n, i);
  factorize(d, f);
  factorize(n/d, f);
}
```

6.12 Simplex

```
const double EPS = 1e-6;
typedef vector<double> vec;
namespace simplex {
  vector<int> X, Y;
  vector<vec> a;
  vec b, c;
  double z;/// Complexity: O(|N|^2 * |M|) N variables, N restrictions
  int n, m;
  void pivot(int x, int y) {
    swap(X[y], Y[x]);
    b[x] /= a[x][y];
    for(int i = 0; i < m; i++)
      if(i != y)
        a[x][i] /= a[x][y];
    a[x][y] = 1 / a[x][y];
    for(int i = 0; i < n; i++)
      if(i != x && abs(a[i][y]) > EPS) {
      b[i] -= a[i][y] * b[x];
```

```
      for(int j = 0; j < m; j++)
        if(j != y)
          a[i][j] -= a[i][y] * a[x][j];
      a[i][y] =- a[i][y] * a[x][y];
    }
    z += c[y] * b[x];
    for(int i = 0; i < m; i++)
      if(i != y)
        c[i] -= c[y] * a[x][i];
    c[y] =- c[y] * a[x][y];
  }
  /// A is a vector of 1 and 0. B is the limit restriction. C is the factors of
       O.F.
  pair<double, vec> simplex(vector<vec> &A, vec &B, vec &C) {
    a = A; b = B; c = C;
    n = b.size(); m = c.size(); z = 0.0;
    X = vector<int>(m);
    Y = vector<int>(n);
    for(int i = 0; i < m; i++) X[i] = i;
    for(int i = 0; i < n; i++) Y[i] = i + m;
    while(1) {
      int x = -1, y = -1;
      double mn = -EPS;
      for(int i = 0; i < n; i++)
        if(b[i] < mn)
          mn = b[i], x = i;
      if(x < 0) break;
      for(int i = 0; i < m; i++)
        if(a[x][i] < -EPS) { y = i; break; }
      assert(y >= 0); // no sol
      pivot(x, y);
    }
    while(1) {
      double mx = EPS;
      int x = -1,y = -1;
      for(int i = 0; i < m; i++)
        if(c[i] > mx)
          mx = c[i], y = i;
      if(y < 0) break;
      double mn = 1e200;
      for(int i = 0; i < n; i++)
        if(a[i][y] > EPS && b[i] / a[i][y] < mn)
          mn = b[i] / a[i][y], x = i;
      assert(x >= 0); // unbound
      pivot(x, y);
    }
    vec r(m);
    for(int i = 0; i < n; i++)
      if(Y[i] < m)
        r[ Y[i] ] = b[i];
    return make_pair(z, r);
  }
}
```

6.13 Simpson

```
inline lf simpson(lf fl, lf fr, lf fmid, lf l, lf r) {
  return (fl + fr + 4.0 * fmid) * (r - l) / 6.0;
}
lf rsimpson (lf slr, lf fl, lf fr, lf fmid, lf l, lf r) {
        lf mid = (l + r) * 0.5;
        lf fml = f((l + mid) * 0.5);
        lf fmr = f((mid + r) * 0.5);
        lf slm = simpson(fl, fmid, fml, l, mid);
        lf smr = simpson(fmid, fr, fmr, mid, r);
        if (fabs(slr - slm - smr) < eps) return slm + smr;
        return rsimpson(slm, fl, fmid, fml, l, mid) + rsimpson(smr, fmid, fr, fmr,
             mid, r);
}
lf integrate(lf l,lf r) {
        lf mid = (l + r) * .5, fl = f(l), fr = f(r), fmid = f(mid);
        return rsimpson(simpson(fl, fr, fmid, l, r), fl, fr, fmid, l, r);
}
```

6.14 Totient and divisors

```
vector<int> count_divisors_sieve() {
  bitset<mx> is_prime; is_prime.set();
  vector<int> cnt(mx, 1);
  is_prime[0] = is_prime[1] = 0;
  for(int i = 2; i < mx; i++) {
    if(!is_prime[i]) continue;
    cnt[i]++;
    for(int j = i+i; j < mx; j += i) {
      int n = j, c = 1;
      while( n%i == 0 ) n /= i, c++;
      cnt[j] *= c;
      is_prime[j] = 0;
    }
  }
  return cnt;
}
vector<int> euler_phi_sieve() {
  bitset<mx> is_prime; is_prime.set();
  vector<int> phi(mx);
  iota(phi.begin(), phi.end(), 0);
  is_prime[0] = is_prime[1] = 0;
  for(int i = 2; i < mx; i++) {
    if(!is_prime[i]) continue;
    for(int j = i; j < mx; j += i) {
      phi[j] -= phi[j]/i;
      is_prime[j] = 0;
    }
  }
  return phi;
}
```

```
ll euler_phi(ll n) {
  ll ans = n;
  for(ll i = 2; i * i <= n; ++i) {
    if(n % i == 0) {
      ans -= ans / i;
      while(n % i == 0) n /= i;
    }
  }
  if(n > 1) ans -= ans / n;
  return ans;
}
```

7 Network flows

7.1 Blossom

```
struct network {
  struct struct_edge { int v; struct_edge * n; };
  typedef struct_edge* edge;
  int n;
  struct_edge pool[MAXE]; ///2*n*n;
  edge top;
  vector<edge> adj;
  queue<int> q;
  vector<int> f, base, inq, inb, inp, match;
  vector<vector<int>> ed;
  network(int n) : n(n), match(n, -1), adj(n), top(pool), f(n), base(n),
                   inq(n), inb(n), inp(n), ed(n, vector<int>(n)) {}
  void add_edge(int u, int v) {
    if(ed[u][v]) return;
    ed[u][v] = 1;
    top->v = v, top->n = adj[u], adj[u] = top++;
    top->v = u, top->n = adj[v], adj[v] = top++;
  }
  int get_lca(int root, int u, int v) {
    fill(inp.begin(), inp.end(), 0);
    while(1) {
      inp[u = base[u]] = 1;
      if(u == root) break;
      u = f[ match[u] ];
    }
    while(1) {
      if(inp[v = base[v]]) return v;
      else v = f[ match[v] ];
    }
  }
  void mark(int lca, int u) {
    while(base[u] != lca) {
      int v = match[u];
      inb[ base[u ]] = 1;
      inb[ base[v] ] = 1;
      u = f[v];
      if(base[u] != lca) f[u] = v;
    }
  }
  void blossom_contraction(int s, int u, int v) {
    int lca = get_lca(s, u, v);
    fill(inb.begin(), inb.end(), 0);
    mark(lca, u); mark(lca, v);
    if(base[u] != lca) f[u] = v;
    if(base[v] != lca) f[v] = u;
    for(int u = 0; u < n; u++)
      if(inb[base[u]]) {
        base[u] = lca;
        if(!inq[u]) {
            inq[u] = 1;
            q.push(u);
        }
      }
  }
  int bfs(int s) {
    fill(inq.begin(), inq.end(), 0);
    fill(f.begin(), f.end(), -1);
    for(int i = 0; i < n; i++) base[i] = i;
    q = queue<int>();
    q.push(s);
    inq[s] = 1;
   while(q.size()) {
      int u = q.front(); q.pop();
      for(edge e = adj[u]; e; e = e->n) {
        int v = e->v;
        if(base[u] != base[v] && match[u] != v) {
          if((v == s) || (match[v] != -1 && f[match[v]] != -1))
            blossom_contraction(s, u, v);
          else if(f[v] == -1) {
            f[v] = u;
            if(match[v] == -1) return v;
            else if(!inq[match[v]]) {
              inq[match[v]] = 1;
              q.push(match[v]);
            }
          }
        }
      }
    }
    return -1;
  }
  int doit(int u) {
    if(u == -1) return 0;
    int v = f[u];
    doit(match[v]);
    match[v] = u; match[u] = v;
    return u != -1;
  }
  int maximum_matching() {
    int ans = 0;  /// (i < net.match[i]) => means match
    for(int u = 0; u < n; u++)
```

```
    ans += (match[u] == -1) && doit(bfs(u));
  return ans;
  }
};
```

7.2 Dinic

```
struct edge { int v, cap, inv, flow; };
struct network {
  int n, s, t;
  vector<int> lvl;
  vector<vector<edge>> g;
  network(int n) : n(n), lvl(n), g(n) {}
  void add_edge(int u, int v, int c) {
    g[u].push_back({v, c, g[v].size(), 0});
    g[v].push_back({u, 0, g[u].size()-1, c});
  }
  bool bfs() {
    fill(lvl.begin(), lvl.end(), -1);
    queue<int> q;
    lvl[s] = 0;
    for(q.push(s); q.size(); q.pop()) {
      int u = q.front();
      for(auto &e : g[u]) {
        if(e.cap > 0 && lvl[e.v] == -1) {
          lvl[e.v] = lvl[u]+1;
          q.push(e.v);
        }
      }
    }
    return lvl[t] != -1;
  }
  int dfs(int u, int nf) {
    if(u == t) return nf;
    int res = 0;
    for(auto &e : g[u]) {
      if(e.cap > 0 && lvl[e.v] == lvl[u]+1) {
        int tf = dfs(e.v, min(nf, e.cap));
        res += tf; nf -= tf; e.cap -= tf;
        g[e.v][e.inv].cap += tf;
        g[e.v][e.inv].flow -= tf;
        e.flow += tf;
        if(nf == 0) return res;
      }
    }
    if(!res) lvl[u] = -1;
    return res;
  }
  int max_flow(int so, int si, int res = 0) {
    s = so; t = si;
    while(bfs()) res += dfs(s, INT_MAX);
    return res;
  }
};
```

7.3 Maximum flow minimum cost

```
template <class type>
struct mcmf {
  struct edge { int u, v, cap, flow; type cost; };
  int n;
  vector<edge> ed;
  vector<vector<int>> g;
  vector<int> p;
  vector<type> d, phi;
  mcmf(int n) : n(n), g(n), p(n), d(n), phi(n) {}
  void add_edge(int u, int v, int cap, type cost) {
    g[u].push_back(ed.size());
    ed.push_back({u, v, cap, 0, cost});
    g[v].push_back(ed.size());
    ed.push_back({v, u, 0, 0, -cost});
  }
  bool dijkstra(int s, int t) {
    fill(d.begin(), d.end(), INF);
    fill(p.begin(), p.end(), -1);
    set<pair<type, int>> q;
    d[s] = 0;
    for(q.insert({d[s], s}); q.size();) {
      int u = (*q.begin()).second; q.erase(q.begin());
      for(auto v : g[u]) {
        auto &e = ed[v];
        type nd = d[e.u]+e.cost+phi[e.u]-phi[e.v];
        if(0 < (e.cap-e.flow) && nd < d[e.v]) {
          q.erase({d[e.v], e.v});
          d[e.v] = nd; p[e.v] = v;
          q.insert({d[e.v], e.v});
        }
      }
    }
    for(int i = 0; i < n; i++) phi[i] = min(INF, phi[i]+d[i]);
    return d[t] != INF;
  }
  pair<int, type> max_flow(int s, int t) {
    type mc = 0;
    int mf = 0;
    fill(phi.begin(), phi.end(), 0);
    while(dijkstra(s, t)) {
      int flow = INF;
      for(int v = p[t]; v != -1; v = p[ ed[v].u ])
        flow = min(flow, ed[v].cap-ed[v].flow);
      for(int v = p[t]; v != -1; v = p[ ed[v].u ]) {
        edge &e1 = ed[v];
        edge &e2 = ed[v^1];
        mc += e1.cost*flow;
```

```
        e1.flow += flow;
        e2.flow -= flow;
      }
      mf += flow;
    }
    return {mf, mc};
  }
};
```

7.4 Weighted matching

```
typedef int type;
struct matching_weighted {
  int l, r;
  vector<vector<type>> c;
  matching_weighted(int l, int r) : l(l), r(r), c(l, vector<type>(r)) {
    assert(l <= r);
  }
  void add_edge(int a, int b, type cost) { c[a][b] = cost; }
  type matching() {
    vector<type> v(r), d(r); // v: potential
    vector<int> ml(l, -1), mr(r, -1); // matching pairs
    vector<int> idx(r), prev(r);
    iota(idx.begin(), idx.end(), 0);
    auto residue = [&](int i, int j) { return c[i][j]-v[j]; };
    for(int f = 0; f < l; ++f) {
      for(int j = 0; j < r; ++j) {
        d[j] = residue(f, j);
        prev[j] = f;
      }
      type w;
      int j, l;
      for (int s = 0, t = 0;;) {
        if(s == t) {
          l = s;
          w = d[ idx[t++] ];
          for(int k = t; k < r; ++k) {
            j = idx[k];
            type h = d[j];
            if (h <= w) {
              if (h < w) t = s, w = h;
              idx[k] = idx[t];
              idx[t++] = j;
            }
          }
          for (int k = s; k < t; ++k) {
            j = idx[k];
            if (mr[j] < 0) goto aug;
          }
        }
        int q = idx[s++], i = mr[q];
        for (int k = t; k < r; ++k) {
          j = idx[k];
          type h = residue(i, j) - residue(i, q) + w;
          if (h < d[j]) {
            d[j] = h;
            prev[j] = i;
            if(h == w) {
              if(mr[j] < 0) goto aug;
              idx[k] = idx[t];
              idx[t++] = j;
            }
          }
        }
      }
      aug: for (int k = 0; k < l; ++k)
        v[ idx[k] ] += d[ idx[k] ] - w;
      int i;
      do {
        mr[j] = i = prev[j];
        swap(j, ml[i]);
      } while (i != f);
    }
    type opt = 0;
    for (int i = 0; i < l; ++i)
      opt += c[i][ml[i]]; // (i, ml[i]) is a solution
    return opt;
  }
};
```

8 Strings

8.1 Aho corasick

```
struct aho_corasick {
  const static int alpha = 300;
  vector<int> fail, cnt_word;
  vector<vector<int>> trie;
  int nodes;
  aho_corasick(int maxn) : nodes(1), trie(maxn, vector<int>(alpha)),
                           fail(maxn), cnt_word(maxn) {}
  void add(string &s) {
    int u = 1;
    for(auto x : s) {
      int c = x-'a';
      if(!trie[u][c]) trie[u][c] = ++nodes;
      u = trie[u][c];
    }
    cnt_word[u]++;
  }
  int mv(int u, int c){
    while(!trie[u][c]) u = fail[u];
    return trie[u][c];
  }
```

```
  void build() {
    queue<int> q;
    for(int i = 0; i < alpha; ++i) {
      if(trie[1][i]) {
        q.push(trie[1][i]);
        fail[ trie[1][i] ] = 1;
      }
      else trie[1][i] = 1;
    }
    while(q.size()) {
      int u = q.front(); q.pop();
      for(int i = 0; i < alpha; ++i){
        int v = trie[u][i];
        if(v) {
          fail[v] = mv(fail[u], i);
          cnt_word[v] += cnt_word[ fail[v] ];
          q.push(v);
        }
      }
    }
  }
};
```

8.2 Hashing

```
const int MODS[] = { 1001864327, 1001265673 };
const mint BASE(256, 256), ZERO(0, 0), ONE(1, 1);
inline int add(int a, int b, const int& mod) { return a+b >= mod ? a+b-mod : a+b;
     }
inline int sbt(int a, int b, const int& mod) { return a-b < 0 ? a-b+mod : a-b; }
inline int mul(int a, int b, const int& mod) { return 1ll*a*b%mod; }
inline ll operator ! (const mint a) { return (ll(a.first)<<32)|ll(a.second); }
inline mint operator + (const mint a, const mint b) {
  return {add(a.first, b.first, MODS[0]), add(a.second, b.second, MODS[1])};
} /// 1000234999, 1000567999, 1000111997, 1000777121
inline mint operator - (const mint a, const mint b) {
  return {sbt(a.first, b.first, MODS[0]), sbt(a.second, b.second, MODS[1])};
}
inline mint operator * (const mint a, const mint b) {
  return {mul(a.first, b.first, MODS[0]), mul(a.second, b.second, MODS[1])};
}
mint base[MAXN];
void prepare() {
  base[0] = ONE;
  for(int i = 1; i < MAXN; i++) base[i] = base[i-1]*BASE;
}
template <class type>
struct hashing {
  vector<mint> code;
  hashing(type &t) {
    code.resize(t.size()+1);
    code[0] = ZERO;
    for (int i = 1; i < code.size(); ++i)
      code[i] = code[i-1]*BASE + mint{t[i-1], t[i-1]};
  }
  mint query(int l, int r) {
    return code[r+1] - code[l]*base[r-l+1];
  }
};
```

8.3 Kmp automaton

```
const int alpha = 256;
int aut[102][alpha];
void kmp_automaton(string &t) {
  vector<int> phi = get_phi(t);
  for(int i = 0; i <= t.size(); ++i) {
    for(int c = 0; c < alpha; ++c) {
      if(i == t.size() || (i > 0 && c != t[i])) aut[i][c] = aut[ phi[i-1] ][c];
      else aut[i][c] = i + (c == t[i]);
    }
  }
}
```

8.4 Kmp

```
vector<int> get_phi(string &p) {
  vector<int> phi(p.size());
  phi[0] = 0;
  for(int i = 1, j = 0; i < p.size(); ++i ) {
    while(j > 0 && p[i] != p[j] ) j = phi[j-1];
    if(p[i] == p[j]) ++j;
    phi[i] = j;
  }
  return phi;
}
int get_match(string &t, string &p) {
  vector<int> phi = get_phi(p);
  int matches = 0;
  for(int i = 0, j = 0; i < t.size(); ++i ) {
    while(j > 0 && t[i] != p[j] ) j = phi[j-1];
    if(t[i] == p[j]) ++j;
    if(j == p.size()) {
      matches++;
      j = phi[j-1];
    }
  }
  return matches;
}
```

8.5 Manacher

```
vector<int> manacher(string &s) {
  int n = s.size(), p = 0, pr = -1;
  vector<int> from(2*n-1);
  for(int i = 0; i < 2*n-1; ++i) {
    int r = i <= 2*pr ? min(p - from[2*p - i], pr) : i/2;
    int l = i - r;
    while(l > 0 && r < n-1 && s[l-1] == s[r+1]) --l, ++r;
    from[i] = l;
    if (r > pr) {
      pr = r;
      p = i;
    } ///len = to - from[i] + 1 = i - 2 * from[i] + 1;
  } ///to = i - from[i];
  return from;
}
```

8.6 Minimun expression

```
int minimum_expression(string s) {
  s = s+s;
  int len = s.size(), i = 0, j = 1, k = 0;
  while (i + k < len && j + k < len) {
    if (s[i+k] == s[j+k]) k++;
    else if (s[i+k] > s[j+k]) {
      i = i+k+1;
      if(i <= j) i = j+1; k = 0;
    }
    else if (s[i+k] < s[j+k]) {
      j = j+k+1;
      if(j <= i) j = i+1; k = 0;
    }
  }
  return min(i, j);
}
```

8.7 Suffix array

```
struct suffix_array {
  const static int alpha = 300;
  int mx, n;
  string s;
  vector<int> pos, tpos, sa, tsa, lcp;
  suffix_array(string t) {
    s = t+"$"; n = s.size(); mx = max(alpha, n)+2;
    pos = tpos = tsa = sa = lcp = vector<int>(n);
  }
  bool check(int i, int gap) {
    if(pos[ sa[i-1] ] != pos[ sa[i] ]) return true;
    if(sa[i-1]+gap < n && sa[i]+gap < n)
      return (pos[ sa[i-1]+gap ] != pos[ sa[i]+gap ]);
    return true;
  }
  void radix_sort(int k) {
    vector<int> cnt(mx);
    for(int i = 0; i < n; i++)
      cnt[(i+k < n) ? pos[i+k]+1 : 1]++;
    for(int i = 1; i < mx; i++)
      cnt[i] += cnt[i-1];
    for(int i = 0; i < n; i++)
      tsa[cnt[(sa[i]+k < n) ? pos[sa[i]+k] : 0]++] = sa[i];
    sa = tsa;
  }
  void build_sa() {
    for(int i = 0; i < n; i++) {
      sa[i] = i;
      pos[i] = s[i];
    }
    for(int gap = 1; gap < n; gap <<= 1) {
      radix_sort(gap);
      radix_sort(0);
      tpos[ sa[0] ] = 0;
      for(int i = 1; i < n; i++)
        tpos[ sa[i] ] = tpos[ sa[i-1] ] + check(i, gap);
      pos = tpos;
      if(pos[ sa[n-1] ] == n-1) break;
    }
  }
  void build_lcp() {
    int k = 0;
    lcp[0] = 0;
    for(int i = 0; i < n; i++) {
      if(pos[i] == 0) continue;
      while(s[i+k] == s[ sa[ pos[i]-1 ]+k ]) k++;
      lcp[ pos[i] ] = k;
      k = max(0, k-1);
    }
  }
  int& operator[] ( int i ){ return sa[i]; }
};
```

8.8 Suffix automaton

```
struct suffix_automaton {
  struct node {
    int len, link; bool end;
    map<char, int> next;
  };
  vector<node> sa;
  int last;
```

```
  suffix_automaton() {}
  suffix_automaton(string s) {
    sa.reserve(s.size()*2);
    last = add_node();
    sa[last].len = 0;
    sa[last].link = -1;
    for(int i = 0; i < s.size(); ++i)
      sa_append(s[i]);
    ///t0 is not suffix
    for(int cur = last; cur; cur = sa[cur].link)
      sa[cur].end = 1;
  }
  int add_node() {
    sa.push_back({});
    return sa.size()-1;
  }
  void sa_append(char c) {
    int cur = add_node();
    sa[cur].len = sa[last].len + 1;
    int p = last;
    while(p != -1 && !sa[p].next[c] ){
      sa[p].next[c] = cur;
      p = sa[p].link;
    }
    if(p == -1) sa[cur].link = 0;
    else {
      int q = sa[p].next[c];
      if(sa[q].len == sa[p].len+1) sa[cur].link = q;
      else {
        int clone = add_node();
        sa[clone] = sa[q];
        sa[clone].len = sa[p].len+1;
        sa[q].link = sa[cur].link = clone;
        while(p != -1 && sa[p].next[c] == q) {
          sa[p].next[c] = clone;
          p = sa[p].link;
        }
      }
    }
    last = cur;
  }
  node& operator[](int i) { return sa[i]; }
};
```

8.9 Z algorithm

```
vector<int> z_algorithm (string s) {
  int n = s.size();
  vector<int> z(n);
  int x = 0, y = 0;
  for(int i = 1; i < n; ++i) {
    z[i] = max(0, min(z[i-x], y-i+1));
    while (i+z[i] < n && s[z[i]] == s[i+z[i]])
      x = i, y = i+z[i], z[i]++;
  }
  return z;
}
```

9 Utilities

9.1 Hash STL

```
struct Hash {
  size_t operator()(const pii &x) const {
    return (size_t) x.first * 37U + (size_t) x.second;
  }
        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_map<pii, T, Hash> mp;
mp.reserve(1024); /// power of 2
mp.max_load_factor(0.25);
```

9.2 Pragma optimizations

```
#pragma GCC optimize ("O3") #pragma GCC target ("sse4")
#pragma GCC target ("avx,tune=native")
```

9.3 Random

```
// Declare number generator
  mt19937 / mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count())
  // or
  random_device rd
  mt19937 / mt19937_64 rng(rd())
// Use it to shuffle a vector
shuffle(permutation.begin(), permutation.end(), rng)
// Use it to generate a random number between [fr, to]
uniform_int_distribution<T> / uniform_real_distribution<T> dis(fr, to);
  dis(rng)
```