Contest (1)

template.cpp

41 lines

```cpp
#include <bits/stdc++.h>
using namespace std;

#ifdef DEBUG
int recur_depth = 0;
#define db(x) {++recur_depth; auto x_
   =x; --recur_depth; cerr<<string(
   recur_depth, '\t')<<"\e[91m"<<
   __func__<<":"<<__LINE__<<"\t"<<#x
   <<" = "<<x_<<"\e[39m"<<endl;}
#else
#define db(x)
#endif
template<typename Ostream, typename
   Cont>
typename enable_if<is_same<Ostream,
   ostream>::value, Ostream&>::type
   operator<<(Ostream& os,  const
   Cont& v){
os<<"[";for(auto& x:v){os<<x<<", ";}
   return os<<"]";
}
template<typename Ostream, typename
   ...Ts>
Ostream& operator<<(Ostream& os,
   const pair<Ts...>& p){
return os<<"{"<<p.first<<", "<<p.
   second<<"}";
}

void solve(){}

int main() {
#ifdef DEBUG
auto started = std::chrono::
   high_resolution_clock::now();
#endif
cin.tie(0)->sync_with_stdio(0);
cin.exceptions(cin.failbit);
int t;
t = 1;
cin >> t;
for (int I = 1; I <= t; I++)
{
#ifdef DEBUG
cerr << "Case #" << I << ":\n";
#endif
solve();
}
#ifdef DEBUG
auto done = std::chrono::
   high_resolution_clock::now();
cerr << "Time = " << std::chrono::
   duration_cast<std::chrono::
   milliseconds>(done - started).
   count() << " ms\n";
#endif
}
```

.bashrc

4 lines

```bash
function cpp()
{
g++ -DLOCAL -std=c++17 -O2 -Wall -
   Wextra -pedantic -Wshadow -Wformat
   =2 -Wfloat-equal -Wconversion -
   Wlogical-op -Wshift-overflow=2 -
   Wduplicated-cond -Wcast-qual -
   Wcast-align -D_GLIBCXX_DEBUG -
   D_GLIBCXX_DEBUG_PEDANTIC -
   D_FORTIFY_SOURCE=2 -fsanitize=
   address -fsanitize=undefined -fno-
   sanitize-recover -fstack-protector
    -o $1{,.cpp}
}
```

Mathematics (2)

2.1 Equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The extremum is given by $x = -b/2a$.

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \Rightarrow \begin{aligned} x &= \frac{ed - bf}{ad - bc} \\ y &= \frac{af - ec}{ad - bc} \end{aligned}$$

In general, given an equation $Ax = b$, the solution to a variable x_i is given by

$$x_i = \frac{\det A_i'}{\det A}$$

where A_i' is A with the i'th column replaced by b.

2.2 Trigonometry

$$\sin(v+w) = \sin v \cos w + \cos v \sin w$$
$$\cos(v+w) = \cos v \cos w - \sin v \sin w$$

$$\tan(v+w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$
$$\sin v + \sin w = 2 \sin \frac{v+w}{2} \cos \frac{v-w}{2}$$
$$\cos v + \cos w = 2 \cos \frac{v+w}{2} \cos \frac{v-w}{2}$$

$$(V+W)\tan(v-w)/2 = (V-W)\tan(v+w)/2$$

where V, W are lengths of sides opposite angles v, w.

$$a \cos x + b \sin x = r \cos(x - \phi)$$
$$a \sin x + b \cos x = r \sin(x + \phi)$$

where $r = \sqrt{a^2+b^2}$, $\phi = \operatorname{atan2}(b, a)$.

2.3 Geometry

2.3.1 Triangles

Side lengths: a, b, c

Semiperimeter: $p = \dfrac{a+b+c}{2}$

Area: $A = \sqrt{p(p-a)(p-b)(p-c)}$

Circumradius: $R = \dfrac{abc}{4A}$

Inradius: $r = \dfrac{A}{p}$

Length of median (divides triangle into two equal-area triangles):
$m_a = \frac{1}{2}\sqrt{2b^2+2c^2-a^2}$

Length of bisector (divides angles in two):
$$s_a = \sqrt{bc\left[1 - \left(\frac{a}{b+c}\right)^2\right]}$$

Law of sines:
$$\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c} = \frac{1}{2R}$$

Law of cosines: $a^2 = b^2 + c^2 - 2bc\cos\alpha$

Law of tangents: $\dfrac{a+b}{a-b} = \dfrac{\tan\frac{\alpha+\beta}{2}}{\tan\frac{\alpha-\beta}{2}}$

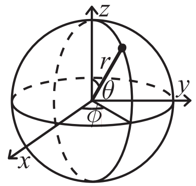
2.3.2 Quadrilaterals

With side lengths a, b, c, d, diagonals e, f, diagonals angle θ, area A and magic flux $F = b^2 + d^2 - a^2 - c^2$:

$$4A = 2ef \cdot \sin\theta = F \tan\theta = \sqrt{4e^2f^2 - F^2}$$

For cyclic quadrilaterals the sum of opposite angles is $180°$, $ef = ac + bd$, and $A = \sqrt{(p-a)(p-b)(p-c)(p-d)}$.

2.3.3 Spherical coordinates



$$x = r \sin\theta \cos\phi \qquad r = \sqrt{x^2+y^2+z^2}$$
$$y = r \sin\theta \sin\phi \qquad \theta = \operatorname{acos}(z/\sqrt{x^2+y^2+z^2})$$
$$z = r\cos\theta \qquad \phi = \operatorname{atan2}(y, x)$$

2.4 Sums

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c-1}, c \neq 1$$

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$
$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(2n+1)(n+1)}{6}$$
$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$
$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

2.5 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots, \ (-\infty < x < \infty)$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, \ (-1 < x \leq 1)$$

$$\sqrt{1+x} = 1+\frac{x}{2}-\frac{x^2}{8}+\frac{2x^3}{32}-\frac{5x^4}{128}+\dots, \ (-1 \le x \le 1)$$

$$\sin x = x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\dots, \ (-\infty < x < \infty)$$

$$\cos x = 1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\dots, \ (-\infty < x < \infty)$$

2.6 Probability theory

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x. It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x x p_X(x)$ and variance $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent X and Y,

$$V(aX + bY) = a^2V(X) + b^2V(Y).$$

Data structures (3)

OrderStatisticTree.h

Description: Policy based data stucture
Time: $\mathcal{O}(\log N)$

<bits/extc++.h> b69a4d, 7 lines

```
using namespace __gnu_pbds;
typedef tree< int, //key type
null_type, // null_type:set,mapped:
    map
less_equal<int>, // comparator
rb_tree_tag,
    tree_order_statistics_node_update>
     ordered_set;
//1)find_by_order(k) returns element
    at index k (zero indexed)
//2)order_of_key(element) returns the
     number of elements strictly
    smaller than element
```

HashMap.h

Description: Hash map with mostly the same API as unordered_map, but ~3x faster. Uses 1.5x memory. Initial capacity must be a power of 2 (if provided).

d77092, 7 lines

```
#include <bits/extc++.h>
// To use most bits rather than just
    the lowest ones:
struct chash { // large odd number
    for C
  const uint64_t C = ll(4e18 * acos
      (0)) | 71;
  ll operator()(ll x) const { return
      __builtin_bswap64(x*C); }
};
__gnu_pbds::gp_hash_table<ll,int,
    chash> h({},{},{},{},{1<<16});
```

SegmentTree.h

Description: Zero indexed segment tree
Time: $\mathcal{O}(\log N)$

df703c, 74 lines

```
class Node
{public:
int value;
Node() { value = INT_MAX; }; //
    Identity
explicit Node(int v) { value = v; }
static Node mergeSegNodes(const Node
    &a, const Node &b)
{Node res;
res.value = min(a.value, b.value);
return res;}
};

template <typename segNode>
class segTree
{int size;
vector<segNode> Seg;
void build(int node, int start, int
    end, const vector<segNode> &Base)
    // Recursively Builds the tree
{if (start == end)
{Seg[node] = Base[start];
return;}
int mid = (start + end) >> 1;
build(node + 1, start, mid, Base);
build(node + 2 * (mid - start + 1),
    mid + 1, end, Base);
Seg[node] = segNode::mergeSegNodes(
    Seg[node + 1], Seg[node + 2 * (mid
     - start + 1)]);
}
```

```
segNode rQuery(int node, int start,
    int end, int qstart, int qend)
    const // Range Query
{if (qend < start || qstart > end ||
    start > end)
return segNode();
if (qstart <= start && end <= qend)
return Seg[node];
int mid = (start + end) >> 1;
segNode l, r;
if (qstart <= mid)
l = rQuery(node + 1, start, mid,
    qstart, qend);
if (qend >= mid + 1)
r = rQuery(node + 2 * (mid - start +
    1), mid + 1, end, qstart, qend);
return segNode::mergeSegNodes(l, r);
}

void pUpdate(int node, int start, int
     end, int pos, segNode val, int
    type)
{if (start == end)
{Seg[node] = type ? segNode::
    mergeSegNodes(Seg[node], val) :
    val;
return;}
int mid = (start + end) >> 1;
if (pos <= mid)
pUpdate(node + 1, start, mid, pos,
    val, type);
else
pUpdate(node + 2 * (mid - start + 1),
     mid + 1, end, pos, val, type);
Seg[node] = segNode::mergeSegNodes(
    Seg[node + 1], Seg[node + 2 * (mid
     - start + 1)]);
}

public:
segTree() : segTree(0){};
explicit segTree(int n) : size(n),
    Seg(n << 1){};
explicit segTree(const vector<segNode
    > &Base) : size(Base.size())
{Seg = vector<segNode>(size << 1);
build(1, 0, size - 1, Base);}

segNode get(int pos) const
{assert(pos >= 0 && pos < size);
return rQuery(1, 0, size - 1, pos,
    pos);}

segNode get(int left, int right)
    const
{assert(left <= right && left >= 0 &&
     right < size);
return rQuery(1, 0, size - 1, left,
    right);}

void update(int pos, segNode val) //
    Updates according to merge
{assert(pos >= 0 && pos < size);
pUpdate(1, 0, size - 1, pos, val, 1);
    }

void set(int pos, segNode val) //
    Force sets value of node
{assert(pos >= 0 && pos < size);
pUpdate(1, 0, size - 1, pos, val, 0);
    }
};
```

LazySegmentTree.h

Time: $\mathcal{O}(\log N)$.

0735d9, 100 lines

```
class Node
{public:
long long value;
Node() { value = 0; }; // Identity
explicit Node(long long v) { value =
    v; }
static Node mergeSegNodes(const Node
    &a, const Node &b)
{Node res;
res.value = (a.value + b.value);
return res;}
void mergeLazyNodes(const Node &b)
{this->value += b.value;}
void mergeSegLazy(const Node &b,
    const int &l, const int &r)
{this->value += (r - l + 1) * b.value
    ;}};

template <typename segNode>
class segTree
{int size;
vector<segNode> Seg;
vector<segNode> Lazy;
vector<bool> isLazy;

void propagate(int node, int l, int r
    )
{if (isLazy[node])
{isLazy[node] = false;
Seg[node].mergeSegLazy(Lazy[node], l,
     r);
if (l != r)
{int mid = (l + r) >> 1;
Lazy[node + 1].mergeLazyNodes(Lazy[
    node]);
```

```
Lazy[node + 2 * (mid - l + 1)].
   mergeLazyNodes(Lazy[node]);
isLazy[node + 1] = true;
isLazy[node + 2 * (mid - l + 1)] =
   true;}
Lazy[node] = segNode();}}

void build(int node, int start, int
   end, const vector<segNode> &Base)
   // Recursively Builds the tree
{if (start == end)
{Seg[node] = Base[start];return;}
int mid = (start + end) >> 1;
build(node + 1, start, mid, Base);
build(node + 2 * (mid - start + 1),
   mid + 1, end, Base);
Seg[node] = segNode::mergeSegNodes(
   Seg[node + 1], Seg[node + 2 * (mid
    - start + 1)]);
}

segNode rQuery(int node, int start,
   int end, int qstart, int qend) //
   Range Query
{propagate(node, start, end);
if (qend < start || qstart > end ||
   start > end)
return segNode();
if (qstart <= start && end <= qend)
return Seg[node];
int mid = (start + end) >> 1;
segNode l, r;
propagate(node + 1, start, mid);
propagate(node + 2 * (mid - start +
   1), mid + 1, end);
if (qstart <= mid)
l = rQuery(node + 1, start, mid,
   qstart, qend);
if (qend >= mid + 1)
r = rQuery(node + 2 * (mid - start +
   1), mid + 1, end, qstart, qend);
return segNode::mergeSegNodes(l, r);}

void rUpdate(int node, int start, int
    end, int qstart, int qend,
   segNode val)
{propagate(node, start, end);
if (qend < start || qstart > end ||
   start > end)
return;
if (qstart <= start && end <= qend)
{isLazy[node] = true;
Lazy[node] = val;
propagate(node, start, end);
return;
}
int mid = (start + end) >> 1;
propagate(node + 1, start, mid);
propagate(node + 2 * (mid - start +
   1), mid + 1, end);
if (qstart <= mid)
rUpdate(node + 1, start, mid, qstart,
    qend, val);
if (qend >= mid + 1)
rUpdate(node + 2 * (mid - start + 1),
    mid + 1, end, qstart, qend, val);
Seg[node] = segNode::mergeSegNodes(
   Seg[node + 1], Seg[node + 2 * (mid
    - start + 1)]);
}

public:
segTree() : segTree(0){};
explicit segTree(int n) : size(n),
   Seg(n << 1), Lazy(n << 1), isLazy(
   n << 1){};
explicit segTree(const vector<segNode
   > &Base) : size(Base.size())
{Seg = vector<segNode>(size << 1);
Lazy = vector<segNode>(size << 1);
isLazy = vector<bool>(size << 1);
build(1, 0, size - 1, Base);}

segNode get(int pos)
{assert(pos >= 0 && pos < size);
return rQuery(1, 0, size - 1, pos,
   pos);}
segNode get(int left, int right)
{assert(left <= right && left >= 0 &&
    right < size);
return rQuery(1, 0, size - 1, left,
   right);}
void update(int pos, segNode val)
{assert(pos >= 0 && pos < size);
rUpdate(1, 0, size - 1, pos, pos, val
   );}
void update(int left, int right,
   segNode val)
{assert(left <= right && left >= 0 &&
    right < size);
rUpdate(1, 0, size - 1, left, right,
   val);}
};
```

UnionFind.h

Description: Disjoint-set data structure.
Time: $\mathcal{O}(\alpha(N))$

7aa27c, 14 lines

```
struct UF {
  vi e;
  UF(int n) : e(n, -1) {}
```

```
  bool sameSet(int a, int b) { return
       find(a) == find(b); }
  int size(int x) { return -e[find(x)
     ]; }
  int find(int x) { return e[x] < 0 ?
       x : e[x] = find(e[x]); }
  bool join(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return false;
    if (e[a] > e[b]) swap(a, b);
    e[a] += e[b]; e[b] = a;
    return true;
  }
};
```

UnionFindRollback.h

Description: Disjoint-set data structure with undo. If undo is not needed, skip st, time() and rollback().
Usage: `int t = uf.time(); ...; uf.rollback(t);`
Time: $\mathcal{O}(\log(N))$

de4ad0, 21 lines

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

LineContainer.h

Description: Container where you can add lines of the form kx+m, and query maximum values at points x. Useful for dynamic programming ("convex hull trick").
Time: $\mathcal{O}(\log N)$

8ec1c7, 30 lines

```
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const
       { return k < o.k; }
  bool operator<(ll x) const { return
       p < x; }
};

struct LineContainer : multiset<Line,
     less<>> {
  // (for doubles, use inf = 1/.0,
       div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored
       division
    return a / b - ((a ^ b) < 0 && a
         % b); }
  bool isect(iterator x, iterator y)
       {
    if (y == end()) return x->p = inf
         , 0;
    if (x->k == y->k) x->p = x->m > y
         ->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k
          - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z
         ++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y)
         ) isect(x, y = erase(y));
    while ((y = x) != begin() && (--x
         )->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

FenwickTree.h

Description: Computes partial sums a[0] + a[1] + ... + a[pos - 1], and updates single elements a[i], taking the difference between the old and new value.
Time: Both operations are $\mathcal{O}(\log N)$.

e62fac, 22 lines

```
struct FT {
  vector<ll> s;
  FT(int n) : s(n) {}
  void update(int pos, ll dif) { // a
       [pos] += dif
    for (; pos < sz(s); pos |= pos +
         1) s[pos] += dif;
  }
```

```
  ll query(int pos) { // sum of values in [0, pos)
    ll res = 0;
    for (; pos > 0; pos &= pos - 1) res += s[pos-1];
    return res;
  }
  int lower_bound(ll sum) {// min pos st sum of [0, pos] >= sum
    // Returns n if no sum is >= sum, or -1 if empty sum is.
    if (sum <= 0) return -1;
    int pos = 0;
    for (int pw = 1 << 25; pw; pw >>= 1) {
      if (pos + pw <= sz(s) && s[pos + pw-1] < sum)
        pos += pw, sum -= s[pos-1];
    }
    return pos;
  }
};
```

FenwickTree2d.h

Description: Computes sums a[i,j] for all i<I, j<J, and increases single elements a[i,j]. Requires that the elements to be updated are known in advance (call fakeUpdate() before init()).

Time: $\mathcal{O}\left(\log^2 N\right)$. (Use persistent segment trees for $\mathcal{O}(\log N)$.)

"FenwickTree.h" 157f07, 22 lines

```
struct FT2 {
  vector<vi> ys; vector<FT> ft;
  FT2(int limx) : ys(limx) {}
  void fakeUpdate(int x, int y) {
    for (; x < sz(ys); x |= x + 1) ys[x].push_back(y);
  }
  void init() {
    for (vi& v : ys) sort(all(v)), ft.emplace_back(sz(v));
  }
  int ind(int x, int y) {
    return (int)(lower_bound(all(ys[x]), y) - ys[x].begin()); }
  void update(int x, int y, ll dif) {
    for (; x < sz(ys); x |= x + 1)
      ft[x].update(ind(x, y), dif);
  }
  ll query(int x, int y) {
    ll sum = 0;
    for (; x; x &= x - 1)
      sum += ft[x-1].query(ind(x-1, y));
    return sum;
  }
};
```

RMQ.h

Description: Range Minimum Queries on an array. Returns min(V[a], V[a + 1], ... V[b - 1]) in constant time.

Usage: RMQ rmq(values);
rmq.query(inclusive, exclusive);

Time: $\mathcal{O}(|V| \log |V| + Q)$

510c32, 16 lines

```
template<class T>
struct RMQ {
  vector<vector<T>> jmp;
  RMQ(const vector<T>& V) : jmp(1, V) {
    for (int pw = 1, k = 1; pw * 2 <= sz(V); pw *= 2, ++k) {
      jmp.emplace_back(sz(V) - pw * 2 + 1);
      rep(j,0,sz(jmp[k]))
        jmp[k][j] = min(jmp[k - 1][j], jmp[k - 1][j + pw]);
    }
  }
  T query(int a, int b) {
    assert(a < b); // or return inf if a == b
    int dep = 31 - __builtin_clz(b - a);
    return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
  }
};
```

MoQueries.h

Description: Answer interval or tree path queries by finding an approximate TSP through the queries, and moving from one query to the next by adding/removing points at the ends. If values are on tree edges, change `step` to add/remove the edge (a, c) and remove the initial `add` call (but keep `in`).

Time: $\mathcal{O}\left(N\sqrt{Q}\right)$

a12ef4, 49 lines

```
void add(int ind, int end) { ... } // add a[ind] (end = 0 or 1)
void del(int ind, int end) { ... } // remove a[ind]
int calc() { ... } // compute current answer

vi mo(vector<pii> Q) {
  int L = 0, R = 0, blk = 350; // ~N/sqrt(Q)
  vi s(sz(Q)), res = s;
#define K(x) pii(x.first/blk, x.second ^ -(x.first/blk & 1))
  iota(all(s), 0);
```

```
  sort(all(s), [&](int s, int t){
     return K(Q[s]) < K(Q[t]); });
  for (int qi : s) {
    pii q = Q[qi];
    while (L > q.first) add(--L, 0);
    while (R < q.second) add(R++, 1);
    while (L < q.first) del(L++, 0);
    while (R > q.second) del(--R, 1);
    res[qi] = calc();
  }
  return res;
}

vi moTree(vector<array<int, 2>> Q,
    vector<vi>& ed, int root=0){
  int N = sz(ed), pos[2] = {}, blk =
     350; // ~N/sqrt(Q)
  vi s(sz(Q)), res = s, I(N), L(N), R
     (N), in(N), par(N);
  add(0, 0), in[0] = 1;
  auto dfs = [&](int x, int p, int
     dep, auto& f) -> void {
    par[x] = p;
    L[x] = N;
    if (dep) I[x] = N++;
    for (int y : ed[x]) if (y != p) f
       (y, x, !dep, f);
    if (!dep) I[x] = N++;
    R[x] = N;
  };
  dfs(root, -1, 0, dfs);
#define K(x) pii(I[x[0]] / blk, I[x
    [1]] ^ -(I[x[0]] / blk & 1))
  iota(all(s), 0);
  sort(all(s), [&](int s, int t){
     return K(Q[s]) < K(Q[t]); });
  for (int qi : s) rep(end,0,2) {
    int &a = pos[end], b = Q[qi][end
       ], i = 0;
#define step(c) { if (in[c]) { del(a,
    end); in[a] = 0; } \
                   else { add(c, end);
                      in[c] = 1; } a
                     = c; }
    while (!(L[b] <= L[a] && R[a] <=
       R[b]))
      I[i++] = b, b = par[b];
    while (a != b) step(par[a]);
    while (i--) step(I[i]);
    if (end) res[qi] = calc();
  }
  return res;
}
```

Numerical (4)

4.1 Polynomials and recurrences

Polynomial.h

c9b7b0, 17 lines

```
struct Poly {
  vector<double> a;
  double operator()(double x) const {
    double val = 0;
    for (int i = sz(a); i--;) (val *=
        x) += a[i];
    return val;
  }
  void diff() {
    rep(i,1,sz(a)) a[i-1] = i*a[i];
    a.pop_back();
  }
  void divroot(double x0) {
    double b = a.back(), c; a.back()
       = 0;
    for(int i=sz(a)-1; i--;) c = a[i
       ], a[i] = a[i+1]*x0+b, b=c;
    a.pop_back();
  }
};
```

PolyRoots.h

Description: Finds the real roots to a polynomial.

Usage: `polyRoots({{2,-3,1}},-1e9,1e9)`
`// solve x^2-3x+2 = 0`

Time: $\mathcal{O}\left(n^2 \log(1/\epsilon)\right)$

"Polynomial.h" b00bfe, 23 lines

```
vector<double> polyRoots(Poly p,
    double xmin, double xmax) {
  if (sz(p.a) == 2) { return {-p.a
     [0]/p.a[1]}; }
  vector<double> ret;
  Poly der = p;
  der.diff();
  auto dr = polyRoots(der, xmin, xmax
     );
  dr.push_back(xmin-1);
  dr.push_back(xmax+1);
  sort(all(dr));
  rep(i,0,sz(dr)-1) {
    double l = dr[i], h = dr[i+1];
    bool sign = p(l) > 0;
    if (sign ^ (p(h) > 0)) {
      rep(it,0,60) { // while (h - l
         > 1e-8)
        double m = (l + h) / 2, f = p
           (m);
        if ((f <= 0) ^ sign) l = m;
```

```
      else h = m;
    }
    ret.push_back((l + h) / 2);
  }
 }
 return ret;
}
```

PolyInterpolate.h

Description: Given n points (x[i], y[i]), computes an n-1-degree polynomial p that passes through them: $p(x) = a[0]*x^0 + ... + a[n-1]*x^{n-1}$. For numerical precision, pick $x[k] = c*\cos(k/(n-1)*\pi), k = 0 \ldots n-1$.

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

08bf48, 13 lines

```
typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
  vd res(n), temp(n);
  rep(k,0,n-1) rep(i,k+1,n)
    y[i] = (y[i] - y[k]) / (x[i] - x[
       k]);
  double last = 0; temp[0] = 1;
  rep(k,0,n) rep(i,0,n) {
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
}
```

4.2 Matrices

Determinant.h

Description: Calculates determinant of a matrix. Destroys the matrix.

Time: $\mathcal{O}(N^3)$

bd5cec, 15 lines

```
double det(vector<vector<double>>& a)
    {
  int n = sz(a); double res = 1;
  rep(i,0,n) {
    int b = i;
    rep(j,i+1,n) if (fabs(a[j][i]) >
       fabs(a[b][i])) b = j;
    if (i != b) swap(a[i], a[b]), res
        *= -1;
    res *= a[i][i];
    if (res == 0) return 0;
    rep(j,i+1,n) {
      double v = a[j][i] / a[i][i];
      if (v != 0) rep(k,i+1,n) a[j][k
         ] -= v * a[i][k];
    }
  }
  return res;
}
```

IntDeterminant.h

Description: Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.

Time: $\mathcal{O}(N^3)$

3313dc, 18 lines

```
const ll mod = 12345;
ll det(vector<vector<ll>>& a) {
  int n = sz(a); ll ans = 1;
  rep(i,0,n) {
    rep(j,i+1,n) {
      while (a[j][i] != 0) { // gcd
         step
        ll t = a[i][i] / a[j][i];
        if (t) rep(k,i,n)
          a[i][k] = (a[i][k] - a[j][k
             ] * t) % mod;
        swap(a[i], a[j]);
        ans *= -1;
      }
    }
    ans = ans * a[i][i] % mod;
    if (!ans) return 0;
  }
  return (ans + mod) % mod;
}
```

MatrixInverse.h

Description: Invert matrix A. Returns rank; result is stored in A unless singular (rank < n). Can easily be extended to prime moduli; for prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.

Time: $\mathcal{O}(n^3)$

ebfff6, 35 lines

```
int matInv(vector<vector<double>>& A)
    {
  int n = sz(A); vi col(n);
  vector<vector<double>> tmp(n,
     vector<double>(n));
  rep(i,0,n) tmp[i][i] = 1, col[i] =
     i;

  rep(i,0,n) {
    int r = i, c = i;
    rep(j,i,n) rep(k,i,n)
      if (fabs(A[j][k]) > fabs(A[r][c
         ]))
        r = j, c = k;
    if (fabs(A[r][c]) < 1e-12) return
       i;
    A[i].swap(A[r]); tmp[i].swap(tmp[
       r]);
    rep(j,0,n)
```

```
      swap(A[j][i], A[j][c]), swap(
          tmp[j][i], tmp[j][c]);
    swap(col[i], col[c]);
    double v = A[i][i];
    rep(j,i+1,n) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      rep(k,i+1,n) A[j][k] -= f*A[i][
          k];
      rep(k,0,n) tmp[j][k] -= f*tmp[i
          ][k];
    }
    rep(j,i+1,n) A[i][j] /= v;
    rep(j,0,n) tmp[i][j] /= v;
    A[i][i] = 1;
  }

  for (int i = n-1; i > 0; --i) rep(j
      ,0,i) {
    double v = A[j][i];
    rep(k,0,n) tmp[j][k] -= v*tmp[i][
        k];
  }

  rep(i,0,n) rep(j,0,n) A[col[i]][col
      [j]] = tmp[i][j];
  return n;
}
```

Number theory (5)

5.1 Modular arithmetic

ModInverse.h

Description: Pre-computation of modular inverses. Assumes LIM $\leq$ mod and that mod is a prime.

6f684f, 3 lines

```
const ll mod = 1000000007, LIM =
    200000;
ll* inv = new ll[LIM] - 1; inv[1] =
    1;
rep(i,2,LIM) inv[i] = mod - (mod / i)
     * inv[mod % i] % mod;
```

ModPow.h

b83e45, 8 lines

```
const ll mod = 1000000007; // faster
    if const

ll modpow(ll b, ll e) {
  ll ans = 1;
  for (; e; b = b * b % mod, e /= 2)
    if (e & 1) ans = ans * b % mod;
  return ans;
}
```

5.2 Primality

FastEratosthenes.h

Description: Prime sieve for generating all primes smaller than LIM.
Time: LIM=1e9 $\approx$ 1.5s

6b2912, 20 lines

```
const int LIM = 1e6;
bitset<LIM> isPrime;
vi eratosthenes() {
  const int S = (int)round(sqrt(LIM))
      , R = LIM / 2;
  vi pr = {2}, sieve(S+1); pr.reserve
      (int(LIM/log(LIM)*1.1));
  vector<pii> cp;
  for (int i = 3; i <= S; i += 2) if
      (!sieve[i]) {
    cp.push_back({i, i * i / 2});
    for (int j = i * i; j <= S; j +=
        2 * i) sieve[j] = 1;
  }
  for (int L = 1; L <= R; L += S) {
    array<bool, S> block{};
    for (auto &[p, idx] : cp)
      for (int i=idx; i < S+L; idx =
          (i+=p)) block[i-L] = 1;
    rep(i,0,min(S, R - L))
      if (!block[i]) pr.push_back((L
          + i) * 2 + 1);
  }
  for (int i : pr) isPrime[i] = 1;
  return pr;
}
```

5.3 Divisibility

euclid.h

Description: Finds two integers x and y, such that $ax + by = \gcd(a, b)$. If you just need gcd, use the built in __gcd instead. If a and b are coprime, then x is the inverse of a (mod b).

33ba8f, 5 lines

```
ll euclid(ll a, ll b, ll &x, ll &y) {
  if (!b) return x = 1, y = 0, a;
```

```
  ll d = euclid(b, a % b, y, x);
  return y -= a/b * x, d;
}
```

5.3.1 Bézout's identity

For $a \neq, b \neq 0$, then $d = gcd(a, b)$ is the smallest positive integer for which there are integer solutions to

$$ax + by = d$$

If (x, y) is one solution, then all solutions are given by

$$\left(x + \frac{kb}{\gcd(a,b)}, y - \frac{ka}{\gcd(a,b)}\right), \quad k \in \mathbb{Z}$$

phiFunction.h

Description: *Euler's* ϕ function is defined as $\phi(n) := \#$ of positive integers $\leq n$ that are coprime with n. $\phi(1) = 1$, p prime $\Rightarrow \phi(p^k) = (p-1)p^{k-1}$, m, n coprime $\Rightarrow \phi(mn) = \phi(m)\phi(n)$. If $n = p_1^{k_1} p_2^{k_2} ... p_r^{k_r}$ then $\phi(n) = (p_1-1)p_1^{k_1-1} ... (p_r-1)p_r^{k_r-1}$. $\phi(n) = n \cdot \prod_{p|n}(1 - 1/p)$.
$\sum_{d|n} \phi(d) = n$, $\sum_{1 \leq k \leq n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$
Euler's thm: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod{n}$.
Fermat's little thm: p prime $\Rightarrow a^{p-1} \equiv 1 \pmod{p}$ $\forall a$.

cf7d6d, 8 lines

```
const int LIM = 5000000;
int phi[LIM];

void calculatePhi() {
  rep(i,0,LIM) phi[i] = i&1 ? i : i
     /2;
  for (int i = 3; i < LIM; i += 2) if
     (phi[i] == i)
    for (int j = i; j < LIM; j += i)
      phi[j] -= phi[j] / i;
}
```

5.4 Pythagorean Triples

The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \;\; b = k \cdot (2mn), \;\; c = k \cdot (m^2 + n^2),$$

with $m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

5.5 Primes

$p = 962592769$ is such that $2^{21} \mid p - 1$, which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.

Primitive roots exist modulo any prime power p^a, except for $p = 2, a > 2$, and there are $\phi(\phi(p^a))$ many. For $p = 2, a > 2$, the group $\mathbb{Z}_{2^a}^\times$ is instead isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$.

5.6 Estimates

$\sum_{d|n} d = O(n \log \log n)$.

The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

5.7 Mobius Function

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

Mobius Inversion:

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

Other useful formulas/forms:

$\sum_{d|n} \mu(d) = [n = 1]$ (very useful)

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

$g(n) = \sum_{1 \leq m \leq n} f(\lfloor \frac{n}{m} \rfloor) \Leftrightarrow f(n) = \sum_{1 \leq m \leq n} \mu(m) g(\lfloor \frac{n}{m} \rfloor)$

Graph (6)

6.1 Network flow

PushRelabel.h

Description: Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only.
Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$

0ae1d4, 48 lines

```
struct PushRelabel {
  struct Edge {
    int dest, back;
    ll f, c;
  };
  vector<vector<Edge>> g;
  vector<ll> ec;
  vector<Edge*> cur;
  vector<vi> hs; vi H;
  PushRelabel(int n) : g(n), ec(n),
      cur(n), hs(2*n), H(n) {}

  void addEdge(int s, int t, ll cap,
      ll rcap=0) {
    if (s == t) return;
    g[s].push_back({t, sz(g[t]), 0,
        cap});
    g[t].push_back({s, sz(g[s])-1, 0,
        rcap});
  }

  void addFlow(Edge& e, ll f) {
    Edge &back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest
        ]].push_back(e.dest);
    e.f += f; e.c -= f; ec[e.dest] +=
        f;
    back.f -= f; back.c += f; ec[back
        .dest] -= f;
  }
  ll calc(int s, int t) {
    int v = sz(g); H[s] = v; ec[t] =
        1;
    vi co(2*v); co[0] = v-1;
    rep(i,0,v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e
        .c);

    for (int hi = 0;;) {
      while (hs[hi].empty()) if (!hi
          --) return -ec[s];
      int u = hs[hi].back(); hs[hi].
          pop_back();
      while (ec[u] > 0)  // discharge
          u
        if (cur[u] == g[u].data() +
            sz(g[u])) {
          H[u] = 1e9;
          for (Edge& e : g[u]) if (e.
              c && H[u] > H[e.dest]+1)
            H[u] = H[e.dest]+1, cur[u
                ] = &e;
          if (++co[H[u]], !--co[hi]
              && hi < v)
            rep(i,0,v) if (hi < H[i]
                && H[i] < v)
              --co[H[i]], H[i] = v +
                  1;
          hi = H[u];
        } else if (cur[u]->c && H[u]
            == H[cur[u]->dest]+1)
          addFlow(*cur[u], min(ec[u],
              cur[u]->c));
        else ++cur[u];
    }
  }
  bool leftOfMinCut(int a) { return H
      [a] >= sz(g); }
};
```

MinCostMaxFlow.h

Description: Min-cost max-flow. cap[i][j] != cap[j][i] is allowed; double edges are not. If costs can be negative, call setpi before maxflow, but note that negative cost cycles are not supported. To obtain the actual flow, look at positive values only.
Time: Approximately $\mathcal{O}\left(E^2\right)$

fe85cc, 81 lines

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

const ll INF = numeric_limits<ll>::
    max() / 4;
typedef vector<ll> VL;

struct MCMF {
  int N;
  vector<vi> ed, red;
  vector<VL> cap, flow, cost;
  vi seen;
  VL dist, pi;
  vector<pii> par;

  MCMF(int N) :
    N(N), ed(N), red(N), cap(N, VL(N)
        ), flow(cap), cost(cap),
    seen(N), dist(N), pi(N), par(N) {
        }

  void addEdge(int from, int to, ll
      cap, ll cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
    ed[from].push_back(to);
    red[to].push_back(from);
  }
```

```
  void path(int s) {
    fill(all(seen), 0);
    fill(all(dist), INF);
    dist[s] = 0; ll di;

    __gnu_pbds::priority_queue<pair<
       ll, int>> q;
    vector<decltype(q)::
       point_iterator> its(N);
    q.push({0, s});

    auto relax = [&](int i, ll cap,
       ll cost, int dir) {
      ll val = di - pi[i] + cost;
      if (cap && val < dist[i]) {
        dist[i] = val;
        par[i] = {s, dir};
        if (its[i] == q.end()) its[i]
           = q.push({-dist[i], i});
        else q.modify(its[i], {-dist[
           i], i});
      }
    };

    while (!q.empty()) {
      s = q.top().second; q.pop();
      seen[s] = 1; di = dist[s] + pi[
         s];
      for (int i : ed[s]) if (!seen[i
         ])
        relax(i, cap[s][i] - flow[s][
           i], cost[s][i], 1);
      for (int i : red[s]) if (!seen[
         i])
        relax(i, flow[i][s], -cost[i
           ][s], 0);
    }
    rep(i,0,N) pi[i] = min(pi[i] +
       dist[i], INF);
  }

  pair<ll, ll> maxflow(int s, int t)
     {
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
      ll fl = INF;
      for (int p,r,x = t; tie(p,r) =
         par[x], x != s; x = p)
        fl = min(fl, r ? cap[p][x] -
           flow[p][x] : flow[x][p]);
      totflow += fl;
      for (int p,r,x = t; tie(p,r) =
         par[x], x != s; x = p)
        if (r) flow[p][x] += fl;
        else flow[x][p] -= fl;
    }
    rep(i,0,N) rep(j,0,N) totcost +=
       cost[i][j] * flow[i][j];
    return {totflow, totcost};
  }

  // If some costs can be negative,
     call this before maxflow:
  void setpi(int s) { // (otherwise,
     leave this out)
    fill(all(pi), INF); pi[s] = 0;
    int it = N, ch = 1; ll v;
    while (ch-- && it--)
      rep(i,0,N) if (pi[i] != INF)
        for (int to : ed[i]) if (cap[
           i][to])
          if ((v = pi[i] + cost[i][to
             ]) < pi[to])
            pi[to] = v, ch = 1;
    assert(it >= 0); // negative cost
       cycle
  }
};
```

EdmondsKarp.h

Description: Flow algorithm with guaranteed complexity $O(VE^2)$. To get edge flow values, compare capacities before and after, and take the positive values only.

482fe0, 35 lines

```
template<class T> T edmondsKarp(
   vector<unordered_map<int, T>>&
   graph, int source, int sink) {
  assert(source != sink);
  T flow = 0;
  vi par(sz(graph)), q = par;

  for (;;) {
    fill(all(par), -1);
    par[source] = 0;
    int ptr = 1;
    q[0] = source;

    rep(i,0,ptr) {
      int x = q[i];
      for (auto e : graph[x]) {
        if (par[e.first] == -1 && e.
           second > 0) {
          par[e.first] = x;
          q[ptr++] = e.first;
          if (e.first == sink) goto
             out;
        }
      }
    }
    return flow;
out:
```

```
    T inc = numeric_limits<T>::max();
    for (int y = sink; y != source; y
        = par[y])
      inc = min(inc, graph[par[y]][y
          ]);

    flow += inc;
    for (int y = sink; y != source; y
        = par[y]) {
      int p = par[y];
      if ((graph[p][y] -= inc) <= 0)
          graph[p].erase(y);
      graph[y][p] += inc;
    }
  }
}
```

MinCut.h

Description: After running max-flow, the left side of a min-cut from s to t is given by all vertices reachable from s, only traversing edges with positive residual capacity.

GlobalMinCut.h

Description: Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.
Time: $\mathcal{O}\left(V^3\right)$

8b0e19, 21 lines

```
pair<int, vi> globalMinCut(vector<vi>
    mat) {
  pair<int, vi> best = {INT_MAX, {}};
  int n = sz(mat);
  vector<vi> co(n);
  rep(i,0,n) co[i] = {i};
  rep(ph,1,n) {
    vi w = mat[0];
    size_t s = 0, t = 0;
    rep(it,0,n-ph) { // O(V^2) -> O(E
        log V) with prio. queue
      w[t] = INT_MIN;
      s = t, t = max_element(all(w))
          - w.begin();
      rep(i,0,n) w[i] += mat[t][i];
    }
    best = min(best, {w[t] - mat[t][t
        ], co[t]});
    co[s].insert(co[s].end(), all(co[
        t]));
    rep(i,0,n) mat[s][i] += mat[t][i
        ];
    rep(i,0,n) mat[i][s] = mat[s][i];
    mat[0][t] = INT_MIN;
  }
  return best;
}
```

6.2 Matching

hopcroftKarp.h

Description: Fast bipartite matching algorithm. Graph g should be a list of neighbors of the left partition, and $btoa$ should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. $btoa[i]$ will be the match for vertex i on the right side, or -1 if it's not matched.
Usage: `vi btoa(m, -1); hopcroftKarp(g, btoa);`
Time: $\mathcal{O}\left(\sqrt{V}E\right)$

f612e4, 42 lines

```
bool dfs(int a, int L, vector<vi>& g,
    vi& btoa, vi& A, vi& B) {
  if (A[a] != L) return 0;
  A[a] = -1;
  for (int b : g[a]) if (B[b] == L +
      1) {
    B[b] = 0;
    if (btoa[b] == -1 || dfs(btoa[b],
        L + 1, g, btoa, A, B))
      return btoa[b] = a, 1;
  }
  return 0;
}

int hopcroftKarp(vector<vi>& g, vi&
    btoa) {
  int res = 0;
  vi A(g.size()), B(btoa.size()), cur
      , next;
  for (;;) {
    fill(all(A), 0);
    fill(all(B), 0);
    cur.clear();
    for (int a : btoa) if(a != -1) A[
        a] = -1;
    rep(a,0,sz(g)) if(A[a] == 0) cur.
        push_back(a);
    for (int lay = 1;; lay++) {
      bool islast = 0;
      next.clear();
      for (int a : cur) for (int b :
          g[a]) {
        if (btoa[b] == -1) {
          B[b] = lay;
          islast = 1;
        }
        else if (btoa[b] != a && !B[b
            ]) {
          B[b] = lay;
          next.push_back(btoa[b]);
        }
```

```
      }
      if (islast) break;
      if (next.empty()) return res;
      for (int a : next) A[a] = lay;
      cur.swap(next);
    }
    rep(a,0,sz(g))
      res += dfs(a, 0, g, btoa, A, B)
          ;
  }
}
```

DFSMatching.h

Description: Simple bipartite matching algorithm. Graph g should be a list of neighbors of the left partition, and *btoa* should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. *btoa*[*i*] will be the match for vertex i on the right side, or -1 if it's not matched.
Usage: `vi btoa(m, -1); dfsMatching(g, btoa);`
Time: $\mathcal{O}(VE)$

522b98, 22 lines

```
bool find(int j, vector<vi>& g, vi&
    btoa, vi& vis) {
  if (btoa[j] == -1) return 1;
  vis[j] = 1; int di = btoa[j];
  for (int e : g[di])
    if (!vis[e] && find(e, g, btoa,
        vis)) {
      btoa[e] = di;
      return 1;
    }
  return 0;
}
int dfsMatching(vector<vi>& g, vi&
    btoa) {
  vi vis;
  rep(i,0,sz(g)) {
    vis.assign(sz(btoa), 0);
    for (int j : g[i])
      if (find(j, g, btoa, vis)) {
        btoa[j] = i;
        break;
      }
  }
  return sz(btoa) - (int)count(all(
      btoa), -1);
}
```

MinimumVertexCover.h

Description: Finds a minimum vertex cover in a bipartite graph. The size is the same as the size of a maximum matching, and the complement is a maximum independent set.

"DFSMatching.h" da4196, 20 lines

```
vi cover(vector<vi>& g, int n, int m)
    {
  vi match(m, -1);
  int res = dfsMatching(g, match);
  vector<bool> lfound(n, true), seen(
      m);
  for (int it : match) if (it != -1)
      lfound[it] = false;
  vi q, cover;
  rep(i,0,n) if (lfound[i]) q.
      push_back(i);
  while (!q.empty()) {
    int i = q.back(); q.pop_back();
    lfound[i] = 1;
    for (int e : g[i]) if (!seen[e]
        && match[e] != -1) {
      seen[e] = true;
      q.push_back(match[e]);
    }
  }
  rep(i,0,n) if (!lfound[i]) cover.
      push_back(i);
  rep(i,0,m) if (seen[i]) cover.
      push_back(n+i);
  assert(sz(cover) == res);
  return cover;
}
```

WeightedMatching.h

Description: Given a weighted bipartite graph, matches every node on the left with a node on the right such that no nodes are in two matchings and the sum of the edge weights is minimal. Takes cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and returns (min cost, match), where L[i] is matched with R[match[i]]. Negate costs for max cost. Requires $N \leq M$.
Time: $\mathcal{O}\left(N^2M\right)$

1e0fe9, 31 lines

```
pair<int, vi> hungarian(const vector<
    vi> &a) {
  if (a.empty()) return {0, {}};
  int n = sz(a) + 1, m = sz(a[0]) +
      1;
  vi u(n), v(m), p(m), ans(n - 1);
  rep(i,1,n) {
    p[0] = i;
    int j0 = 0; // add "dummy" worker
        0
    vi dist(m, INT_MAX), pre(m, -1);
    vector<bool> done(m + 1);
    do { // dijkstra
      done[j0] = true;
```

```
      int i0 = p[j0], j1, delta =
         INT_MAX;
      rep(j,1,m) if (!done[j]) {
        auto cur = a[i0 - 1][j - 1] -
           u[i0] - v[j];
        if (cur < dist[j]) dist[j] =
            cur, pre[j] = j0;
        if (dist[j] < delta) delta =
            dist[j], j1 = j;
      }
      rep(j,0,m) {
        if (done[j]) u[p[j]] += delta
            , v[j] -= delta;
        else dist[j] -= delta;
      }
      j0 = j1;
    } while (p[j0]);
    while (j0) { // update
       alternating path
      int j1 = pre[j0];
      p[j0] = p[j1], j0 = j1;
    }
  }
  rep(j,1,m) if (p[j]) ans[p[j] - 1]
     = j - 1;
  return {-v[0], ans}; // min cost
}
```

GeneralMatching.h

Description: Matching for general graphs. Fails with probability N/mod.
Time: $\mathcal{O}\left(N^3\right)$

"../numerical/MatrixInverse-mod.h" cb1912, 40 lines

```
vector<pii> generalMatching(int N,
   vector<pii>& ed) {
  vector<vector<ll>> mat(N, vector<ll
     >(N)), A;
  for (pii pa : ed) {
    int a = pa.first, b = pa.second,
       r = rand() % mod;
    mat[a][b] = r, mat[b][a] = (mod -
        r) % mod;
  }

  int r = matInv(A = mat), M = 2*N -
     r, fi, fj;
  assert(r % 2 == 0);

  if (M != N) do {
    mat.resize(M, vector<ll>(M));
    rep(i,0,N) {
      mat[i].resize(M);
      rep(j,N,M) {
        int r = rand() % mod;
        mat[i][j] = r, mat[j][i] = (
           mod - r) % mod;
      }
    }
  } while (matInv(A = mat) != M);

  vi has(M, 1); vector<pii> ret;
  rep(it,0,M/2) {
    rep(i,0,M) if (has[i])
      rep(j,i+1,M) if (A[i][j] && mat
         [i][j]) {
        fi = i; fj = j; goto done;
    } assert(0); done:
    if (fj < N) ret.emplace_back(fi,
       fj);
    has[fi] = has[fj] = 0;
    rep(sw,0,2) {
      ll a = modpow(A[fi][fj], mod-2)
         ;
      rep(i,0,M) if (has[i] && A[i][
         fj]) {
        ll b = A[i][fj] * a % mod;
        rep(j,0,M) A[i][j] = (A[i][j]
            - A[fi][j] * b) % mod;
      }
      swap(fi,fj);
    }
  }
  return ret;
}
```

6.3 DFS algorithms

SCC.h

Description: Finds strongly connected components in a directed graph. If vertices u, v belong to the same component, we can reach u from v and vice versa.
Usage: `scc(graph, [&](vi& v) { ... })` visits all components in reverse topological order. `comp[i]` holds the component index of a node (a component only has edges to components with lower index). `ncomps` will contain the number of components.
Time: $\mathcal{O}(E+V)$

76b5c9, 24 lines

```
vi val, comp, z, cont;
int Time, ncomps;
template<class G, class F> int dfs(
   int j, G& g, F& f) {
```

```
  int low = val[j] = ++Time, x; z.
      push_back(j);
  for (auto e : g[j]) if (comp[e] <
      0)
    low = min(low, val[e] ?: dfs(e,g,
        f));

  if (low == val[j]) {
    do {
      x = z.back(); z.pop_back();
      comp[x] = ncomps;
      cont.push_back(x);
    } while (x != j);
    f(cont); cont.clear();
    ncomps++;
  }
  return val[j] = low;
}
template<class G, class F> void scc(G
    & g, F f) {
  int n = sz(g);
  val.assign(n, 0); comp.assign(n,
      -1);
  Time = ncomps = 0;
  rep(i,0,n) if (comp[i] < 0) dfs(i,
      g, f);
}
```

BiconnectedComponents.h

Description: Finds all biconnected components in an undirected graph, and runs a callback for the edges in each. In a biconnected component there are at least two distinct paths between any two nodes. Note that a node can be in several components. An edge which is not in a component is a bridge, i.e., not part of any cycle.

Usage: `int eid = 0; ed.resize(N);`
`for each edge (a,b) {`
`ed[a].emplace_back(b, eid);`
`ed[b].emplace_back(a, eid++); }`
`bicomps([&](const vi& edgelist)`
`{...});`
Time: $\mathcal{O}(E+V)$

2965e5, 33 lines

```
vi num, st;
vector<vector<pii>> ed;
int Time;
template<class F>
int dfs(int at, int par, F& f) {
  int me = num[at] = ++Time, e, y,
      top = me;
  for (auto pa : ed[at]) if (pa.
      second != par) {
    tie(y, e) = pa;
    if (num[y]) {
      top = min(top, num[y]);
      if (num[y] < me)
        st.push_back(e);
    } else {
      int si = sz(st);
      int up = dfs(y, e, f);
      top = min(top, up);
      if (up == me) {
        st.push_back(e);
        f(vi(st.begin() + si, st.end
            ()));
        st.resize(si);
      }
      else if (up < me) st.push_back(
          e);
      else { /* e is a bridge */ }
    }
  }
  return top;
}

template<class F>
void bicomps(F f) {
  num.assign(sz(ed), 0);
  rep(i,0,sz(ed)) if (!num[i]) dfs(i,
      -1, f);
}
```

2sat.h

Description: Calculates a valid assignment to boolean variables a, b, c,... to a 2-SAT problem, so that an expression of the type $(a\|\|b)\&\&(!a\|\|c)\&\&(d\|\|!b)\&\&...$ becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions (~x).

Usage: `TwoSat ts(number of boolean variables);`
`ts.either(0, ~3); // Var 0 is true or var 3 is false`
`ts.setValue(2); // Var 2 is true`
`ts.atMostOne({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true`
`ts.solve(); // Returns true iff it is solvable`
`ts.values[0..N-1] holds the assigned values to the vars`
Time: $\mathcal{O}(N+E)$, where N is the number of boolean variables, and E is the number of clauses.

5f9706, 56 lines

```
struct TwoSat {
  int N;
  vector<vi> gr;
  vi values; // 0 = false, 1 = true

  TwoSat(int n = 0) : N(n), gr(2*n) {
      }
```

```
  int addVar() { // (optional)
    gr.emplace_back();
    gr.emplace_back();
    return N++;
  }

  void either(int f, int j) {
    f = max(2*f, -1-2*f);
    j = max(2*j, -1-2*j);
    gr[f].push_back(j^1);
    gr[j].push_back(f^1);
  }
  void setValue(int x) { either(x, x)
     ; }

  void atMostOne(const vi& li) { // (
      optional)
    if (sz(li) <= 1) return;
    int cur = ~li[0];
    rep(i,2,sz(li)) {
      int next = addVar();
      either(cur, ~li[i]);
      either(cur, next);
      either(~li[i], next);
      cur = ~next;
    }
    either(cur, ~li[1]);
  }

  vi val, comp, z; int time = 0;
  int dfs(int i) {
    int low = val[i] = ++time, x; z.
        push_back(i);
    for(int e : gr[i]) if (!comp[e])
      low = min(low, val[e] ?: dfs(e)
          );
    if (low == val[i]) do {
      x = z.back(); z.pop_back();
      comp[x] = low;
      if (values[x>>1] == -1)
        values[x>>1] = x&1;
    } while (x != i);
    return val[i] = low;
  }

  bool solve() {
    values.assign(N, -1);
    val.assign(2*N, 0); comp = val;
    rep(i,0,2*N) if (!comp[i]) dfs(i)
        ;
    rep(i,0,N) if (comp[2*i] == comp
        [2*i+1]) return 0;
    return 1;
  }
};
```

EulerWalk.h

Description: Eulerian undirected/directed path/cycle algorithm. Input should be a vector of (dest, global edge index), where for undirected graphs, forward/backward edges have the same index. Returns a list of nodes in the Eulerian path/cycle with src at both start and end, or empty list if no cycle/path exists. To get edge indices back, add .second to s and ret.

Time: $\mathcal{O}(V+E)$

780b64, 15 lines

```
vi eulerWalk(vector<vector<pii>>& gr,
    int nedges, int src=0) {
  int n = sz(gr);
  vi D(n), its(n), eu(nedges), ret, s
      = {src};
  D[src]++; // to allow Euler paths,
      not just cycles
  while (!s.empty()) {
    int x = s.back(), y, e, &it = its
        [x], end = sz(gr[x]);
    if (it == end){ ret.push_back(x);
         s.pop_back(); continue; }
    tie(y, e) = gr[x][it++];
    if (!eu[e]) {
      D[x]--, D[y]++;
      eu[e] = 1; s.push_back(y);
    }}
  for (int x : D) if (x < 0 || sz(ret
      ) != nedges+1) return {};
  return {ret.rbegin(), ret.rend()};
}
```

6.4 Coloring

EdgeColoring.h

Description: Given a simple, undirected graph with max degree D, computes a $(D+1)$-coloring of the edges such that no neighboring edges share a color. (D-coloring is NP-hard, but can be done for bipartite graphs by repeated matchings of max-degree nodes.)

Time: $\mathcal{O}(NM)$

e210e2, 31 lines

```
vi edgeColoring(int N, vector<pii>
    eds) {
  vi cc(N + 1), ret(sz(eds)), fan(N),
      free(N), loc;
  for (pii e : eds) ++cc[e.first], ++
      cc[e.second];
  int u, v, ncols = *max_element(all(
      cc)) + 1;
  vector<vi> adj(N, vi(ncols, -1));
  for (pii e : eds) {
    tie(u, v) = e;
    fan[0] = v;
    loc.assign(ncols, 0);
```

```
    int at = u, end = u, d, c = free[
        u], ind = 0, i = 0;
    while (d = free[v], !loc[d] && (v
         = adj[u][d]) != -1)
      loc[d] = ++ind, cc[ind] = d,
          fan[ind] = v;
    cc[loc[d]] = c;
    for (int cd = d; at != -1; cd ^=
        c ^ d, at = adj[at][cd])
      swap(adj[at][cd], adj[end = at
          ][cd ^ c ^ d]);
    while (adj[fan[i]][d] != -1) {
      int left = fan[i], right = fan
          [++i], e = cc[i];
      adj[u][e] = left;
      adj[left][e] = u;
      adj[right][e] = -1;
      free[right] = e;
    }
    adj[u][d] = fan[i];
    adj[fan[i]][d] = u;
    for (int y : {fan[0], u, end})
      for (int& z = free[y] = 0; adj[
          y][z] != -1; z++);
  }
  rep(i,0,sz(eds))
    for (tie(u, v) = eds[i]; adj[u][
        ret[i]] != v;) ++ret[i];
  return ret;
}
```

6.5 Trees

BinaryLifting.h

Description: Calculate power of two jumps in a tree, to support fast upward jumps and LCAs. Assumes the root node points to itself.

Time: construction $\mathcal{O}(N \log N)$, queries $\mathcal{O}(\log N)$

bfce85, 25 lines

```
vector<vi> treeJump(vi& P){
  int on = 1, d = 1;
  while(on < sz(P)) on *= 2, d++;
  vector<vi> jmp(d, P);
  rep(i,1,d) rep(j,0,sz(P))
    jmp[i][j] = jmp[i-1][jmp[i-1][j
        ]];
  return jmp;
}

int jmp(vector<vi>& tbl, int nod, int
     steps){
  rep(i,0,sz(tbl))
    if(steps&(1<<i)) nod = tbl[i][nod
        ];
  return nod;
}

int lca(vector<vi>& tbl, vi& depth,
    int a, int b) {
  if (depth[a] < depth[b]) swap(a, b)
      ;
  a = jmp(tbl, a, depth[a] - depth[b
      ]);
  if (a == b) return a;
  for (int i = sz(tbl); i--;) {
    int c = tbl[i][a], d = tbl[i][b];
    if (c != d) a = c, b = d;
  }
  return tbl[0][a];
}
```

LCA.h

Description: Data structure for computing lowest common ancestors in a tree (with 0 as root). C should be an adjacency list of the tree, either directed or undirected.

Time: $\mathcal{O}(N \log N + Q)$

"../data-structures/RMQ.h" 0f62fb, 21 lines

```
struct LCA {
  int T = 0;
  vi time, path, ret;
  RMQ<int> rmq;

  LCA(vector<vi>& C) : time(sz(C)),
      rmq((dfs(C,0,-1), ret)) {}
  void dfs(vector<vi>& C, int v, int
      par) {
    time[v] = T++;
    for (int y : C[v]) if (y != par)
        {
      path.push_back(v), ret.
          push_back(time[v]);
      dfs(C, y, v);
    }
  }

  int lca(int a, int b) {
    if (a == b) return a;
    tie(a, b) = minmax(time[a], time[
        b]);
    return path[rmq.query(a, b)];
  }
  //dist(a,b){return depth[a] + depth
      [b] - 2*depth[lca(a,b)];}
};
```

CompressTree.h

Description: Given a rooted tree and a subset S of nodes, compute the minimal subtree that contains all the nodes by adding all (at most $|S|-1$) pairwise LCA's and compressing edges. Returns a list of (par, orig_index) representing a tree rooted at 0. The root points to itself.
Time: $\mathcal{O}(|S|\log|S|)$

"LCA.h" 9775a0, 21 lines

```
typedef vector<pair<int, int>> vpi;
vpi compressTree(LCA& lca, const vi&
    subset) {
  static vi rev; rev.resize(sz(lca.
      time));
  vi li = subset, &T = lca.time;
  auto cmp = [&](int a, int b) {
      return T[a] < T[b]; };
  sort(all(li), cmp);
  int m = sz(li)-1;
  rep(i,0,m) {
    int a = li[i], b = li[i+1];
    li.push_back(lca.lca(a, b));
  }
  sort(all(li), cmp);
  li.erase(unique(all(li)), li.end())
      ;
  rep(i,0,sz(li)) rev[li[i]] = i;
  vpi ret = {pii(0, li[0])};
  rep(i,0,sz(li)-1) {
    int a = li[i], b = li[i+1];
    ret.emplace_back(rev[lca.lca(a, b
        )], b);
  }
  return ret;
}
```

HLD.h

Description: Decomposes a tree into vertex disjoint heavy paths and light edges such that the path from any leaf to the root contains at most log(n) light edges. Code does additive modifications and max queries, but can support commutative segtree modifications/queries on paths and subtrees. Takes as input the full adjacency list. VALS_EDGES being true means that values are stored in the edges, as opposed to the nodes. All values initialized to the segtree default. Root must be 0.
Time: $\mathcal{O}\left((\log N)^2\right)$

"../data-structures/LazySegmentTree.h" 6f34db, 46 lines

```
template <bool VALS_EDGES> struct HLD
    {
  int N, tim = 0;
  vector<vi> adj;
  vi par, siz, depth, rt, pos;
  Node *tree;
  HLD(vector<vi> adj_)
    : N(sz(adj_)), adj(adj_), par(N,
      -1), siz(N, 1), depth(N),
      rt(N),pos(N),tree(new Node(0, N
        )){ dfsSz(0); dfsHld(0); }
  void dfsSz(int v) {
    if (par[v] != -1) adj[v].erase(
        find(all(adj[v]), par[v]));
    for (int& u : adj[v]) {
      par[u] = v, depth[u] = depth[v]
          + 1;
      dfsSz(u);
      siz[v] += siz[u];
      if (siz[u] > siz[adj[v][0]])
        swap(u, adj[v][0]);
    }
  }
  void dfsHld(int v) {
    pos[v] = tim++;
    for (int u : adj[v]) {
      rt[u] = (u == adj[v][0] ? rt[v]
          : u);
      dfsHld(u);
    }
  }
  template <class B> void process(int
      u, int v, B op) {
    for (; rt[u] != rt[v]; v = par[rt
        [v]]) {
      if (depth[rt[u]] > depth[rt[v
          ]]) swap(u, v);
      op(pos[rt[v]], pos[v] + 1);
    }
    if (depth[u] > depth[v]) swap(u,
        v);
    op(pos[u] + VALS_EDGES, pos[v] +
        1);
  }
  void modifyPath(int u, int v, int
      val) {
    process(u, v, [&](int l, int r) {
        tree->add(l, r, val); });
  }
  int queryPath(int u, int v) { //
      Modify depending on problem
    int res = -1e9;
    process(u, v, [&](int l, int r) {
        res = max(res, tree->query(l,
            r));
    });
    return res;
  }
  int querySubtree(int v) { //
      modifySubtree is similar
```

```
    return tree->query(pos[v] +
       VALS_EDGES, pos[v] + siz[v]);
  }
};
```

LinkCutTree.h

Description: Represents a forest of unrooted trees. You can add and remove edges (as long as the result is still a forest), and check whether two nodes are in the same tree.

Time: All operations take amortized $\mathcal{O}(\log N)$.

5909e2, 90 lines

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

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

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

```
      pp->c[1] = u; pp->fix(); u = pp
          ;
    }
    return u;
  }
};
```

sack.h

Description: Offline trees
Time: $\mathcal{O}(nlogn)$ Amortized

9f4f90, 70 lines

```
int color[MAXN];
vector<int> adj[MAXN];
int tin[MAXN], tout[MAXN], ti[MAXN];

int cnt[MAXN];
long long sum[MAXN]; // sum of
    elements with frequency
long long ans[MAXN];

int sz[MAXN];

int mx = 0;
int timer = 0;

void dfs(int u, int p)
{
    sz[u] = 1;
    ti[timer] = u;
    tin[u] = timer++;
    for (const auto &it : adj[u])
        if (it != p)
            dfs(it, u), sz[u] += sz[
                it];
    tout[u] = timer;
}

void add(int x)
{
    sum[cnt[color[x]]] -= color[x];
    if (cnt[color[x]] == mx)
        ++mx;
    ++cnt[color[x]];
    sum[cnt[color[x]]] += color[x];
}

void remove(int x)
{
    sum[cnt[color[x]]] -= color[x];
    if (cnt[color[x]] == mx && sum[
        cnt[color[x]]] == 0)
        --mx;
    --cnt[color[x]];
    sum[cnt[color[x]]] += color[x];
}

void dfsMerge(int u, int p, int keep)
{
    int Max = -1, bigChild = -1;
    for (const auto &it : adj[u])
        if (it != p && sz[it] > Max)
            Max = sz[it], bigChild =
                it;

    for (const auto &it : adj[u])
        if (it != p && it != bigChild
            )
            dfsMerge(it, u, 0);

    if (bigChild != -1)
    {
        dfsMerge(bigChild, u, 1);
    }
    add(u);
    for (const auto &it : adj[u])
        if (it != p && it != bigChild
            )
            for (int i = tin[it]; i <
                tout[it]; i++)
                add(ti[i]);

    ans[u] = sum[mx];
    if (!keep)
    {
        for (int i = tin[u]; i < tout
            [u]; i++)
            remove(ti[i]);
    }
}
```

Geometry (7)

7.1 Geometric primitives

Point.h

Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

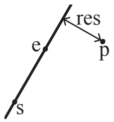
47ec0a, 28 lines

```
template <class T> int sgn(T x) {
    return (x > 0) - (x < 0); }
template<class T>
struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x=0, T y=0) : x(x)
      , y(y) {}
  bool operator<(P p) const { return
      tie(x,y) < tie(p.x,p.y); }
```

```
bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }
P operator+(P p) const { return P(x+p.x, y+p.y); }
P operator-(P p) const { return P(x-p.x, y-p.y); }
P operator*(T d) const { return P(x*d, y*d); }
P operator/(T d) const { return P(x/d, y/d); }
T dot(P p) const { return x*p.x + y*p.y; }
T cross(P p) const { return x*p.y - y*p.x; }
T cross(P a, P b) const { return (a-*this).cross(b-*this); }
T dist2() const { return x*x + y*y; }
double dist() const { return sqrt((double)dist2()); }
// angle to x-axis in interval [-pi, pi]
double angle() const { return atan2(y, x); }
P unit() const { return *this/dist(); } // makes dist()=1
P perp() const { return P(-y, x); } // rotates +90 degrees
P normal() const { return perp().unit(); }
// returns point rotated 'a' radians ccw around the origin
P rotate(double a) const {
  return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
friend ostream& operator<<(ostream& os, P p) {
  return os << "(" << p.x << "," << p.y << ")"; }
};
```

lineDistance.h

Description:
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long. Using Point3D will always give a non-negative distance. For Point3D, call .dist on the result of the cross product.

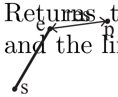


"Point.h" f6bf6b, 4 lines

```
template<class P>
double lineDist(const P& a, const P& b, const P& p) {
  return (double)(b-a).cross(p-a)/(b-a).dist();
}
```

SegmentDistance.h

Description:
Returns the shortest distance between point p and the line segment from point s to e.



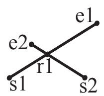
Usage: Point<double> a, b(2,2), p(1,1);
bool onSegment = segDist(a,b,p) < 1e-10;

"Point.h" 5c88f4, 6 lines

```
typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
  if (s==e) return (p-s).dist();
  auto d = (e-s).dist2(), t = min(d,max(.0,(p-s).dot(e-s)));
  return ((p-s)*d-(e-s)*t).dist()/d;
}
```

SegmentIntersection.h

Description:
If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.



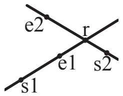
Usage: vector<P> inter = segInter(s1,e1,s2,e2);
if (sz(inter)==1)
cout << "segments intersect at " << inter[0] << endl;

"Point.h", "OnSegment.h" 9d57f2, 13 lines

```
template<class P> vector<P> segInter(
    P a, P b, P c, P d) {
  auto oa = c.cross(d, a), ob = c.
      cross(d, b),
       oc = a.cross(b, c), od = a.
           cross(b, d);
  // Checks if intersection is single
      non-endpoint point.
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc
      ) * sgn(od) < 0)
    return {(a * ob - b * oa) / (ob -
        oa)};
  set<P> s;
  if (onSegment(c, d, a)) s.insert(a)
      ;
  if (onSegment(c, d, b)) s.insert(b)
      ;
  if (onSegment(a, b, c)) s.insert(c)
      ;
  if (onSegment(a, b, d)) s.insert(d)
      ;
  return {all(s)};
}
```

lineIntersection.h

Description:
If a unique intersection point of the lines going through s1,e1 and s2,e2 exists {1, point} is returned. If no intersection point exists {0, (0,0)} is returned and if infinitely many exists {-1, (0,0)} is returned. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or ll.



Usage: `auto res = lineInter(s1,e1,s2,e2);`
`if (res.first == 1)`
`cout << "intersection point at " << res.second << endl;`

"Point.h" a01f81, 8 lines

```
template<class P>
pair<int, P> lineInter(P s1, P e1, P
    s2, P e2) {
  auto d = (e1 - s1).cross(e2 - s2);
  if (d == 0) // if parallel
    return {-(s1.cross(e1, s2) == 0),
         P(0, 0)};
  auto p = s2.cross(e1, e2), q = s2.
      cross(e2, s1);
  return {1, (s1 * p + e1 * q) / d};
}
```

sideOf.h

Description: Returns where p is as seen from s towards e. 1/0/-1 ⇔ left/on line/right. If the optional argument *eps* is given 0 is returned if p is within distance *eps* from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.
Usage: `bool left = sideOf(p1,p2,q)==1;`

"Point.h" 3af81c, 9 lines

```
template<class P>
int sideOf(P s, P e, P p) { return
    sgn(s.cross(e, p)); }

template<class P>
int sideOf(const P& s, const P& e,
    const P& p, double eps) {
  auto a = (e-s).cross(p-s);
  double l = (e-s).dist()*eps;
  return (a > l) - (a < -l);
}
```

OnSegment.h

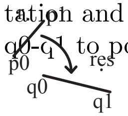
Description: Returns true iff p lies on the line segment from s to e. Use `(segDist(s,e,p)<=epsilon)` instead when using Point<double>.

"Point.h" c597e8, 3 lines

```
template<class P> bool onSegment(P s,
     P e, P p) {
  return p.cross(s, e) == 0 && (s - p
      ).dot(e - p) <= 0;
}
```

linearTransformation.h

Description:
Apply the linear transformation (translation, rotation and scaling) which takes line p0-p1 to line q0-q1 to point r.



"Point.h" 03a306, 6 lines

```
typedef Point<double> P;
P linearTransformation(const P& p0,
    const P& p1,
     const P& q0, const P& q1, const P
        & r) {
  P dp = p1-p0, dq = q1-q0, num(dp.
      cross(dq), dp.dot(dq));
  return q0 + P((r-p0).cross(num), (r
      -p0).dot(num))/dp.dist2();
}
```

Angle.h

Description: A class for ordering angles (as represented by int points and a number of rotations around the origin). Useful for rotational sweeping. Sometimes also represents points or vectors.

Usage: vector<Angle> v = {w[0], w[0].t360() ...}; // sorted
int j = 0; rep(i,0,n) { while (v[j] < v[i].t180()) ++j; }
// sweeps j such that (j-i) represents the number of positively oriented triangles with vertices at 0 and i

0f0602, 35 lines

```
struct Angle {
  int x, y;
  int t;
  Angle(int x, int y, int t=0) : x(x)
      , y(y), t(t) {}
  Angle operator-(Angle b) const {
      return {x-b.x, y-b.y, t}; }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0)
        ;
  }
  Angle t90() const { return {-y, x,
      t + (half() && x >= 0)}; }
  Angle t180() const { return {-x, -y
      , t + half()}; }
  Angle t360() const { return {x, y,
      t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to
      also compare distances
  return make_tuple(a.t, a.half(), a.
      y * (ll)b.x) <
          make_tuple(b.t, b.half(), a.
              x * (ll)b.y);
}

// Given two points, this calculates
    the smallest angle between
// them, i.e., the angle that covers
    the defined line segment.
pair<Angle, Angle> segmentAngles(
    Angle a, Angle b) {
  if (b < a) swap(a, b);
  return (b < a.t180() ?
           make_pair(a, b) : make_pair
               (b, a.t360()));
}
Angle operator+(Angle a, Angle b) {
    // point a + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) {
    // angle b - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y
      - a.y*b.x, tu - (b < a)};
}
```

7.2 Circles

CircleIntersection.h

Description: Computes the pair of points at which two circles intersect. Returns false in case of no intersection.

"Point.h" 84d6d3, 11 lines

```
typedef Point<double> P;
bool circleInter(P a,P b,double r1,
    double r2,pair<P, P>* out) {
  if (a == b) { assert(r1 != r2);
      return false; }
  P vec = b - a;
  double d2 = vec.dist2(), sum = r1+
      r2, dif = r1-r2,
           p = (d2 + r1*r1 - r2*r2)/(d2
               *2), h2 = r1*r1 - p*p*d2;
  if (sum*sum < d2 || dif*dif > d2)
      return false;
  P mid = a + vec*p, per = vec.perp()
      * sqrt(fmax(0, h2) / d2);
  *out = {mid + per, mid - per};
  return true;
}
```

CircleTangents.h

Description: Finds the external tangents of two circles, or internal if r2 is negated. Can return 0, 1, or 2 tangents – 0 if one circle contains the other (or overlaps it, in the internal case, or if the circles are the same); 1 if the circles are tangent to each other (in which case .first = .second and the tangent line is perpendicular to the line between the centers). .first and .second give the tangency points at circle 1 and 2 respectively. To find the tangents of a circle with a point set r2 to 0.

"Point.h" b0153d, 13 lines

```
template<class P>
vector<pair<P, P>> tangents(P c1,
    double r1, P c2, double r2) {
  P d = c2 - c1;
  double dr = r1 - r2, d2 = d.dist2()
      , h2 = d2 - dr * dr;
  if (d2 == 0 || h2 < 0)  return {};
  vector<pair<P, P>> out;
  for (double sign : {-1, 1}) {
    P v = (d * dr + d.perp() * sqrt(
        h2) * sign) / d2;
    out.push_back({c1 + v * r1, c2 +
        v * r2});
  }
  if (h2 == 0) out.pop_back();
  return out;
}
```

CirclePolygonIntersection.h

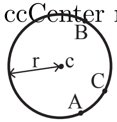
Description: Returns the area of the intersection of a circle with a ccw polygon.
Time: $\mathcal{O}(n)$

"../../content/geometry/Point.h" a1ee63, 19 lines

```
typedef Point<double> P;
#define arg(p, q) atan2(p.cross(q), p
    .dot(q))
double circlePoly(P c, double r,
    vector<P> ps) {
  auto tri = [&](P p, P q) {
    auto r2 = r * r / 2;
    P d = q - p;
    auto a = d.dot(p)/d.dist2(), b =
        (p.dist2()-r*r)/d.dist2();
    auto det = a * a - b;
    if (det <= 0) return arg(p, q) *
        r2;
    auto s = max(0., -a-sqrt(det)), t
        = min(1., -a+sqrt(det));
    if (t < 0 || 1 <= s) return arg(p
        , q) * r2;
    P u = p + d * s, v = p + d * t;
    return arg(p,u) * r2 + u.cross(v)
        /2 + arg(v,q) * r2;
  };
  auto sum = 0.0;
  rep(i,0,sz(ps))
    sum += tri(ps[i] - c, ps[(i + 1)
        % sz(ps)] - c);
  return sum;
}
```

circumcircle.h

Description:
The circumcirle of a triangle is the circle intersecting all three vertices. ccRadius returns the radius of the circle going through points A, B and C and ccCenter returns the center of the same circle.



"Point.h" 1caa3a, 9 lines

```
typedef Point<double> P;
double ccRadius(const P& A, const P&
    B, const P& C) {
  return (B-A).dist()*(C-B).dist()*(A
      -C).dist()/
       abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B,
    const P& C) {
  P b = C-A, c = B-A;
  return A + (b*c.dist2()-c*b.dist2()
      ).perp()/b.cross(c)/2;
}
```

MinimumEnclosingCircle.h

Description: Computes the minimum circle that encloses a set of points.
Time: expected $\mathcal{O}(n)$

"circumcircle.h" 09dd0a, 17 lines

```
pair<P, double> mec(vector<P> ps) {
  shuffle(all(ps), mt19937(time(0)));
  P o = ps[0];
  double r = 0, EPS = 1 + 1e-8;
  rep(i,0,sz(ps)) if ((o - ps[i]).
      dist() > r * EPS) {
    o = ps[i], r = 0;
    rep(j,0,i) if ((o - ps[j]).dist()
         > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      rep(k,0,j) if ((o - ps[k]).dist
          () > r * EPS) {
        o = ccCenter(ps[i], ps[j], ps
            [k]);
        r = (o - ps[i]).dist();
      }
    }
```

```
  }
  return {o, r};
}
```

7.3 Polygons

InsidePolygon.h

Description: Returns true if p lies within the polygon. If strict is true, it returns false for points on the boundary. The algorithm uses products in intermediate steps so watch out for overflow.
Usage: vector<P> v = {P{4,4}, P{1,2}, P{2,1}};
bool in = inPolygon(v, P{3, 3}, false);
Time: $\mathcal{O}(n)$

"Point.h", "OnSegment.h", "SegmentDistance.h" 2bf504, 11 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a,
    bool strict = true) {
  int cnt = 0, n = sz(p);
  rep(i,0,n) {
    P q = p[(i + 1) % n];
    if (onSegment(p[i], q, a)) return
        !strict;
    //or: if (segDist(p[i], q, a) <=
        eps) return !strict;
    cnt ^= ((a.y<p[i].y) - (a.y<q.y))
        * a.cross(p[i], q) > 0;
  }
  return cnt;
}
```

PolygonArea.h

Description: Returns twice the signed area of a polygon. Clockwise enumeration gives negative area. Watch out for overflow if using int as T!

"Point.h" f12300, 6 lines

```
template<class T>
T polygonArea2(vector<Point<T>>& v) {
  T a = v.back().cross(v[0]);
  rep(i,0,sz(v)-1) a += v[i].cross(v[
      i+1]);
  return a;
}
```

PolygonCenter.h

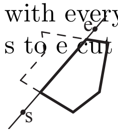
Description: Returns the center of mass for a polygon.
Time: $\mathcal{O}(n)$

"Point.h" 9706dc, 9 lines

```
typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
  P res(0, 0); double A = 0;
  for (int i = 0, j = sz(v) - 1; i <
      sz(v); j = i++) {
    res = res + (v[i] + v[j]) * v[j].
        cross(v[i]);
    A += v[j].cross(v[i]);
  }
  return res / A / 3;
}
```

PolygonCut.h

Description:
Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.



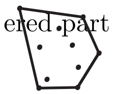
Usage: vector<P> p = ...;
p = polygonCut(p, P(0,0), P(1,0));

"Point.h", "lineIntersection.h" f2b7d4, 13 lines

```
typedef Point<double> P;
vector<P> polygonCut(const vector<P>&
    poly, P s, P e) {
  vector<P> res;
  rep(i,0,sz(poly)) {
    P cur = poly[i], prev = i ? poly[
        i-1] : poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) <
        0))
      res.push_back(lineInter(s, e,
          cur, prev).second);
    if (side)
      res.push_back(cur);
  }
  return res;
}
```

ConvexHull.h

Description:
Returns a vector of the points of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.



Time: $\mathcal{O}(n \log n)$

"Point.h" 310954, 13 lines

```
typedef Point<ll> P;
vector<P> convexHull(vector<P> pts) {
  if (sz(pts) <= 1) return pts;
  sort(all(pts));
  vector<P> h(sz(pts)+1);
  int s = 0, t = 0;
```

```
  for (int it = 2; it--; s = --t,
      reverse(all(pts)))
    for (P p : pts) {
      while (t >= s + 2 && h[t-2].
          cross(h[t-1], p) <= 0) t--;
      h[t++] = p;
    }
  return {h.begin(), h.begin() + t -
      (t == 2 && h[0] == h[1])};
}
```

HullDiameter.h

Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).

Time: $\mathcal{O}(n)$

"Point.h" c571b8, 12 lines

```
typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S)
    {
  int n = sz(S), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {S
      [0], S[0]}});
  rep(i,0,j)
    for (;; j = (j + 1) % n) {
      res = max(res, {(S[i] - S[j]).
          dist2(), {S[i], S[j]}});
      if ((S[(j + 1) % n] - S[j]).
          cross(S[i + 1] - S[i]) >= 0)
        break;
    }
  return res.second;
}
```

PointInsideHull.h

Description: Determine whether a point t lies inside a convex hull (CCW order, with no collinear points). Returns true if point lies within the hull. If strict is true, points on the boundary aren't included.

Time: $\mathcal{O}(\log N)$

"Point.h", "sideOf.h", "OnSegment.h" 71446b, 14 lines

```
typedef Point<ll> P;

bool inHull(const vector<P>& l, P p,
    bool strict = true) {
  int a = 1, b = sz(l) - 1, r = !
      strict;
  if (sz(l) < 3) return r &&
      onSegment(l[0], l.back(), p);
  if (sideOf(l[0], l[a], l[b]) > 0)
      swap(a, b);
  if (sideOf(l[0], l[a], p) >= r ||
      sideOf(l[0], l[b], p)<= -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (sideOf(l[0], l[c], p) > 0 ? b :
        a) = c;
  }
  return sgn(l[a].cross(l[b], p)) < r
      ;
}
```

LineHullIntersection.h

Description: Line-convex polygon intersection. The polygon must be ccw and have no collinear points. lineHull(line, poly) returns a pair describing the intersection of a line with the polygon: • $(-1,-1)$ if no collision, • $(i,-1)$ if touching the corner i, • (i,i) if along side $(i,i+1)$, • (i,j) if crossing sides $(i,i+1)$ and $(j,j+1)$. In the last case, if a corner i is crossed, this is treated as happening on side $(i,i+1)$. The points are returned in the same order as the line hits the polygon. `extrVertex` returns the point of a hull with the max projection onto a line.

Time: $\mathcal{O}(\log n)$

"Point.h" 7cf45b, 39 lines

```
#define cmp(i,j) sgn(dir.perp().cross
    (poly[(i)%n]-poly[(j)%n]))
#define extr(i) cmp(i + 1, i) >= 0 &&
     cmp(i, i - 1 + n) < 0
template <class P> int extrVertex(
    vector<P>& poly, P dir) {
  int n = sz(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo + 1 < hi) {
    int m = (lo + hi) / 2;
    if (extr(m)) return m;
    int ls = cmp(lo + 1, lo), ms =
        cmp(m + 1, m);
    (ls < ms || (ls == ms && ls ==
        cmp(lo, m)) ? hi : lo) = m;
  }
  return lo;
}

#define cmpL(i) sgn(a.cross(poly[i],
    b))
template <class P>
```

```
array<int, 2> lineHull(P a, P b,
    vector<P>& poly) {
  int endA = extrVertex(poly, (a - b)
      .perp());
  int endB = extrVertex(poly, (b - a)
      .perp());
  if (cmpL(endA) < 0 || cmpL(endB) >
      0)
    return {-1, -1};
  array<int, 2> res;
  rep(i,0,2) {
    int lo = endB, hi = endA, n = sz(
        poly);
    while ((lo + 1) % n != hi) {
      int m = ((lo + hi + (lo < hi ?
          0 : n)) / 2) % n;
      (cmpL(m) == cmpL(endB) ? lo :
          hi) = m;
    }
    res[i] = (lo + !cmpL(hi)) % n;
    swap(endA, endB);
  }
  if (res[0] == res[1]) return {res
      [0], -1};
  if (!cmpL(res[0]) && !cmpL(res[1]))
    switch ((res[0] - res[1] + sz(
        poly) + 1) % sz(poly)) {
      case 0: return {res[0], res[0]}
          ;
      case 2: return {res[1], res[1]}
          ;
    }
  return res;
}
```

7.4 Misc. Point Set Problems

ClosestPair.h

Description: Finds the closest pair of points.
Time: $\mathcal{O}(n \log n)$

"Point.h" ac41a6, 17 lines

```
typedef Point<ll> P;
pair<P, P> closest(vector<P> v) {
  assert(sz(v) > 1);
  set<P> S;
  sort(all(v), [](P a, P b) { return
      a.y < b.y; });
  pair<ll, pair<P, P>> ret{LLONG_MAX,
      {P(), P()}};
  int j = 0;
  for (P p : v) {
    P d{1 + (ll)sqrt(ret.first), 0};
    while (v[j].y <= p.y - d.x) S.
        erase(v[j++]);
    auto lo = S.lower_bound(p - d),
        hi = S.upper_bound(p + d);
    for (; lo != hi; ++lo)
      ret = min(ret, {(*lo - p).dist2
          (), {*lo, p}});
    S.insert(p);
  }
  return ret.second;
}
```

7.5 3D

PolyhedronVolume.h

Description: Magic formula for the volume of a polyhedron. Faces should point outwards.

3058c3, 6 lines

```
template<class V, class L>
double signedPolyVolume(const V& p,
    const L& trilist) {
  double v = 0;
  for (auto i : trilist) v += p[i.a].
      cross(p[i.b]).dot(p[i.c]);
  return v / 6;
}
```

Point3D.h

Description: Class to handle points in 3D space. T can be e.g. double or long long.

8058ae, 32 lines

```
template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z
      =0) : x(x), y(y), z(z) {}
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.
        y, p.z); }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p
        .y, p.z); }
  P operator+(R p) const { return P(x
      +p.x, y+p.y, z+p.z); }
  P operator-(R p) const { return P(x
      -p.x, y-p.y, z-p.z); }
  P operator*(T d) const { return P(x
      *d, y*d, z*d); }
  P operator/(T d) const { return P(x
      /d, y/d, z/d); }
  T dot(R p) const { return x*p.x + y
      *p.y + z*p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x
        *p.z, x*p.y - y*p.x);
  }
```

```
  T dist2() const { return x*x + y*y
     + z*z; }
  double dist() const { return sqrt((
     double)dist2()); }
  //Azimuthal angle (longitude) to x-
     axis in interval [-pi, pi]
  double phi() const { return atan2(y
     , x); }
  //Zenith angle (latitude) to the z-
     axis in interval [0, pi]
  double theta() const { return atan2
     (sqrt(x*x+y*y),z); }
  P unit() const { return *this/(T)
     dist(); } //makes dist()=1
  //returns unit vector normal to *
     this and p
  P normal(P p) const { return cross(
     p).unit(); }
  //returns point rotated 'angle'
     radians ccw around axis
  P rotate(double angle, P axis)
     const {
    double s = sin(angle), c = cos(
       angle); P u = axis.unit();
    return u*dot(u)*(1-c) + (*this)*c
        - cross(u)*s;
  }
};
```

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

611f07, 8 lines

```
double sphericalDistance(double f1,
    double t1,
     double f2, double t2, double
        radius) {
  double dx = sin(t2)*cos(f2) - sin(
     t1)*cos(f1);
  double dy = sin(t2)*sin(f2) - sin(
     t1)*sin(f1);
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*
     dz);
  return radius*2*asin(d/2);
}
```

Strings (8)

KMP.h

Description: pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.
Time: $\mathcal{O}(n)$

d4375c, 16 lines

```
vi pi(const string& s) {
  vi p(sz(s));
  rep(i,1,sz(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g
       -1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

vi match(const string& s, const
    string& pat) {
  vi p = pi(pat + '\0' + s), res;
  rep(i,sz(p)-sz(s),sz(p))
    if (p[i] == sz(pat)) res.
       push_back(i - 2 * sz(pat));
  return res;
}
```

Zfunc.h

Description: z[x] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)
Time: $\mathcal{O}(n)$

ee09e2, 12 lines

```
vi Z(const string& S) {
  vi z(sz(S));
  int l = -1, r = -1;
  rep(i,1,sz(S)) {
    z[i] = i >= r ? 0 : min(r - i, z[
       i - l]);
    while (i + z[i] < sz(S) && S[i +
       z[i]] == S[z[i]])
      z[i]++;
    if (i + z[i] > r)
      l = i, r = i + z[i];
  }
  return z;
}
```

Manacher.h

Description: For each position in a string, computes p[0][i] = half length of longest even palindrome around pos i, p[1][i] = longest odd (half rounded down).

Time: $\mathcal{O}(N)$

e7ad79, 13 lines

```
array<vi, 2> manacher(const string& s
   ) {
  int n = sz(s);
  array<vi,2> p = {vi(n+1), vi(n)};
  rep(z,0,2) for (int i=0,l=0,r=0; i
      < n; i++) {
    int t = r-i+!z;
    if (i<r) p[z][i] = min(t, p[z][l+
       t]);
    int L = i-p[z][i], R = i+p[z][i
       ]-!z;
    while (L>=1 && R+1<n && s[L-1] ==
        s[R+1])
      p[z][i]++, L--, R++;
    if (R>r) l=L, r=R;
  }
  return p;
}
```

MinRotation.h

Description: Finds the lexicographically smallest rotation of a string.

Usage: `rotate(v.begin(), v.begin()+minRotation(v), v.end());`

Time: $\mathcal{O}(N)$

d07a42, 8 lines

```
int minRotation(string s) {
  int a=0, N=sz(s); s += s;
  rep(b,0,N) rep(k,0,N) {
    if (a+k == b || s[a+k] < s[b+k])
       {b += max(0, k-1); break;}
    if (s[a+k] > s[b+k]) { a = b;
       break; }
  }
  return a;
}
```

SuffixArray.h

Description: Builds suffix array for a string. `sa[i]` is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size $n+1$, and `sa[0] = n`. The `lcp` array contains longest common prefixes for neighbouring strings in the suffix array: `lcp[i] = lcp(sa[i], sa[i-1])`, `lcp[0] = 0`. The input string must not contain any zero bytes.

Time: $\mathcal{O}(n \log n)$

38db9f, 23 lines

```
struct SuffixArray {
  vi sa, lcp;
  SuffixArray(string& s, int lim=256)
      { // or basic_string<int>
    int n = sz(s) + 1, k = 0, a, b;
    vi x(all(s)+1), y(n), ws(max(n,
       lim)), rank(n);
    sa = lcp = y, iota(all(sa), 0);
    for (int j = 0, p = 0; p < n; j =
        max(1, j * 2), lim = p) {
      p = j, iota(all(y), n - j);
      rep(i,0,n) if (sa[i] >= j) y[p
         ++] = sa[i] - j;
      fill(all(ws), 0);
      rep(i,0,n) ws[x[i]]++;
      rep(i,1,lim) ws[i] += ws[i -
         1];
      for (int i = n; i--;) sa[--ws[x
         [y[i]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] =
         0;
      rep(i,1,n) a = sa[i - 1], b =
         sa[i], x[b] =
        (y[a] == y[b] && y[a + j] ==
           y[b + j]) ? p - 1 : p++;
    }
    rep(i,1,n) rank[sa[i]] = i;
    for (int i = 0, j; i < n - 1; lcp
       [rank[i++]] = k)
      for (k && k--, j = sa[rank[i] -
         1];
         s[i + k] == s[j + k]; k++);
  }
};
```

SuffixTree.h

Description: Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r) into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r) substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).

Time: $\mathcal{O}(26N)$

aae0b8, 50 lines

```
struct SuffixTree {
  enum { N = 200010, ALPHA = 26 }; //
      N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
  string a; // v = cur node, q = cur
     position
  int t[N][ALPHA],l[N],r[N],p[N],s[N
     ],v=0,q=0,m=2;

  void ukkadd(int i, int c) { suff:
    if (r[v]<=q) {
```

```
      if (t[v][c]==-1) { t[v][c]=m; l[m]=i;
        p[m++]=v; v=s[v]; q=r[v]; goto suff; }
      v=t[v][c]; q=l[v];
    }
    if (q==-1 || c==toi(a[q])) q++; else {
      l[m+1]=i; p[m+1]=m; l[m]=l[v]; r[m]=q;
      p[m]=p[v]; t[m][c]=m+1; t[m][toi(a[q])]=v;
      l[v]=q; p[v]=m; t[p[m]][toi(a[l[m]])]=m;
      v=s[p[m]]; q=l[m];
      while (q<r[m]) { v=t[v][toi(a[q])]; q+=r[v]-l[v]; }
      if (q==r[m]) s[m]=v; else s[m]=m+2;
      q=r[v]-(q-r[m]); m+=2; goto suff;
    }
  }

  SuffixTree(string a) : a(a) {
    fill(r,r+N,sz(a));
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1],t[1]+ALPHA,0);
    s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] = 0;
    rep(i,0,sz(a)) ukkadd(i, toi(a[i]));
  }

  // example: find longest common substring (uses ALPHA = 28)
  pii best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node]) : 0;
    rep(c,0,ALPHA) if (t[node][c] != -1)
      mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
      best = max(best, {len, r[node] - len});
    return mask;
  }
  static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2));
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
  }
};
```

Hashing.h

Description: Self-explanatory methods for string hashing.

2d2a67, 44 lines

```
// Arithmetic mod 2^64-1. 2x slower than mod 2^64 and more
// code, but works on evil test data (e.g. Thue-Morse, where
// ABBA... and BAAB... of length 2^10 hash the same mod 2^64).
// "typedef ull H;" instead if you think test data is random,
// or work mod 10^9+7 if the Birthday paradox is not a problem.
typedef uint64_t ull;
struct H {
  ull x; H(ull x=0) : x(x) {}
  H operator+(H o) { return x + o.x + (x + o.x < x); }
  H operator-(H o) { return *this + ~o.x; }
  H operator*(H o) { auto m = (__uint128_t)x * o.x;
    return H((ull)m) + (ull)(m >> 64); }
  ull get() const { return x + !~x; }
  bool operator==(H o) const { return get() == o.get(); }
  bool operator<(H o) const { return get() < o.get(); }
};
static const H C = (ll)1e11+3; // (order ~ 3e9; random also ok)

struct HashInterval {
  vector<H> ha, pw;
  HashInterval(string& str) : ha(sz(str)+1), pw(ha) {
    pw[0] = 1;
    rep(i,0,sz(str))
      ha[i+1] = ha[i] * C + str[i],
      pw[i+1] = pw[i] * C;
  }
  H hashInterval(int a, int b) { // hash [a, b)
    return ha[b] - ha[a] * pw[b - a];
  }
```

```
};

vector<H> getHashes(string& str, int
    length) {
  if (sz(str) < length) return {};
  H h = 0, pw = 1;
  rep(i,0,length)
    h = h * C + str[i], pw = pw * C;
  vector<H> ret = {h};
  rep(i,length,sz(str)) {
    ret.push_back(h = h * C + str[i]
        - pw * str[i-length]);
  }
  return ret;
}

H hashString(string& s){H h{}; for(
    char c:s) h=h*C+c;return h;}
```

AhoCorasick.h

Description: Aho-Corasick automaton, used for multiple pattern matching. Initialize with AhoCorasick ac(patterns); the automaton start node will be at index 0. find(word) returns for each position the index of the longest word that ends there, or -1 if none. findAll(−, word) finds all words (up to $N\sqrt{N}$ many if no duplicate patterns) that start at each position (shortest first). Duplicate patterns are allowed; empty patterns are not. To find the longest words that start at each position, reverse all input. For large alphabets, split each symbol into chunks, with sentinel bits for symbol boundaries.
Time: construction takes $\mathcal{O}(26N)$, where $N =$ sum of length of patterns. find(x) is $\mathcal{O}(N)$, where N = length of x. findAll is $\mathcal{O}(NM)$.

f35677, 66 lines

```
struct AhoCorasick {
  enum {alpha = 26, first = 'A'}; //
      change this!
  struct Node {
    // (nmatches is optional)
    int back, next[alpha], start =
        -1, end = -1, nmatches = 0;
    Node(int v) { memset(next, v,
        sizeof(next)); }
  };
  vector<Node> N;
  vi backp;
  void insert(string& s, int j) {
    assert(!s.empty());
    int n = 0;
    for (char c : s) {
      int& m = N[n].next[c - first];
      if (m == -1) { n = m = sz(N); N
          .emplace_back(-1); }
      else n = m;
    }
    if (N[n].end == -1) N[n].start =
        j;
    backp.push_back(N[n].end);
    N[n].end = j;
    N[n].nmatches++;
  }
  AhoCorasick(vector<string>& pat) :
      N(1, -1) {
    rep(i,0,sz(pat)) insert(pat[i], i
        );
    N[0].back = sz(N);
    N.emplace_back(0);

    queue<int> q;
    for (q.push(0); !q.empty(); q.pop
        ()) {
      int n = q.front(), prev = N[n].
          back;
      rep(i,0,alpha) {
        int &ed = N[n].next[i], y = N
            [prev].next[i];
        if (ed == -1) ed = y;
        else {
          N[ed].back = y;
          (N[ed].end == -1 ? N[ed].
              end : backp[N[ed].start
              ])
            = N[y].end;
          N[ed].nmatches += N[y].
              nmatches;
          q.push(ed);
        }
      }
    }
  }
  vi find(string word) {
    int n = 0;
    vi res; // ll count = 0;
    for (char c : word) {
      n = N[n].next[c - first];
      res.push_back(N[n].end);
      // count += N[n].nmatches;
    }
    return res;
  }
  vector<vi> findAll(vector<string>&
      pat, string word) {
    vi r = find(word);
    vector<vi> res(sz(word));
    rep(i,0,sz(word)) {
      int ind = r[i];
      while (ind != -1) {
```

```
        res[i - sz(pat[ind]) + 1].
            push_back(ind);
        ind = backp[ind];
      }
    }
    return res;
  }
};
```

Various (9)

9.1 Intervals

IntervalContainer.h

Description: Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).
Time: $\mathcal{O}(\log N)$

edce47, 23 lines

```
set<pii>::iterator addInterval(set<
    pii>& is, int L, int R) {
  if (L == R) return is.end();
  auto it = is.lower_bound({L, R}),
      before = it;
  while (it != is.end() && it->first
      <= R) {
    R = max(R, it->second);
    before = it = is.erase(it);
  }
  if (it != is.begin() && (--it)->
      second >= L) {
    L = min(L, it->first);
    R = max(R, it->second);
    is.erase(it);
  }
  return is.insert(before, {L,R});
}

void removeInterval(set<pii>& is, int
    L, int R) {
  if (L == R) return;
  auto it = addInterval(is, L, R);
  auto r2 = it->second;
  if (it->first == L) is.erase(it);
  else (int&)it->second = L;
  if (R != r2) is.emplace(R, r2);
}
```

IntervalCover.h

Description: Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add `|| R.empty()`. Returns empty set on failure (or if G is empty).
Time: $\mathcal{O}(N \log N)$

9e9d8d, 19 lines

```
template<class T>
vi cover(pair<T, T> G, vector<pair<T,
    T>> I) {
  vi S(sz(I)), R;
  iota(all(S), 0);
  sort(all(S), [&](int a, int b) {
    return I[a] < I[b]; });
  T cur = G.first;
  int at = 0;
  while (cur < G.second) { // (A)
    pair<T, int> mx = make_pair(cur,
        -1);
    while (at < sz(I) && I[S[at]].
        first <= cur) {
      mx = max(mx, make_pair(I[S[at
          ]].second, S[at]));
      at++;
    }
    if (mx.second == -1) return {};
    cur = mx.first;
    R.push_back(mx.second);
  }
  return R;
}
```

9.2 Optimization tricks

9.2.1 Bit hacks

- `x & -x` is the least bit in `x`.
- `for (int x = m; x; ) { --x &= m; ... }` loops over all subset masks of `m` (except `m` itself).
- `c = x&-x, r = x+c; (((r^x) >> 2)/c) |` is the next number after `x` with the same number of bits set.
- `rep(b,0,K) rep(i,0,(1 << K))`
  `if (i & 1 << b) D[i] += D[i^(1 << b`
  computes all sums of subsets.