Contents

1 DP

1.1 SOS DP

```
/// O( N * 2^N )
///memory optimized version
for (int i = 0; i < (1 << N); ++i) F[i] = A[i];
for (int i = 0; i < N; ++i) for (int mask = 0; mask < (1 << N); ++mask)
  if (mask & (1 << i)) F[mask] += F[mask ^ (1 << i)];
/// How many pairs in ara[] such that (ara[i] & ara[j]) = 0
/// N --> Max number of bits of any array element
const int N = 20;
int inv = (1 << N) - 1;
int F[(1 << N) + 10];
int ara[MAX];
/// ara is 0 based
long long howManyZeroPairs(int n, int ara[]) {
 CLR(F);
 for (int i = 0; i < n; i++) F[ara[i]]++;
 for (int i = 0; i < N; ++i) for (int mask = 0; mask < (1 << N); ++mask) {
   if (mask & (1 << i)) F[mask] += F[mask ^ (1 << i)];
  }
 long long ans = 0;
 for (int i = 0; i < n; i++) ans += F[ara[i] ^ inv];
 return ans;
}
/// F[mask] = sum of A[i] given that (i&mask)=mask
for (int i = 0; i < (1 << N); ++i) F[i] = A[i];
for (int i = 0; i < N; ++i)
 for (int mask = (1 << N) - 1; mask >= 0; --mask) {
  if (!(mask & (1 << i))) F[mask] += F[mask | (1 << i)];
 }
/// Number of subsequences of ara[0:n-1] such that
/// sub[0] & sub[2] & ... & sub[k-1] = 0
const int N = 20;
int inv = (1 << N) - 1;
int F[(1 << N) + 10];
int ara[MAX];
int p2[MAX]; /// p2[i] = 2^i
///0 based array
int howManyZeroSubSequences(int n, int ara[]) {
 CLR(F);
 for (int i = 0; i < n; i++) F[ara[i]]++;
 for (int i = 0; i < N; ++i)
  for (int mask = (1 << N) - 1; mask >= 0; --mask) {
   if (!(mask & (1 << i)))
    F[mask] += F[mask | (1 << i)];
  }
 int ans = 0;
 for (int mask = 0; mask < (1 << N); mask++) {
  if (__builtin_popcount(mask) & 1) ans = sub(ans, p2[F[mask]]);
  else ans = add(ans, p2[F[mask]]);
 }
 return ans;
}
/// Number of subsequences of ara[0:n-1] such that
/// sub[0] | sub[2] | ... | sub[k-1] = Q
int F[(1 << 20) + 10], ara[MAX];
int p2[MAX]; /// p2[i] = 2^i
/// ara is 0 based
int howManySubsequences(int n, int ara[], int m, int Q) {
 CLR(F);
 for (int i = 0; i < n; i++) F[ara[i]]++;
 if (Q == 0) return sub(p2[F[0]], 1);
 for (int i = 0; i < m; ++i)
  for (int mask = 0; mask < (1 << m); ++mask) {
   if (mask & (1 << i))
    F[mask] += F[mask ^ (1 << i)];
  }
 int ans = 0;
 for (int mask = 0; mask < (1 << m); mask++) {
  if (mask & Q != mask) continue;
  if (__builtin_popcount(mask ^ Q) & 1) ans = sub(ans, p2[F[mask]]);
  else ans = add(ans, p2[F[mask]]);
 }
 return ans;
}
```

2 Data Structures

2.1 2D Fenwick Tree

```
struct FenwickTree2D {
 vector<vector<int>> bit;
 int n, m;
// init(...) { ... }
 int sum(int x, int y) {
  int ret = 0;
  for (int i = x; i >= 0; i = (i & (i + 1)) - 1)
   for (int j = y; j >= 0; j = (j & (j + 1)) - 1)
    ret += bit[i][j];
  return ret;
 }
 void add(int x, int y, int delta) {
  for (int i = x; i < n; i = i | (i + 1))
   for (int j = y; j < m; j = j | (j + 1))
    bit[i][j] += delta;
 }
}
```

2.2 Fenwick Tree

```
int bit[1000], arra[1000];
int n;
void update( int idx, int val ) {
 for ( int i = idx; i <= n; i += i & (-i) )
   bit[i] += val;
 return;
}
int query( int idx ) {
 int sum = 0;
 for ( int i = idx; i > 0; i -= i & (-i) ) sum += bit[i];
 return sum;
}
```

2.3 LIS

```
int lis(vector<int> a) {
 int n = a.size();
 vector<int>d(n + 1, INF);
 d[0] = -INF;
 for (int i = 0; i < n; i++) {
  int j = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
  if (d[j - 1] < a[i] and a[i] < d[j]) d[j] = a[i];
 }
 int ret = 0;
 for (int i = 1; i <= n; i++)
  if (d[i] < INF) ret = i;
 return ret;
}
```

2.4 MO's Algo

```
const int mx = 100005;
const int sz = 100005;
struct query {
 int l, r, id;
 bool operator<(const query &a) const {
  int x = l / sz; int y = a.l / sz;
  if (x != y) return x < y;
  if (x % 2) return r < a.r;
  return r > a.r;
 }
} ques[mx];
void add(int indx) { }
void baad(int indx) { }
void solve() {
 int l = 0;
 int r = -1;
 sort(ques + 1, ques + q + 1);
 for (int i = 1; i <= q; i++) {
  while (l > ques[i].l) add(--l);
  while (r < ques[i].r) add(++r);
  while (l < ques[i].l) baad(l++);
  while (r > ques[i].r) baad(r--);
```

```
  ans[ques[i].id] = sum[now];
 }
 for (int i = 1; i <= q; i++) cout << ans[i] <<
 ↪ " ";
}
```

2.5 PBDS

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template<class T>
using ordered_set = tree<T, null_type, less<T>,
      rb_tree_tag,
 ↪ tree_order_statistics_node_update> ;
/*
0 based indexing
1) insert(value)
2) erase(value)
3) order_of_key(value) // Number of items
 ↪ strictly smaller than value
4) *find_by_order(k) : K-th element in a
 ↪ set(counting from zero)
*/
```

2.6 Persistent Seg Tree (Pointer)

```
struct node {
 ll sum;
 node *left, *right;
 node(ll _sum = 0) {
  sum = _sum;
  left = right = NULL;
 }
 void build(int l, int r) {
  if (l == r) return;
  left = new node();
  right = new node();
  int mid = (l + r) / 2;
  left->build(l, mid);
  right->build(mid + 1, r);
 }
 node *update(int l, int r, int i, int x) {
  if (l > i || r < i) return this;
  if (l == r) return new node(x);
  int mid = (l + r) / 2;
  node *ret = new node();
  ret->left = left->update(l, mid, i, x);
  ret->right = right->update(mid + 1, r, i, x);
  ret->sum = ret->left->sum + ret->right->sum;
  return ret;
 }
 ll query(int tL, int tR, int rL, int rR) {
  if (tL > rR || tR < rL) return 0;
  if (tL >= rL && tR <= rR) return sum;
  int mid = (tL + tR) / 2;
  ll a = left->query(tL, mid, rL, rR);
  ll b = right->query(mid + 1, tR, rL, rR);
  return a + b;
 }
 int size() { return sizeof(*this) /
 ↪ sizeof(node*); }
};
const int mx = 2e5 + 5;
node *root[mx];
void solve() {
 root[0] = new node();
 root[0]->build(1, n);
 for (int i = 1; i <= n; i++) {
  int x;
  cin >> x;
  root[0] = root[0]->update(1, n, i, x);
 }
 int sz = 0;
 while (q--) {
  int op; cin >> op;
  if (op == 1) {
   int version, i, x;
   cin >> version >> i >> x;
   version--;
   root[version] = root[version]->update(1, n,
 ↪  i, x);
  }
  else if (op == 2) {
   int version, l, r;
   cin >> version >> l >> r;
   version--;
   cout << root[version]->query(1, n, l, r) <<
 ↪  "\n";
  }
  else {
   int version;
   cin >> version;
   version--;
   root[++sz] = root[version];
  }
 }
}
```

2.7 RMQ

```
template<typename T>
struct sparse_table {
 vector<T>ara;
 vector<int>logs;
 vector<vector<T>>table;
 sparse_table(int n) {
  ara.resize(n + 1);
  logs.resize(n + 1);
 }
 T func(T a, T b) { }
 void build(int n) {
  logs[1] = 0;
  for (int i = 2; i <= n; i++) logs[i] = logs[i
 ↪  / 2] + 1;
  table.resize(n + 1, vector<T>(logs[n] + 1));
  for (int i = 1; i <= n; i++) table[i][0] =
 ↪  ara[i];
  for (int j = 1; j <= logs[n]; j++) {
   int sz = 1 << j;
   for (int i = 1; i + sz - 1 <= n; i++)
    table[i][j] = func(table[i][j - 1], table[i
 ↪  + sz / 2][j - 1]);
  }
 }
 T query(int l, int r) {
  int d = logs[r - l + 1];
  return func(table[l][d], table[r - (1 << d) +
 ↪  1][d]);
 }
};
```

2.8 Seg Tree(Inz)

```
class SegmentTree {
#define Lc(idx) idx * 2
#define Rc(idx) idx * 2 + 1
public:
 struct node {
  int value;
  int lazy;
  node() {
   this->value = ??;
   this->lazy = ??;
  }
 };
 vector <node> segT;
 vector <int> A;
 SegmentTree(int sz) {
// need to clear!
  segT.resize(4 * sz + 10);
  A.resize(sz + 1);
 }
 node Merge(node L, node R) {
  node F;
  F = ??
      return F;
 }
 void Relax(int L, int R, int idx) {
//Do something
  segT[idx].lazy = ??; //after Relaxing
 }
 void MakeSegmentTree(int L, int R, int idx) {
  if (L == R) {
   segT[idx].value = ??;
   return;
  }
  int M = (L + R) / 2;
  MakeSegmentTree(L, M, Lc(idx));
  MakeSegmentTree(M + 1, R, Rc(idx));
  segT[idx] = Merge(segT[Lc(idx)],
 ↪  segT[Rc(idx)]);
 }
 node RangeQuery(int L, int R, int idx, int l,
 ↪  int r) {
```

```
  Relax(L, R, idx);
  node F;
  if (L > r || R < l) return F;
  if (L >= l && R <= r) return segT[idx];
  int M = (L + R) / 2;
  F = Merge(RangeQuery(L, M, Lc(idx), l, r),
   RangeQuery(M + 1, R, Rc(idx), l, r));
  segT[idx] = Merge(segT[Lc(idx)],
   segT[Rc(idx)]); //is it useful?
  return F;
 }
 void RangeUpdate(int L, int R, int idx, int l,
   int r, int lz) {
  Relax(L, R, idx);
  if (L > r || R < l) return;
  if (L >= l && R <= r) {
// Do something
   segT[idx].lazy = ??;
   Relax(L, R, idx);
   return;
  }
  int M = (L + R) / 2;
  RangeUpdate(L, M, Lc(idx), l, r, lz);
  RangeUpdate(M + 1, R, Rc(idx ), l, r, lz);
  segT[idx] = Merge( segT[Lc(idx)],
   segT[Rc(idx)]);
 }
};
```

2.9 Seg Tree(Lazy Prop)

```
int n, q, arra[100005];
struct idk {
 int sum, prop;
} tree[300005];
void init( int node, int b, int e ) {
 if ( b == e ) {
  tree[node].sum = arra[b];
  return;
 }
 int left = node * 2;
 int right = node * 2 + 1;
 int mid = (b + e) / 2;
 init(left, b, mid);
 init(right, mid + 1, e);
 tree[node].sum = tree[left].sum +
   tree[right].sum;
 return;
}
void update( int node, int b, int e, int i, int
   j, int val ) {
 if ( b > j || e < i ) return;
 if ( b >= i && e <= j ) {
  tree[node].sum += (e - b + 1) * val;
  tree[node].prop += val;
  return;
 }
 int left = node * 2;
 int right = node * 2 + 1;
 int mid = (b + e) / 2;
 update( left, b, mid, i, j, val );
 update( right, mid + 1, e, i, j, val );
 tree[node].sum = tree[left].sum +
    tree[right].sum + (e - b + 1) *
    tree[node].prop;
 return;
}
int query( int node, int b, int e, int i, int
   j, int carry ) {
 if ( b > j || e < i ) return 0;
 if ( b >= i && e <= j ) return tree[node].sum
   + (e - b + 1) * carry;
 int left = node * 2;
 int right = node * 2 + 1;
 int mid = (b + e) / 2;
 int p1 = query(left, b, mid, i, j, carry +
   tree[node].prop );
 int p2 = query(right, mid + 1, e, i, j, carry
   + tree[node].prop );
 return p1 + p2;
}
```

2.10 Seg Tree(point update, range query)

```
ll tree[4 * N];
inline ll merge(ll a, ll b) { return a + b; }
void update(int rt, int l, int r, int p, ll v) {
 if (l == r) return void(tree[rt] = v);
 int m = l + r >> 1, lc = rt << 1, rc = lc | 1;
 if (p <= m) update(lc, l, m, p, v);
 else update(rc, m + 1, r, p, v);
 tree[rt] = merge(tree[lc], tree[rc]);
}
ll query(int rt, int l, int r, int b, int e) {
 if (l > e or r < b or b > e) return 0;
 if (l >= b and r <= e) return tree[rt];
 int m = l + r >> 1, lc = rt << 1, rc = lc | 1;
 return merge(query(lc, l, m, b, e), query(rc,
   m + 1, r, b, e));
}
```

2.11 Seg Tree(range update, range query)

```
ll tree[4 * N], lazy[4 * N];
inline ll merge(ll a, ll b) { return a + b; }
void push(int rt, int l, int r) {
 if (l ^ r) {
  lazy[rt << 1] += lazy[rt];
  lazy[rt << 1 | 1] += lazy[rt];
 }
 tree[rt] += (r - l + 1) * lazy[rt];
 lazy[rt] = 0;
}
void update(int rt, int l, int r, int b, int e,
   ll v) {
 if (lazy[rt]) push(rt, l, r);
 if (l > e or r < b or b > e) return;
 if (l >= b and r <= e) {
  lazy[rt] += v;
  return push(rt, l, r);
 }
 int m = l + r >> 1, lc = rt << 1, rc = lc | 1;
 update(lc, l, m, b, e, v); update(rc, m + 1,
   r, b, e, v);
 tree[rt] = merge(tree[lc], tree[rc]);
}
ll query(int rt, int l, int r, int b, int e) {
 if (lazy[rt]) push(rt, l, r);
 if (l > e or r < b or b > e) return 0;
 if (l >= b and r <= e) return tree[rt];
 int m = l + r >> 1, lc = rt << 1, rc = lc | 1;
 return merge(query(lc, l, m, b, e), query(rc,
   m + 1, r, b, e));
}
```

2.12 Sqrt Decomposition

```
int block_size = ??;
int Block[block_size + 5];
int getBlock(int idx) {
 return (idx + block_size - 1) / block_size;
   //for 1-base index
 return idx / block_size; //for 0-base index
}
int getQueryAns(int L, int R) //0-base index
{
 int ANS = 0;
 int CL = L / block_size;
 int CR = R / block_size;
 if (CL == CR) {
  for (int i = L; i <= R; ++i)
   ANS += ArrName[i];
 }
 else {
  for (int i = L, LM = (CL + 1) * block_size -
   1; i <= LM; ++i)
   ANS += ArrName[i];
  for (int i = CL + 1; i <= CR - 1; ++i)
   ANS += Block[i];
  for (int i = CR * block_size; i <= R; ++i)
   ANS += ArrName[i];
 }
 return ANS;
}
//Update : Block[ idx / block_size ] += ??
```

3 Geometry

3.1 2D point line- segment

```
const double PI = acos(-1.0);
const double EPS = 1e-12;
/***
```

```cpp
u . v = |u|*|v|*cos(theta)
= u.x*v.x + u.y*v.y
= How much parallel they are
= Dot product does not change if one vector move perpendicular to the other
u x v = |u|*|v|*sin(theta)
= u.x*v.y - v.x*u.y
= How much perpendicular they are
= Cross product does not change if one vector move parallel to the other
dot(a-b,a-b) returns squared distance between pt a and pt b
***/
struct pt {
 double x, y;
 pt() {}
 pt(double x, double y) : x(x) , y(y) {}
 pt operator + (const pt &p) const {
  return pt( x + p.x , y + p.y );
 }
 pt operator - (const pt &p) const {
  return pt( x - p.x , y - p.y );
 }
 pt operator * (double c) const {
  return pt( x * c , y * c );
 }
 pt operator / (double c) const {
  return pt( x / c , y / c );
 }
 bool operator == (const pt &p) const {
  return ( fabs( x - p.x ) < EPS && fabs( y - p.y ) < EPS );
 }
 bool operator != (const pt &p) const {
  return !(pt(x, y) == p);
 }
};
ostream& operator << (ostream& os, pt p) {
 return os << "(" << p.x << "," << p.y << ")";
}
// u.v = |u|*|v|*cos(theta)
inline double dot(pt u, pt v) {
 return u.x * v.x + u.y * v.y;
}
// a x b = |a|*|b|*sin(theta)
inline double cross(pt u, pt v) {
 return u.x * v.y - u.y * v.x;
}
// returns |u|
inline double norm(pt u) { return sqrt(dot(u, u)); }
// returns angle between two vectors
inline double angle(pt u, pt v) {
 double cosTheta = dot(u, v) / norm(u) / norm(v);
 return acos(max(-1.0, min(1.0, cosTheta))); // keeping cosTheta in [-1, 1]
}
// returns ang radian rotated version of vector u
// ccw rotation if angle is positive else cw rotation
inline pt rotate(pt u, double ang) {
 return pt( u.x * cos(ang) - u.y * sin(ang) , u.x * sin(ang) + u.y * cos(ang) );
}
// returns a vector perpendicular to v
inline pt perp(pt u) { return pt( -u.y , u.x ); }
// returns 2*area of triangle
inline double triArea2(pt a, pt b, pt c) {
 return cross(b - a, c - a);
}
// compare function for angular sort around point P0
inline bool comp(pt P0, pt a, pt b) {
 double d = triArea2(P0, a, b);
 if (d < 0) return false;
 if (d == 0 && dot(P0 - a, P0 - a) > dot(P0 - b, P0 - b) ) return false;
 return true;
}
/***
if line equation is, ax + by = c
then,
v --> direction vector of the line (b,-a)
c --> v cross p
p --> Any point(vector) on the line
side(p) = ( v cross p) - c )
= triArea2(origin,v,p)
if side(p) is,
positive --> p is above the line
zero --> p is on the line
negative --> p is below the line
***/
struct line {
 pt v;
 double c;
 line(pt v, double c) : v(v), c(c) {}
// From equation ax + by = c
 line(double a, double b, double c) : v( {b, -a}), c(c) {}
// From points p and q
 line(pt p, pt q) : v(q - p), c(cross(v, p)) {}
// |v| * dist
// dist --> distance of p from the line
 double side(pt p) { return cross(v, p) - c; }
// better to using sqDist than dist
 double dist(pt p) {
  return abs(side(p)) / norm(v);
 }
 double sqDist(pt p) {
  return side(p) * side(p) / dot(v, v);
 }
// perpendicular line through point p
// 90deg ccw rotated line
 line perpThrough(pt p) {
  return {p, p + perp(v)};
 }
// translates a line by vector t(dx,dy)
// every point (x,y) of previous line is translated to (x + dx, y + dy)
 line translate(pt t) {
  return {v, c + cross(v, t)};
 }
// for every point
// distance between previous position and current position is dist
 line shiftLeft(double dist) {
  return {v, c + dist * norm(v)};
 }
// projection of point p on the line
 pt projection(pt p) {
  return p - perp(v) * side(p) / dot(v, v);
 }
// reflection of point p wrt the line
 pt reflection(pt p) {
  return p - perp(v) * side(p) * 2.0 / dot(v, v);
 }
};
inline bool lineLineIntersection(line l1, line l2, pt &out) {
 double d = cross(l1.v, l2.v);
 if (d == 0) return false;
 out = (l2.v * l1.c - l1.v * l2.c) / d;
 return true;
}
// interior = true for interior bisector
// interior = false for exterior bisector
inline line bisector(line l1, line l2, bool interior) {
 assert(cross(l1.v, l2.v) != 0); // l1 and l2 cannot be parallel!
 double sign = interior ? 1 : -1;
 return {l2.v / norm(l2.v) + (l1.v * sign) / norm(l1.v),
         l2.c / norm(l2.v) + (l1.c * sign) / norm(l1.v)};
}
/*** Segment ***/
/// C --> A circle which have diameter ab
/// returns true if point p is inside C or on the border of C
inline bool inDisk(pt a, pt b, pt p) {
 return
    , | dot(a - p, b - p) <= 0;
}
/// returns true if point p is on the segment
inline bool onSegment(pt a, pt b, pt p) {
 return triArea2(a, b, p) == 0 && inDisk(a, b, p);
}
inline bool segSegIntersection(pt a, pt b, pt c, pt d, pt &out) {
 if (onSegment(a, b, c)) return out = c, true;
 if (onSegment(a, b, d)) return out = d, true;
 if (onSegment(c, d, a)) return out = a, true;
```

```
 if (onSegment(c, d, b)) return out = b, true;
 double oa = triArea2(c, d, a);
 double ob = triArea2(c, d, b);
 double oc = triArea2(a, b, c);
 double od = triArea2(a, b, d);
 if (oa * ob < 0 && oc * od < 0) {
  out = (a * ob - b * oa) / (ob - oa);
  return true;
 }
 return false;
}
// returns distance between segment ab and
 ↪ point p
inline double segPointDist(pt a, pt b, pt p) {
 if ( norm(a - b) == 0 ) {
  line l(a, b);
  pt pr = l.projection(p);
  if (onSegment(a, b, p)) return l.dist(p);
 }
 return min(norm(a - p), norm(b - p));
}
// returns distance between segment ab and
 ↪ segment cd
inline double segSegDist(pt a, pt b, pt c, pt
 ↪ d) {
 double oa = triArea2(c, d, a);
 double ob = triArea2(c, d, b);
 double oc = triArea2(a, b, c);
 double od = triArea2(a, b, d);
 if (oa * ob < 0 && oc * od < 0) return 0; //
 ↪ proper intersection
// If the segments don't intersect, the result
 ↪ will be minimum of these four
 return min({segPointDist(a, b, c),
 ↪ segPointDist(a, b, d),
         segPointDist(c, d, a),
 ↪ segPointDist(c, d, b)
        });
}
```

3.2 Circle-line intersection

```
struct Point {
 double x, y;
 Point(double px, double py) {
  x = px;
  y = py;
 }
 Point sub(Point p2) {
  return Point(x - p2.x, y - p2.y);
 }
 Point add(Point p2) {
  return Point(x + p2.x, y + p2.y);
 }
 double distance(Point p2) {
  return sqrt((x - p2.x) * (x - p2.x) + (y -
 ↪ p2.y) * (y - p2.y));
 }
 Point normal() {
  double length = sqrt(x * x + y * y);
  return Point(x / length, y / length);
 }
 Point scale(double s) {
  return Point(x * s, y * s);
 }
};
struct line // Creates a line with equation ax
 ↪ + by + c = 0
{
 double a, b, c;
 line() {}
 line( Point p1, Point p2 ) {
  a = p1.y - p2.y;
  b = p2.x - p1.x;
  c = p1.x * p2.y - p2.x * p1.y;
 }
};
inline bool eq(double a, double b) {
 return fabs( a - b ) < eps;
}
struct Circle {
 double x, y, r, left, right;
 Circle () {}
 Circle(double cx, double cy, double cr) {
  x = cx;
  y = cy;
  r = cr;
  left = x - r;
  right = x + r;
 }
 pair<Point, Point> intersections(Circle c) {
  Point P0(x, y);
  Point P1(c.x, c.y);
  double d, a, h;
  d = P0.distance(P1);
  a = (r * r - c.r * c.r + d * d) / (2 * d);
  h = sqrt(r * r - a * a);
  Point P2 = P1.sub(P0).scale(a / d).add(P0);
  double x3, y3, x4, y4;
  x3 = P2.x + h * (P1.y - P0.y) / d;
  y3 = P2.y - h * (P1.x - P0.x) / d;
  x4 = P2.x - h * (P1.y - P0.y) / d;
  y4 = P2.y + h * (P1.x - P0.x) / d;
  return pair<Point, Point>(Point(x3, y3),
 ↪   Point(x4, y4));
 }
};
inline double Distance( Point a, Point b ) {
 return sqrt( ( a.x - b.x ) * ( a.x - b.x ) + (
 ↪   a.y - b.y ) * ( a.y - b.y ) );
}
inline double Distance( Point P, line L ) {
 return fabs( L.a * P.x + L.b * P.y + L.c ) /
 ↪   sqrt( L.a * L.a + L.b * L.b );
}
bool intersection(Circle C, line L, Point &p1,
 ↪   Point &p2) {
 if ( Distance( {C.x, C.y}, L ) > C.r + eps )
 ↪   return false;
 double a, b, c, d, x = C.x, y = C.y;
 d = C.r * C.r - x * x - y * y;
 if ( eq( L.a, 0) ) {
  p1.y = p2.y = -L.c / L.b;
  a = 1;
  b = 2 * x;
  c = p1.y * p1.y - 2 * p1.y * y - d;
  d = b * b - 4 * a * c;
  d = sqrt( fabs (d) );
  p1.x = ( b + d ) / ( 2 * a );
  p2.x = ( b - d ) / ( 2 * a );
 }
 else {
  a = L.a * L.a + L.b * L.b;
  b = 2 * ( L.a * L.a * y - L.b * L.c - L.a *
 ↪   L.b * x);
  c = L.c * L.c + 2 * L.a * L.c * x - L.a * L.a
 ↪   * d;
  d = b * b - 4 * a * c;
  d = sqrt( fabs(d) );
  p1.y = ( b + d ) / ( 2 * a );
  p2.y = ( b - d ) / ( 2 * a );
  p1.x = ( -L.b * p1.y - L.c ) / L.a;
  p2.x = ( -L.b * p2.y - L.c ) / L.a;
 }
 return true;
}
```

3.3 Circle

```
struct circle {
 pt c;
 double r;
 circle() {}
 circle(pt c, double r) : c(c) , r(r) {}
};
/* returns circumcircle of a triangle
the radius of circumcircle --> intersection
 ↪ point of the perpendicular
bisectors of the three sides */
circle circumCircle(pt a, pt b, pt c) {
 b = b - a, c = c - a; // consider coordinates
 ↪ relative to point a
 assert(cross(b, c) != 0); // no circumcircle
 ↪ if A, B, C are co - linear
// detecting the intersection point using the
 ↪ same technique used in line line
 ↪ intersection
 pt center = a + ( perp( b * dot(c, c) - c *
 ↪ dot(b, b) ) / cross(b, c) / 2 );
 return {center, norm(center - a)};
}
int sgn(double val) {
 if (val > 0) return 1;
 else if (val == 0) return 0;
 else return -1;
}
/* returns number of intersection points
 ↪ between a line and a circle
```

```cpp
O --> Center
I,J --> Intersection points
P -- > Projection of O onto line l
IP = JP = h , OP = d */
int circleLineIntersection(circle c, line l,
    pair<pt, pt> &out) {
  double h2 = c.r * c.r - l.sqDist(c.c); // h^2
  if (h2 >= 0) { // the line touches the circle
    pt p = l.proj(c.c); // point P
    pt h = l.v * sqrt(h2) / norm(l.v); // vector
        parallel to l, of length h
    out = {p - h, p + h}; // {I,J}
  }
  return 1 + sgn(h2); // number of intersection
      points
}
/* returns number of intersection points between
, two circles
O_i --> Center of circle i
I,J --> Intersection points
P -- > Projection of O onto line IJ
IP = JP = h , O_10_2 = d */
int circleCircleIntersection(circle c1, circle
    c2, pair<pt, pt> &out) {
  pt d = c2.c - c1.c; double d2 = dot(d, d); //
      d^2
  if (d2 == 0) { // concentric circle
    assert(c1.r != c2.r); // same circle
    return 0;
  }
  double pd = (d2 + c1.r * c1.r - c2.r * c2.r) /
      2; // = | O_1P | * d
  double h2 = c1.r * c1.r - pd * pd / d2; // =
      h^2
  if (h2 >= 0) {
    pt p = c1.c + d * pd / d2, h = perp(d) *
        sqrt(h2 / d2);
    out = {p - h, p + h};
  }
  return 1 + sgn(h2);
}
/* inner --> if true returns inner tangents
* if the radius of c2 is 0, returns tangents
    that go through the center
of circle c2 (value of inner is does not matter
    in this case)
* if there are 2 tangents, it fills out with
    two pairs of points: the pairs
of tangency points on each circle (P1; P2), for
    each of the tangents
* if there is 1 tangent, the circles are
    tangent to each other at some point
P, out just contains P 4 times, and the tangent
    line can be found as
line(c1.c,p).perpThrough(p)
* if there are 0 tangents, it does nothing
* if the circles are identical, it aborts. */
int tangents(circle c1, circle c2, bool inner,
    vector < pair <pt, pt> > &out) {
  if (inner) c2.r = -c2.r;
  pt d = c2.c - c1.c;
  double dr = c1.r - c2.r, d2 = dot(d, d), h2 =
      d2 - dr * dr;
  if (d2 == 0 || h2 < 0) {//assert(h2 != 0);
    return 0;
  }
  for (double sign : { -1, 1}) {
    pt v = (d * dr + perp(d) * sqrt(h2) * sign) /
        d2;
    out.push_back({c1.c + v * c1.r, c2.c + v *
        c2.r});
  }
  return 1 + (h2 > 0);
}
```

3.4 Convex Hull

```cpp
struct Point {
  int x, y;
};
Point p0;
Point nextToTop(stack<Point> &S) {
  Point p = S.top();
  S.pop();
  Point res = S.top();
  S.push(p);
  return res;
}
void swap(Point &p1, Point &p2) {
  Point temp = p1;
  p1 = p2;
  p2 = temp;
}
int distSq(Point p1, Point p2) {
  return (p1.x - p2.x) * (p1.x - p2.x) +
         (p1.y - p2.y) * (p1.y - p2.y);
}
int orientation(Point p, Point q, Point r) {
  int val = (q.y - p.y) * (r.x - q.x) -
            (q.x - p.x) * (r.y - q.y);
  if (val == 0) return 0;
  return (val > 0) ? 1 : 2;
}
int compare(const void *vp1, const void *vp2) {
  Point *p1 = (Point *)vp1;
  Point *p2 = (Point *)vp2;
  int o = orientation(p0, *p1, *p2);
  if (o == 0)
    return (distSq(p0, *p2) >= distSq(p0, *p1)) ?
        -1 : 1;
  return (o == 2) ? -1 : 1;
}
void convexHull(Point points[], int n) {
  int ymin = points[0].y, min = 0;
  for (int i = 1; i < n; i++) {
    int y = points[i].y;
    if ((y < ymin) || (ymin == y && points[i].x <
        points[min].x))
      ymin = points[i].y, min = i;
  }
  swap(points[0], points[min]);
  p0 = points[0];
  qsort(&points[1], n - 1, sizeof(Point),
      compare);
  int m = 1;
  for (int i = 1; i < n; i++) {
    while (i < n - 1 && orientation(p0,
        points[i], points[i + 1]) == 0)
      i++;
    points[m] = points[i];
    m++;
  }
  if (m < 3) return;
  stack<Point> S;
  S.push(points[0]);
  S.push(points[1]);
  S.push(points[2]);
  for (int i = 3; i < m; i++) {
    while (S.size() > 1 &&
        orientation(nextToTop(S), S.top(),
        points[i]) != 2)
      S.pop();
    S.push(points[i]);
  }
  while (!S.empty()) {
    Point p = S.top();
    cout << "(" << p.x << ", " << p.y << ")" <<
        endl;
    S.pop();
  }
}
int main() {
  Point points[100005];
  int n;
  scanf("%d", &n);
  for ( int i = 0; i < n; i++ )scanf("%d %d",
      &points[i].x, &points[i].y);
  convexHull(points, n);
  return 0;
}
```

3.5 Point inside Poly (Ray Shooting)

```cpp
// if strict, returns false when a is on the
    boundary
inline bool insidePoly(pt *P, int np, pt a,
    bool strict = true) {
  int numCrossings = 0;
  for (int i = 0; i < np; i++) {
    if (onSegment(P[i], P[(i + 1) % np], a))
        return !strict;
    numCrossings += crossesRay(a, P[i], P[(i + 1)
        % np]);
  }
```

```
  return (numCrossings & 1); // inside if odd
  ↪  number of crossings
}
```

3.6 pointInPolygon

```
// Test if a point is inside a convex polygon
↪  in O(lg n) time
typedef long long ll;
typedef pair <ll, ll> point;
#define x first
#define y second
struct segment {
  point P1, P2;
  segment () {}
  segment (point P1, point P2) : P1(P1), P2(P2)
  ↪  {}
};
inline ll ccw (point A, point B, point C) {
  return (B.x - A.x) * (C.y - A.y) - (C.x -
  ↪  A.x) * (B.y - A.y);
}
inline bool pointOnSegment (segment S, point P)
↪  {
  ll x = P.x, y = P.y, x1 = S.P1.x, y1 =
  ↪  S.P1.y, x2 = S.P2.x, y2 = S.P2.y;
  ll a = x - x1, b = y - y1, c = x2 - x1, d =
  ↪  y2 - y1, dot = a * c + b * d, len = c * c +
  ↪  d * d;
  if (x1 == x2 and y1 == y2) return x1 == x and
  ↪  y1 == y;
  if (dot < 0 or dot > len) return 0;
  return x1 * len + dot * c == x * len and y1 *
  ↪  len + dot * d == y * len;
}
const int M = 17;
const int N = 10010;
struct polygon {
  int n; // n > 1
  point p[N]; // clockwise order
  polygon () {}
  polygon (int _n, point *T) {
    n = _n;
    for (int i = 0; i < n; ++i) p[i] = T[i];
  }
  bool contains (point P, bool strictlyInside) {
    int lo = 1, hi = n - 1;
    while (lo < hi) {
      int mid = lo + hi >> 1;
      if (ccw(p[0], P, p[mid]) > 0) lo = mid +
  ↪  1;
      else hi = mid;
    }
    if (ccw(p[0], P, p[lo]) > 0) lo = 1;
    if (!strictlyInside and
  ↪  pointOnSegment(segment(p[0], p[n - 1]), P))
  ↪  return 1;
    if (!strictlyInside and
  ↪  pointOnSegment(segment(p[lo], p[lo - 1]),
  ↪  P)) return 1;
    if (lo == 1 or ccw(p[0], P, p[n - 1]) == 0)
  ↪  return 0;
    return ccw(p[lo], P, p[lo - 1]) < 0;
  }
};
point P;
polygon p;
```

3.7 tmp

```
const double EPS = 1e-9, pi = acos(-1.0);
//try to use point_i whenever possible
struct point_i {
 int x, y;
 point_i() {x = y = 0;}
 point_i(int _x, int _y): x(_x), y(_y) {}
};
struct point {
 double x, y;
 point() {x = y = 0.0;}
 point(double _x, double _y) : x(_x), y(_y) {}
 // operator overloading to sort the points
 bool operator < (point other) const {
  if (fabs(x - other.x) > EPS)
   return x < other.x;
  return y < other.y;
 }
 // to check if the points are equal
 bool operator == (point other) const {
  return (fabs(x - other.x) < EPS && (fabs(y -
  ↪  other.y) < EPS));
 }
};
double dist(point p1, point p2) {
 return (p1.x - p2.x) * (p1.x - p2.x) + (p1.y -
 ↪  p2.y) * (p1.y - p2.y);
}
double DEG_to_RAD(double theta) { return theta
↪  * pi / 180.0;}
// rotate a point by theta degree
point rotate (point p, double theta) { //theta
↪  is in degree
 double rad = DEG_to_RAD(theta);
 return point(p.x * cos(rad) - p.y * sin(rad),
              p.x * sin(rad) + p.y * cos(rad));
 //don't know how it works
}
struct line {
 double a, b, c;
 // ax + by + c = 0, but b = 1.0, so y = -ax -
 ↪  c,
};
line pointsToLine (point p1, point p2) {
 line l;
 if (fabs(p1.x - p2.x) < EPS) { // vertical line
  l.a = 1.0; l.b = 0.0 ; l.c = -p1.x;
 } else {
  l.a = -(double)(p1.y = p2.y) / (p1.x - p2.x);
  l.b = 1.0;
  l.c = -(double)(l.a * p1.x) - p1.y;
 }
}
bool areParallel(line l1, line l2) {
 return (fabs(l1.a - l2.a) < EPS) && (fabs(l1.b
 ↪  - l2.b) < EPS);
}
bool areSame(line l1, line l2) {
 return areParallel(l1, l2) && (fabs(l1.c -
 ↪  l2.c) < EPS);
}
// check areParallel before calling this
point areIntersect(line l1, line l2) {
 point p;
 p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a *
 ↪  l1.b - l1.a * l2.b);
 // test for vertical line
 if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x +
 ↪  l1.c);
 else p.y = -(l2.a * p.x + l2.c);
 return p;
}
struct vec {
 double x, y;
 vec(double _x, double _y) : x(_x), y(_y) {}
};
vec toVec(point a, point b) { // convert 2
 ↪  points to vector a->b
 return vec(b.x - a.x, b.y - a.y);
}
vec scale(vec v, double s) { // nonnegative s =
 ↪  [<1 .. 1 .. >1]
 // shorter.same.longer
 return vec(v.x * s, v.y * s);
}
point translate(point p, vec v) { // translate
 ↪  p according to v
 return point(p.x + v.x , p.y + v.y);
}
double dot(vec a, vec b) { return (a.x * b.x +
 ↪  a.y * b.y); }
double norm_sq(vec v) { return v.x * v.x + v.y
 ↪  * v.y; }
// returns the distance from p to the line
 ↪  defined by
// two points a and b (a and b must be
 ↪  different)
// the closest point is stored in the 4th
 ↪  parameter (byref)
double distToLine(point p, point a, point b,
 ↪  point &c) {
// formula: c = a + u * ab
 vec ap = toVec(a, p), ab = toVec(a, b);
 double u = dot(ap, ab) / norm_sq(ab);
 c = translate(a, scale(ab, u)); // translate a
 ↪  to c
```

```cpp
 return dist(p, c);
} // Euclidean distance between p and c
double angle(point a, point o, point b) { // returns angle aob in rad
 vec oa = toVec(o, a), ob = toVec(o, b);
 return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}
double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }
// note: to accept collinear points, we have to change the > 0
// returns true if point r is on the left side of line pq
//counter clock wise test
bool ccw(point p, point q, point r) {
 return cross(toVec(p, q), toVec(p, r)) > 0;
}
// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
 return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
}
// circles
int insideCircle(point_i p, point_i c, int r) { // all integer version
 int dx = p.x - c.x, dy = p.y - c.y;
 int Euc = dx * dx + dy * dy, rSq = r * r; // all integer
 return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;
}      //inside/border/outside
//inscribed circle or incircle radius
double area(double ab, double bc, double ca) {
 double s = (ab + bc + ca) / 2.0;
 return sqrt(s * (s - ab) * (s - bc) * (s - ca));
}
//returns radius of incircle
double rInCircle(double ab, double bc, double ca) {
 return area(ab, bc, ca) / (0.5 * (ab + bc + ca));
}
double rInCircle(point a, point b, point c) {
 return rInCircle(dist(a, b), dist(b, c), dist(c, a));
}
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
 r = rInCircle(p1, p2, p3);
 if (fabs(r) < EPS) return 0; // no inCircle center
 line l1, l2; // compute these two angle bisectors
 double ratio = dist(p1, p2) / dist(p1, p3);
 point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
 l1 = pointsToLine(p1, p);
 ratio = dist(p2, p1) / dist(p2, p3);
 p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
 l2 = pointsToLine(p2, p);
 ctr = areIntersect(l1, l2); // get their intersection point
 return 1;
}
//radius of circumcircle
double rCircumCircle(double ab, double bc, double ca) {
 return ab * bc * ca / (4.0 * area(ab, bc, ca));
}
double rCircumCircle(point a, point b, point c) {
 return rCircumCircle(dist(a, b), dist(b, c), dist(c, a));
}
//polygon
//vector P = set of all points of a polygon
// P[0] = P[n -1]
double perimeter(const vector<point> &P) {
 double result = 0.0;
 for (int i = 0; i < (int)P.size() - 1; i++) // remember that P[0] = P[n-1]
  result += dist(P[i], P[i + 1]);
 return result;
}
// returns the area, which is half the determinant
double area(const vector<point> &P) {
 double result = 0.0, x1, y1, x2, y2;
 for (int i = 0; i < (int)P.size() - 1; i++) {
  x1 = P[i].x; x2 = P[i + 1].x;
  y1 = P[i].y; y2 = P[i + 1].y;
  result += (x1 * y2 - x2 * y1);
 }
 return fabs(result) / 2.0;
}
// returns true if all three consecutive vertices of P form the same turns
bool isConvex(const vector<point> &P) {
 int sz = (int)P.size();
 if (sz <= 3) return false; // a point/sz=2 or a line/sz=3 is not convex
 bool isLeft = ccw(P[0], P[1], P[2]); // remember one result
 for (int i = 1; i < sz - 1; i++) // then compare with the others
  if (ccw(P[i], P[i + 1], P[(i + 2) == sz ? 1 : i + 2]) != isLeft)
   return false; // different sign -> this polygon is concave
 return true;
}
// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
 if ((int)P.size() == 0) return false;
 double sum = 0; // assume the first vertex is equal to the last vertex
 for (int i = 0; i < (int)P.size() - 1; i++) {
  if (ccw(pt, P[i], P[i + 1]))
   sum += angle(P[i], pt, P[i + 1]); // left turn/ccw
  else sum -= angle(P[i], pt, P[i + 1]);
 } // right turn/cw
 return fabs(fabs(sum) - 2 * pi) < EPS;
}
// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A, point B) {
 double a = B.y - A.y;
 double b = A.x - B.x;
 double c = B.x * A.y - A.x * B.y;
 double u = fabs(a * p.x + b * p.y + c);
 double v = fabs(a * q.x + b * q.y + c);
 return point((p.x * v + q.x * u) / (u + v), (p.y * v + q.y * u) / (u + v));
}
// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
 vector<point> P;
 for (int i = 0; i < (int)Q.size(); i++) {
  double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
  if (i != (int)Q.size() - 1) left2 = cross(toVec(a, b), toVec(a, Q[i + 1]));
  if (left1 > -EPS) P.push_back(Q[i]); // Q[i] is on the left of ab
  if (left1 * left2 < -EPS) // edge (Q[i], Q[i+1]) crosses line ab
   P.push_back(lineIntersectSeg(Q[i], Q[i + 1], a, b));
 }
 if (!P.empty() && !(P.back() == P.front()))
  P.push_back(P.front()); // make Ps first point = Ps last point
 return P;
}
// convex hull
point pivot(0, 0);
bool angleCmp(point a, point b) { // angle-sorting function
 if (collinear(pivot, a, b)) // special case
```

```
  return dist(pivot, a) < dist(pivot, b); //
   ↪ check which one is closer
 double d1x = a.x - pivot.x, d1y = a.y -
   ↪ pivot.y;
 double d2x = b.x - pivot.x, d2y = b.y -
   ↪ pivot.y;
 return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;
} // compare two angles
vector<point> CH(vector<point> P) { // the
   ↪ content of P may be reshuffled
 int i, j, n = (int)P.size();
 if (n <= 3) {
  if (!(P[0] == P[n - 1])) P.push_back(P[0]);
   ↪ // safeguard from corner case
  return P;
 } // special case, the CH is P itself
// first, find P0 = point with lowest Y and if
   ↪ tie: rightmost X
 int P0 = 0;
 for (i = 1; i < n; i++)
  if (P[i].y < P[P0].y || (P[i].y == P[P0].y &&
   ↪ P[i].x > P[P0].x))
   P0 = i;
 point temp = P[0]; P[0] = P[P0]; P[P0] = temp;
   ↪ // swap P[P0] with P[0]
// second, sort points by angle w.r.t. pivot P0
 pivot = P[0]; // use this global variable as
   ↪ reference
 sort(++P.begin(), P.end(), angleCmp); // we do
   ↪ not sort P[0]
 // third, the ccw tests
 vector<point> S;
 S.push_back(P[n - 1]); S.push_back(P[0]);
   ↪ S.push_back(P[1]); // initial S
 i = 2; // then, we check the rest
 while (i < n) { // note: N must be >= 3 for
   ↪ this method to work
  j = (int)S.size() - 1;
  if (ccw(S[j - 1], S[j], P[i]))
   ↪ S.push_back(P[i++]); // left turn, accept
  else S.pop_back();
 } // or pop the top of S until we have a left
   ↪ turn
 return S; // return the result
}
```

4 Graph

4.1 Articulation Point Detection

```
vector<int> adj[N];
bool vis[N], articulation[N];
int low[N], tin[N], taim;
void dfs(int node, int par = -1) {
 vis[node] = 1;
 tin[node] = low[node] = taim++;
 int children = 0;
 for (int x : adj[node]) {
  if (x == par) continue;
  if (vis[x]) low[node] = min(low[node],
   ↪ tin[x]);
  else {
   dfs(x, node);
   low[node] = min(low[node], low[x]);
   if (low[x] >= tin[node] && par != -1) {
    articulation[node] = 1;
   }
   children++;
  }
 }
 if (children > 1 and par == -1)
   ↪ articulation[node] = 1;
}
```

4.2 BlockCutTree

```
const int N = 300010;
bitset <N> art, good;
vector <int> g[N], tree[N], st, comp[N];
int n, m, ptr, cur, in[N], low[N], id[N];
void dfs (int u, int from = -1) {
  in[u] = low[u] = ++ptr;
  st.emplace_back(u);
  for (int v : g[u]) if (v ^ from) {
      if (!in[v]) {
        dfs(v, u);
        low[u] = min(low[u], low[v]);
        if (low[v] >= in[u]) {
          art[u] = in[u] > 1 or in[v] > 2;
          comp[++cur].emplace_back(u);
          while (comp[cur].back() ^ v) {
            comp[cur].emplace_back(st.back());
            st.pop_back();
          }
        }
      } else {
        low[u] = min(low[u], in[v]);
      }
    }
}
void buildTree() {
  ptr = 0;
  for (int i = 1; i <= n; ++i) {
    if (art[i]) id[i] = ++ptr;
  }
  for (int i = 1; i <= cur; ++i) {
    int x = ++ptr;
    for (int u : comp[i]) {
      if (art[u]) {
        tree[x].emplace_back(id[u]);
        tree[id[u]].emplace_back(x);
      } else {
        id[u] = x;
      }
    }
  }
}
int main() {
  cin >> n >> m;
  while (m--) {
    int u, v;
    scanf("%d %d", &u, &v);
    g[u].emplace_back(v);
    g[v].emplace_back(u);
  }
  for (int i = 1; i <= n; ++i)
    if (!in[i]) dfs(i);
  buildTree();
}
```

4.3 Bridge Detection

```
vector<int> adj[N];
bool visited[N];
int low[N], tin[N], timer;
vector<pair<int, int>> bridges;
void IS_BRIDGE(int a, int b) {
 bridges.push_back({min(a, b), max(a, b)});
}
void dfs(int v, int p = -1) {
 visited[v] = true;
 tin[v] = low[v] = timer++;
 for (int to : adj[v]) {
  if (to == p) continue;
  if (visited[to]) low[v] = min(low[v],
   ↪ tin[to]);
  else {
   dfs(to, v);
   low[v] = min(low[v], low[to]);
   if (low[to] > tin[v]) IS_BRIDGE(v, to);
  }
 }
}
```

4.4 BridgeTree

```
const int N = 300005;
vector<pair<int, int> > edge[N];  // {adjacent
   ↪ edge, index}
vector<int> dfsTime(N), low(N), comp(N),
   ↪ bridgeTree[N];
vector<bool> vis(N, 0), isBridge(N, 0);
int timer = 0;
//to find bridges
void dfs(int u, int par) {
 vis[u] = 1;
 dfsTime[u] = low[u] = ++timer;
 for (int i = 0; i < edge[u].size(); i++) {
  int v = edge[u][i].first, ind =
   ↪ edge[u][i].second;
  if (v == par)
   continue; // don't visit the parent node
  if (vis[v]) {  // cross edge
   low[u] = min(low[u], dfsTime[v]);
```

```
  } else {
   dfs(v, u);
   low[u] = min(low[u], low[v]);
   if (low[v] > dfsTime[u]) {  // checking
    among the back edges
    // u->v is a bridge
    isBridge[ind] = 1;
   }
  }
 }
}
// to assign unique component number to each
   component and its children
void dfs2(int u, int comp_number) {
 vis[u] = 1;
 comp[u] = comp_number;
 for (int i = 0; i < edge[u].size(); i++) {
  int v = edge[u][i].first, ind =
    edge[u][i].second;
  if (!vis[v] && !isBridge[ind])
   dfs2(v, comp_number);
 }
}
void make_bridge_tree(int n) {
 // assign unique component number to each
    component
 vis.assign(n + 1, 0);
 comp.assign(n + 1, -1);
 int comp_number = 1;
 for (int i = 1; i <= n; i++) {
  if (!vis[i]) {
   dfs2(i, i);
   //i will be the root of its component
  }
 }
 for (int i = 0; i <= n;
    i++)bridgeTree[i].clear();
 //creating bridge tree
 for (int i = 1; i <= n; i++) {
  for (int j = 0; j < edge[i].size(); j++) {
   int v = edge[i][j].first;
   if (comp[i] != comp[v]) {
    bridgeTree[comp[i]].push_back(comp[v]);
    bridgeTree[comp[v]].push_back(comp[i]);
   }
  }
 }
}
void find_bridges(int n) {
 timer = 0;
 vis.assign(n + 1, 0);
 low.assign(n + 1, -1);
 dfsTime.assign(n + 1, -1);
 for (int i = 1; i <= n; i++)
  if (!vis[i])dfs(i, -1);
}
```

4.5 Centroid Decomposition

```
// Builds a centroid tree of height O(logn) in
   O(nlogn).
const int M = 2e5 + 3;
int sz[M], done[M], cpar[M], root;
vector<int>ctree[M];
void go(int u, int p = -1) {
 sz[u] = 1;
 for (int v : g[u]) {
  if (v == p or done[v]) continue;
  go(v, u);
  sz[u] += sz[v];
 }
}
int find_centroid(int v, int p, int n) {
 for (int x : g[v]) {
  if (x != p and !done[x] and sz[x] > n / 2)
    return find_centroid(x, v, n);
 }
 return v;
}
void decompose(int v = 0, int p = -1) {
 go(v);
 int c = find_centroid(v, -1, sz[v]);
 if (p == -1) root = c;
 done[c] = 1;
 cpar[c] = p;
 if (p != -1) ctree[p].push_back(c);
 for (int x : g[c]) {
  if (!done[x]) decompose(x, c);
 }
}
```

4.6 DSU on Tree

```
vector <int> G[mx]; /// adjacency list of the
    tree
int sub[mx]; /// subtree size of a node
int color[mx]; /// color of a node
int freq[mx];
int n;
void calcSubSize(int s, int p) {
 sub[s] = 1;
 for (int x : G[s]) {
  if (x == p) continue;
  calcSubSize(x, s);
  sub[s] += sub[x];
 }
}
void add(int s, int p, int v, int bigchild =
    -1) {
 freq[color[s]] += v;
 for (int x : G[s]) {
  if (x == p || x == bigchild) continue;
  add(x, s, v);
 }
}
void dfs(int s, int p, bool keep) {
 int bigChild = -1;
 for (int x : G[s]) {
  if (x == p) continue;
  if (bigChild == -1 || sub[bigChild] < sub[x]
    ) bigChild = x;
 }
 for (int x : G[s]) {
  if (x == p || x == bigChild) continue;
  dfs(x, s, 0);
 }
 if (bigChild != -1) dfs(bigChild, s, 1);
 add(s, p, 1, bigChild);
/// freq[c] now contains the number of nodes in
/// the subtree of 'node' that have color c
/// Save the answer for the queries here
 if (keep == 0) add(s, p, -1);
}
int main() {
 input color
 construct G
 calcSubSize(root, -1);
 dfs(root, -1, 0);
 return 0;
}
```

4.7 Dijkstra

```
#define pii pair<long long,int>
vector<int>Edges[100005];
vector<long long>Cost[100005];
long long dis[100005];
int vis[100005];
void dijkstra( int source ) {
 priority_queue< pii, vector<pii>, greater<pii>
   >Q;
 Q.push( pii(0, source) );
 dis[source] = 0;
 pii q;
 while ( !Q.empty() ) {
  q = Q.top();
  Q.pop();
  int u = q.second;
  if ( vis[u] != -1 ) continue; // *idk why*
  vis[u] = 1; // *idk why*
  for ( int i = 0; i < Edges[u].size(); i++ ) {
   int v = Edges[u][i];
   if ( vis[v] != -1 ) continue; // *idk why*
   if ( dis[u] + Cost[u][i] < dis[ v ] ) {
    dis[ v ] = dis[u] + Cost[u][i];
    Q.push( pii( dis[v], v ) );
   }
  }
 }
 return;
}
```

4.8 Dinic

```
// O(V^2 E), solves SPOJ FASTFLOW
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
struct edge {
 int u, v;
 ll cap, flow;
 edge () {}
 edge (int u, int v, ll cap) : u(u), v(v),
   cap(cap), flow(0) {}
};
struct Dinic {
 int N;
 vector <edge> E;
 vector <vector <int>> g;
 vector <int> d, pt;
 Dinic (int N) : N(N), E(0), g(N), d(N), pt(N)
   {}
 void AddEdge (int u, int v, ll cap) {
  if (u ^ v) {
   E.emplace_back(u, v, cap);
   g[u].emplace_back(E.size() - 1);
   E.emplace_back(v, u, 0);
   g[v].emplace_back(E.size() - 1);
  }
 }
 bool BFS (int S, int T) {
  queue <int> q({S});
  fill(d.begin(), d.end(), N + 1);
  d[S] = 0;
  while (!q.empty()) {
   int u = q.front(); q.pop();
   if (u == T) break;
   for (int k : g[u]) {
    edge &e = E[k];
    if (e.flow < e.cap and d[e.v] > d[e.u] + 1)
    {
     d[e.v] = d[e.u] + 1;
     q.emplace(e.v);
    }
   }
  } return d[T] != N + 1;
 }
 ll DFS (int u, int T, ll flow = -1) {
  if (u == T or flow == 0) return flow;
  for (int &i = pt[u]; i < g[u].size(); ++i) {
   edge &e = E[g[u][i]];
   edge &oe = E[g[u][i] ^ 1];
   if (d[e.v] == d[e.u] + 1) {
    ll amt = e.cap - e.flow;
    if (flow != -1 and amt > flow) amt = flow;
    if (ll pushed = DFS(e.v, T, amt)) {
     e.flow += pushed;
     oe.flow -= pushed;
     return pushed;
    }
   }
  } return 0;
 }
 ll MaxFlow (int S, int T) {
  ll total = 0;
  while (BFS(S, T)) {
   fill(pt.begin(), pt.end(), 0);
   while (ll flow = DFS(S, T)) total += flow;
  }
  return total;
 }
};
int main() {
 int N, E;
 scanf("%d %d", &N, &E);
 Dinic dinic(N);
 for (int i = 0, u, v; i < E; ++i) {
  ll cap;
  scanf("%d %d %lld", &u, &v, &cap);
  dinic.AddEdge(u - 1, v - 1, cap);
  dinic.AddEdge(v - 1, u - 1, cap);
 }
 printf("%lld\n", dinic.MaxFlow(0, N - 1));
 return 0;
}
```

4.9 HeavyLight Decomposition

```
vector < int > List[ ?? ]; // Tree's Adj List
  -> Need to Clear ??
class HeavyLightDecomposition
{
#define L_R ? ?
public :
 vector<int> ValueOfNode;
 vector<int> Position;
 vector<int> Parent;
 vector<int> Depth;
 vector<int> Heavy;
 vector<int> Head;
 int CurrentPosition = 1; // 0/1 - index based
 segmentTree ST = segmentTree( ?? ) /
   AnyQueryTree;
 HeavyLightDecomposition(int NN) {
  ValueOfNode.resize(NN);
  Position.resize(NN);
  Parent.resize(NN, -1);
  Depth.resize(NN, 0);
  Heavy.resize(NN, -1);
  Head.resize(NN);
 }
 int DFS(int Vertex) {
  int TotalSize = 1;
  int MaxChildSize = 0;
  for (int i = 0; i < List[Vertex].size(); ++i)
   {
   int Child = List[Vertex][i];
   if (Child != Parent[Vertex]) {
    Parent[Child] = Vertex;
    Depth[Child] = Depth[Vertex] + 1;
    int ChildSize = DFS(Child);
    TotalSize += ChildSize;
    if (ChildSize > MaxChildSize) {
     MaxChildSize = ChildSize;
     Heavy[Vertex] = Child;
    }
   }
  }
  return TotalSize;
 }
 void TreeDecompose(int Vertex, int Hd) {
  Head[Vertex] = Hd;
  ST.A[CurrentPosition] = ValueOfNode[Vertex];
  Position[Vertex] = CurrentPosition++;
  if (Heavy[Vertex] != -1)
   TreeDecompose(Heavy[Vertex], Hd);
  for (int i = 0; i < List[Vertex].size(); ++i)
  {
   int Child = List[Vertex][i];
   if (Child != Parent[Vertex] && Child !=
   Heavy[Vertex])
    TreeDecompose(Child, Child);
  }
 }
 void MakeQueryTree() { // ?? = Number of Node
   in Tree;
// Build Query Data Structure ? ?
 }
 int Query(int NodeA, int NodeB) {
  int Res = 0;
  while (Head[NodeA] != Head[NodeB]) {
   if (Depth[Head[NodeA]] > Depth[Head[NodeB]])
    swap(NodeA, NodeB);
   int CurrentPathResult =
     ST.rangeQuery(L_R, Position[Head[NodeB]],
   Position[NodeB]).Value;
   Res = ? ? (Res, CurrentPathResult);
   NodeB = Parent[Head[NodeB]];
  }
  if (Depth[NodeA] > Depth[NodeB])
   swap(NodeA, NodeB);
  int LastHeavyPathResult =
    ST.rangeQuery(L_R, Position[NodeA],
   Position[NodeB]).Value;
  Res = ? ? (Res, LastHeavyPathResult);
  return Res;
 }
 int Update(int NodeA, int NodeB, int X) {
  while (Head[NodeA] != Head[NodeB]) {
   if (Depth[Head[NodeA]] > Depth[Head[NodeB]])
    swap(NodeA, NodeB);
   ST.rangeUpdate(L_R, Position[Head[NodeB]],
   Position[NodeB], X);
   NodeB = Parent[Head[NodeB]];
  }
  if (Depth[NodeA] > Depth[NodeB])
   swap(NodeA, NodeB);
```

```
  ST.rangeUpdate(L_R, Position[NodeA],
    Position[NodeB], X);
 }
};
```

4.10 Hopcroft Karp

```
// Maximum biparite matching. Complexity :
    O(E*sqrt(V))
//define NIL (dummy vertex), M and INF
vector<int>g[M];
int Lmatch[M], Rmatch[M], dist[M];
bool bfs(int n) {
 queue<int>q;
 for (int u = 1; u <= n; u++) {
  if (Lmatch[u] == NIL) dist[u] = 0, q.push(u);
  else dist[u] = INF;
 }
 dist[NIL] = INF;
 while (!q.empty()) {
  int u = q.front();
  q.pop();
  if (dist[u] < dist[NIL]) {
   for (int v : g[u]) {
    if (dist[Rmatch[v]] == INF) {
     dist[Rmatch[v]] = dist[u] + 1;
     q.push(Rmatch[v]);
    }
   }
  }
 }
 return dist[NIL] != INF;
}
bool dfs(int u) {
 if (u == NIL) return true;
 for (int v : g[u]) {
  if (dist[Rmatch[v]] == dist[u] + 1 and
    dfs(Rmatch[v])) {
   Rmatch[v] = u;
   Lmatch[u] = v;
   return true;
  }
 }
 dist[u] = INF;
 return false;
}
int HopcroftKarp(int n, int m) {
 fill(Lmatch, Lmatch + n + 1, 0);
 fill(Rmatch, Rmatch + m + 1, 0);
 int res = 0;
 while (bfs(n)) {
  for (int u = 1; u <= n; u++) {
   if (Lmatch[u] == NIL and dfs(u)) res++;
  }
 }
 return res;
}
```

4.11 Hungarian

```
/*returns maximum/minimum weighted bipartite
, matching. Complexity : O(N^2 * M)
flag = -1 minimizes, flag = 1 maximizes. */
#define CLR(a) memset(a, 0, sizeof a)
ll weight[N][M];
int used[M], P[M], way[M], match[M];
ll U[M], V[M], minv[M], ara[N][M];
ll hungarian(int n, int m, int flag) {
 CLR(U), CLR(V), CLR(P), CLR(ara), CLR(way);
 for (int i = 1; i <= n; i++) {
  for (int j = 1; j <= m; j++) {
   ara[i][j] = -flag * weight[i][j];
  }
 }
 if (n > m) m = n;
 int a, b, d;
 ll r, w;
 for (int i = 1; i <= n; i++) {
  P[0] = i, b = 0;
  for (int j = 0; j <= m; j++) minv[j] = INF,
    used[j] = false;
  do {
   used[b] = true;
   a = P[b], d = 0, w = INF;
   for (int j = 1; j <= m; j++) {
    if (!used[j]) {
     r = ara[a][j] - U[a] - V[j];
     if (r < minv[j]) minv[j] = r, way[j] = b;
     if (minv[j] < w) w = minv[j], d = j;
    }
   }
   for (int j = 0; j <= m; j++) {
    if (used[j]) U[P[j]] += w, V[j] -= w;
    else minv[j] -= w;
   }
   b = d;
  } while (P[b] != 0);
  do {
   d = way[b];
   P[b] = P[d], b = d;
  } while (b != 0);
 }
 for (int j = 1; j <= m; j++) match[P[j]] = j;
 return flag * V[0];
}
```

4.12 LCA(sparse table)

```
vector<int>Edges[10000];
int p[10005][17], level[10005], n, lg;
bool vis[10005];
void DFS( int par, int node ) {
 vis[node] = 1;
 if ( par != -1 ) level[node] = level[par] + 1;
 p[node][0] = par;
 for ( int i = 1; i <= lg; i++ ) {
  if ( p[node][i - 1] != -1 ) p[node][i] =
    p[p[node][i - 1] ][i - 1];
 }
 for ( int i = 0; i < Edges[node].size(); i++ )
   {
  if ( vis[ Edges[node][i] ] == 0 ) DFS( node,
    Edges[node][i] );
 }
 return;
}
int LCA( int u, int v ) {
 if ( level[u] < level[v] ) swap(u, v);
 for ( int i = lg; i >= 0; i-- ) {
  int par = p[u][i];
  if ( level[par] >= level[v] ) {
   u = par;
  }
 }
 if ( u == v ) return u;
 for ( int i = lg; i >= 0; i-- ) {
  int U = p[u][i];
  int V = p[v][i];
  if ( U != V ) {
   u = U; v = V;
  }
 }
 return p[u][0];
}
```

4.13 Max Flow Edmond Karp

```
int n;
vector<vector<int>> capacity;
vector<vector<int>> adj;
int bfs(int s, int t, vector<int> &parent) {
 fill(parent.begin(), parent.end(), -1);
 parent[s] = -2;
 queue<pair<int, int>> q;
 q.push({s, INF});
 while (!q.empty()) {
  int cur = q.front().first;
  int flow = q.front().second;
  q.pop();
  for (int next : adj[cur]) {
   if (parent[next] == -1 &&
    capacity[cur][next]) {
    parent[next] = cur;
    int new_flow = min(flow,
    capacity[cur][next]);
    if (next == t) return new_flow;
    q.push({next, new_flow});
   }
  }
 }
 return 0;
}
int maxflow(int s, int t) {
 int flow = 0;
```

```
 vector<int> parent(n);
 int new_flow;
 while (new_flow = bfs(s, t, parent)) {
  flow += new_flow;
  int cur = t;
  while (cur != s) {
   int prev = parent[cur];
   capacity[prev][cur] -= new_flow;
   capacity[cur][prev] += new_flow;
   cur = prev;
  }
 }
 return flow;
}
```

4.14 Max Flow-1

```
int graph[105][105];
int rgraph[105][105];
int par[105];
int n;
int bfs( int s, int d ) {
 bool vis[105];
 memset( vis, 0, sizeof(vis) );
 queue<int>Q;
 Q.push(s);
 while ( !Q.empty() ) {
  int q = Q.front();
  Q.pop();
  for ( int i = 1; i <= n; i++ ) {
   if ( vis[i] == 0 && rgraph[q][i] > 0 ) {
    vis[i] = 1;
    par[i] = q;
    if ( i == d ) return 1;
    Q.push(i);
   }
  }
 }
 return 0;
}
int max_flow( int s, int d ) {
 int total_flow = 0;
 for ( int i = 1; i <= n; i++ ) {
  for ( int j = 1; j <= n; j++ ) rgraph[i][j] =
   graph[i][j];
 }
 int mn;
 while ( bfs( s, d ) == 1 ) {
  mn = INT_MAX;
  for ( int child = d; child != s; child =
   par[child] ) {
   int P = par[child];
   mn = min(mn, rgraph[P][child] );
  }
  for ( int child = d; child != s; child =
   par[child] ) {
   int P = par[child];
   rgraph[P][child] -= mn;
   rgraph[child][P] += mn;
  }
  total_flow += mn;
 }
 return total_flow;
}
```

4.15 Online Bridge

```
vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
int bridges;
int lca_iteration;
vector<int> last_visit;
void init(int n) {
 par.resize(n);
 dsu_2ecc.resize(n);
 dsu_cc.resize(n);
 dsu_cc_size.resize(n);
 lca_iteration = 0;
 last_visit.assign(n, 0);
 for (int i = 0; i < n; ++i) {
  dsu_2ecc[i] = i;
  dsu_cc[i] = i;
  dsu_cc_size[i] = 1;
  par[i] = -1;
 }
 bridges = 0;
}
int find_2ecc(int v) {
 if (v == -1)
  return -1;
 return dsu_2ecc[v] == v ? v : dsu_2ecc[v] =
    find_2ecc(dsu_2ecc[v]);
}
int find_cc(int v) {
 v = find_2ecc(v);
 return dsu_cc[v] == v ? v : dsu_cc[v] =
    find_cc(dsu_cc[v]);
}
void make_root(int v) {
 v = find_2ecc(v);
 int root = v;
 int child = -1;
 while (v != -1) {
  int p = find_2ecc(par[v]);
  par[v] = child;
  dsu_cc[v] = root;
  child = v;
  v = p;
 }
 dsu_cc_size[root] = dsu_cc_size[child];
}
void merge_path (int a, int b) {
 ++lca_iteration;
 vector<int> path_a, path_b;
 int lca = -1;
 while (lca == -1) {
  if (a != -1) {
   a = find_2ecc(a);
   path_a.push_back(a);
   if (last_visit[a] == lca_iteration) {
    lca = a;
    break;
   }
   last_visit[a] = lca_iteration;
   a = par[a];
  }
  if (b != -1) {
   b = find_2ecc(b);
   path_b.push_back(b);
   if (last_visit[b] == lca_iteration) {
    lca = b;
    break;
   }
   last_visit[b] = lca_iteration;
   b = par[b];
  }
 }
 for (int v : path_a) {
  dsu_2ecc[v] = lca;
  if (v == lca)
   break;
  --bridges;
 }
 for (int v : path_b) {
  dsu_2ecc[v] = lca;
  if (v == lca)
   break;
  --bridges;
 }
}
void add_edge(int a, int b) {
 a = find_2ecc(a);
 b = find_2ecc(b);
 if (a == b) return;
 int ca = find_cc(a);
 int cb = find_cc(b);
 if (ca != cb) {
  ++bridges;
  if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
   swap(a, b);
   swap(ca, cb);
  }
  make_root(a);
  par[a] = dsu_cc[a] = b;
  dsu_cc_size[cb] += dsu_cc_size[a];
 } else {
  merge_path(a, b);
 }
}
```

4.16 SCC

```
/*In a directed graph, an SCC is a connected
component where all nodes are pairwise
reachable. ,
```

```
condesation graph is the DAG built on a directed
, graph by compressing each SCC into a node.
define M */
vector<int>g[M], gr[M];
set<int>gc[M];
int vis[M], id[M], sz[M];
vector<int>order, comp, roots;
namespace SCC {
void addEdge(int u, int v) {
 g[u].push_back(v), gr[v].push_back(u);
}
void dfs1(int u) {
 vis[u] = 1;
 for (int x : g[u])
  if (!vis[x]) dfs1(x);
 order.push_back(u);
}
void dfs2(int u) {
 vis[u] = 1;
 comp.push_back(u);
 for (int x : gr[u])
  if (!vis[x]) dfs2(x);
}
void condense(int n) {
 fill(vis, vis + n + 1, 0);
 for (int i = 1; i <= n; i++)
  if (!vis[i]) dfs1(i);
 reverse(order.begin(), order.end());
 fill(vis, vis + n + 1, 0);
 for (int u : order) {
  if (!vis[u]) {
   dfs2(u); //this part of the code processes
    components, returns them in comp
   for (int v : comp) id[v] = u;
   sz[u] = (int)comp.size();
   roots.push_back(u);
   comp.clear();
  }
 }
 fill(vis, vis + n + 1, 0);
 for (int u = 1; u <= n; u++) {
  for (int v : g[u]) {
   if (id[u] != id[v])
    gc[id[u]].insert(id[v]);
  }
 }
}
void reset(int n) {
 order.clear(), comp.clear(), roots.clear();
 for (int i = 1; i <= n; i++) {
  g[i].clear(), gr[i].clear(), gc[i].clear();
  id[i] = vis[i] = sz[i] = 0;
 }
}
}
```

4.17 centroid_root_decomposition

```
const int MAXN = 100050;
const int LOGN = 17;
int par[LOGN][MAXN];   // par[i][v]: (2^i)th
    ancestor of v
int level[MAXN], sub[MAXN];  // sub[v]: size of
    subtree whose root is v
int ctPar[MAXN], n; // ctPar[v]: parent of v in
    centroid tree
vector<int> adj[MAXN];
bool vis[MAXN];
int ans[MAXN];  // ans[v]: shortest distance
    between v and red nodes in subtree
    corresponding to centroid v
long long INF = 1e18;
// calculate level by dfs
void dfsLevel(int node, int pnode) {
 for(auto cnode : adj[node]) {
  if(cnode != pnode) {
   par[0][cnode] = node;
   level[cnode] = level[node] + 1;
   dfsLevel(cnode, node);
  }
 }
}
void preprocess() {
 level[0] = 0; par[0][0] = 0;
 dfsLevel(0, -1);
 for(int i = 1; i < LOGN; i++)
  for(int node = 0; node < n; node++)
   par[i][node] = par[i-1][par[i-1][node]];
}
int lca(int u, int v) {
 if(level[u] > level[v]) swap(u, v);
 int d = level[v] - level[u];
 // make u, v same level
 for(int i = 0; i < LOGN; i++) {
  if(d & (1 << i)) {
   v = par[i][v];
  }
 }
 if(u == v) return u;
 // find LCA
 for(int i = LOGN - 1; i >= 0; i--) {
  if(par[i][u] != par[i][v]) {
   u = par[i][u];
   v = par[i][v];
  }
 }
 return par[0][u];
}
int dist(int u, int v) {
 return level[u] + level[v] - 2 * level[lca(u,
    v)];
}
/* Centroid decomposition */
// Calculate size of subtrees by dfs
void dfsSubtree(int node, int pnode) {
 sub[node] = 1;
 for(auto cnode : adj[node]) {
  if(cnode != pnode && vis[cnode] == 0) {
   dfsSubtree(cnode, node);
   sub[node] += sub[cnode];
  }
 }
}
// find Centroid
int dfsCentroid(int node, int pnode, int size) {
 for(auto cnode : adj[node]) {
  if(cnode != pnode && sub[cnode] > size / 2 &&
    vis[cnode] == 0 )
   return dfsCentroid(cnode, node, size);
 }
 return node;
}
// Centroid decomposition
void decompose(int node, int pCtr) {
 dfsSubtree(node, -1);
 int ctr = dfsCentroid(node, node, sub[node]);
 vis[ctr] = 1;
 if(pCtr == -1)
  pCtr = ctr; // root of centroid tree
 ctPar[ctr] = pCtr;

 for(auto cnode : adj[ctr]) {
  if( vis[cnode] == 0 ) decompose(cnode, ctr);
 }
 adj[ctr].clear();
}
// color node v red
void update(int v) {
 int rNode = v;
 while(1) {
  ans[v] = min(ans[v], dist(rNode, v));
  if(v == ctPar[v]) break;
  v = ctPar[v];
 }
}
// reply query
int query(int v) {
 int start = v;
 int minD = INF;
 while(1) {
  minD = min(minD, dist(start, v) + ans[v]);
  if(v == ctPar[v]) break;
  v = ctPar[v];
 }
 return minD;
}
int main() {
 preprocess();
 decompose(0, -1);
 fill(ans, ans + n, INF);
 update(0);
}
```

5 Grundy

5.1 grundy

```
int grundyValue[ Nn ];
int mexValue[ Nn ], MEX;
void calculateGrundyValue() {
      grundyValue[ 1 ] = grundyValue[ 2 ] = 0;
      for ( int i = 3; i <= 10000; ++i ) {
            for ( int j = 1; j + j < i; ++j )
                  mexValue[ grundyValue[ j ] ^ grundyValue[ i - j ] ] = i;
            MEX = 0;
            while ( mexValue[ MEX ] == i )
                  MEX++;
            grundyValue[ i ] = MEX;
      }
}
void solve( int t ) {
      int N, a;
      cin >> N;
      int XOR = 0;
      while ( N-- ) {
            cin >> a;
            XOR ^= grundyValue[ a ];
      }
      cout << "Case " << t << ": ";
      if ( XOR )   cout << "Alice\n";
}
```

6 Math

6.1 Matrices

6.1.1 Gauss-Jordan Elimination in GF(2)

```
const int SZ = 105;
const int MOD = 1e9 + 7;
bitset <SZ> mat[SZ];
int where[SZ];
bitset <SZ> ans;
ll bigMod(ll a, ll b, ll m) {
 ll ret = 1LL;
 a %= m;
 while (b) {
  if (b & 1LL) ret = (ret * a) % m;
  a = (a * a) % m;
  b >>= 1LL;
 }
 return ret;
}
/// n for row, m for column, modulo 2
int GaussJordan(int n, int m) {
 SET(where); /// sets to -1
 for (int r = 0, c = 0; c < m && r < n; c++) {
  for (int i = r; i < n; i++)
   if ( mat[i][c] ) {
    swap(mat[i], mat[r]); break;
   }
  if ( !mat[r][c] ) continue;
  where[c] = r;
  for (int i = 0; i < n; ++i) if (i != r && mat[i][c]) mat[i] ^= mat[r];
  r++;
 }
 for (int j = 0; j < m; j++) {
  if (where[j] != -1) ans[j] = mat[where[j]][m] / mat[where[j]][j];
  else ans[j] = 0;
 }
 for (int i = 0; i < n; i++) {
  int sum = 0;
  for (int j = 0; j < m; j++) sum ^= (ans[j] & mat[i][j]);
  if ( sum != mat[i][m] ) return 0; /// no solution
 }
 int cnt = 0;
 for (int j = 0; j < m; j++) if (where[j] == -1) cnt++;
 return bigMod(2, cnt, MOD); /// how many solutions modulo some other MOD
}
```

6.1.2 Gauss-Jordan Elimination in GF(P)

```
const int SZ = 105;
const int MOD = 1e9 + 7;
int mat[SZ][SZ], where[SZ], ans[SZ];
ll bigMod(ll a, ll b, ll m) {
 ll ret = 1LL;
 a %= m;
 while (b) {
  if (b & 1LL) ret = (ret * a) % m;
  a = (a * a) % m;
  b >>= 1LL;
 }
 return ret;
}
int GaussJordan(int n, int m, int P) {
 SET(where); /// sets to -1
 for (int r = 0, c = 0; c < m && r < n; c++) {
  int mx = r;
  for (int i = r; i < n; i++) if ( mat[i][c] > mat[mx][c] ) mx = i;
  if ( mat[mx][c] == 0 ) continue;
  if (r != mx) for (int j = c; j <= m; j++) swap(mat[r][j], mat[mx][j]);
  where[c] = r;
  int mul, minv = bigMod(mat[r][c], P - 2, P);
  int temp;
  for (int i = 0; i < n; i++) {
   if ( i != r && mat[i][c] != 0) {
    mul = ( mat[i][c] * (long long) minv ) % P;
    for (int j = c; j <= m; j++) {
     temp = mat[i][j];
     temp -= ( ( mul * (long long) mat[r][j] ) % P );
     temp += P;
     if ( temp >= P ) temp -= P;
     mat[i][j] = temp;
    }
   }
  }
  r++;
 }
 for (int j = 0; j < m; j++) {
  if (where[j] != -1) ans[j] = (mat[where[j]][m] * 1LL * bigMod(mat[where[j]][j], P - 2, P) ) % P;
  else ans[j] = 0;
 }
 for (int i = 0; i < n; i++) {
  int sum = 0;
  for (int j = 0; j < m; j++) {
   sum += ( ans[j] * 1LL * mat[i][j] ) % P;
   if (sum >= P) sum -= P;
  }
  if ( sum != mat[i][m] ) return 0; /// no solution
 }
 int cnt = 0;
 for (int j = 0; j < m; j++) if (where[j] == -1) cnt++;
 return bigMod(P, cnt, MOD);
}
```

6.1.3 Gauss-Jordan Elimination

```
/*** mat is 0 based
* In every test case, clear mat first and then do the changes
* For solving problems on graphs with probability/expectation, make sure the graph
is connected and a single component. If not, then re-number the vertex and solve
for each connected component separately.
* Complexity --> O( min(n,m) * nm ) **/
const int SZ = 105;
const double EPS = 1e-9;
double mat[SZ][SZ], ans[SZ];
int where[SZ];
int GaussJordan(int n, int m) {
 SET(where); /// sets to -1
 for (int r = 0, c = 0; c < m && r < n; c++) {
  int mx = r;
  for (int i = r; i < n; i++) if ( abs(mat[i][c]) > abs(mat[mx][c]) ) mx = i;
  if ( abs(mat[mx][c]) < EPS ) continue;
  if (r != mx) for (int j = c; j <= m; j++) swap(mat[r][j], mat[mx][j]);
  where[c] = r;
  for (int i = 0; i < n; i++) if ( i != r ) {
   double mul = mat[i][c] / mat[r][c];
   for (int j = c; j <= m; j++) mat[i][j] -= mul * mat[r][j];
```

```
  }
  r++;
 }
 for (int j = 0; j < m; j++) {
  if (where[j] != -1) ans[j] = mat[where[j]][m]
  ↪ / mat[where[j]][j];
  else ans[j] = 0;
 }
 for (int i = 0; i < n; i++) {
  double sum = 0;
  for (int j = 0; j < m; j++) sum += ans[j] *
  ↪ mat[i][j];
  if ( abs(sum - mat[i][m]) > EPS ) return 0;
  ↪ /// no solution
 }
 for (int j = 0; j < m; j++) if (where[j] ==
 ↪ -1) return INF;
 return 1;
}
```

6.1.4 Matrix expo

```
long long a, b, n, m, F[2][2], f[2][2];
long long p = 1e9 + 7;
void multiply( long long a[2][2], long long
 ↪ b[2][2]) {
 long long g[2][2];
 for ( int i = 0; i < 2; i++ ) {
  for ( int j = 0; j < 2; j++ ) {
   g[i][j] = 0; for ( int k = 0; k < 2; k++)
    g[i][j] = ( (g[i][j] % p) + ((a[i][k] % p)
 ↪ * (b[k][j] % p)) % p ) % p;
  }
 }
 for ( int i = 0; i < 2; i++ ) {
  for ( int j = 0; j < 2; j++ ) F[i][j] =
  ↪ g[i][j];
 }
}
void power( long long N ) {
 if ( N == 1 ) return;
 if ( N % 2 == 0 ) { power( N / 2 );
 ↪ multiply(F, F); }
 else {power(N - 1); multiply(F, f);}
 return;
}
```

6.2 Modular Arithmetic

6.2.1 Chinese Remainder Theorem

```
/*** X = a_1 % m_1
X = a_2 % m_2
X = a_3 % m_3
m_1,m_2,m_3 are pair wise co-prime
M = m_1*m_2*m_3
u_i = Modular inverse of (M/m_i) with respect
 ↪ m_i
X = ( a_1 * (M/m_1) * u_1 + a_2 * (M/m_2) * u_2
 ↪ + a_3 * (M/m_3) * u_3 ) % M ***/
ll inv(ll a, ll m) {
 ll m0 = m, t, q;
 ll x0 = 0, x1 = 1;
 if (m == 1) return 0;
 while (a > 1) {
  q = a / m; t = m; m = a % m, a = t; t = x0;
 ↪ x0 = x1 - q * x0; x1 = t;
 }
 if (x1 < 0) x1 += m0;
 return x1;
}
ll findMinX(ll num[], ll rem[], ll k) {
 ll prod = 1;
 for (ll i = 0; i < k; i++) prod *= num[i];
 ll result = 0;
 for (ll i = 0; i < k; i++) {
  ll pp = prod / num[i];
  result += rem[i] * inv(pp, num[i]) * pp;
 }
 return result % prod;
}
int main() {
 ll num[15], rem[15], n, t, i, j;
 scanf("%lld", &t);
 for (i = 1; i <= t; i++) {
  scanf("%lld", &n);
  for (j = 0; j < n; j++)
   scanf("%lld %lld", &num[j], &rem[j]);
  printf("Case %lld: %lld\n", i, findMinX(num,
 ↪ rem, n));
 }
}
```

6.2.2 Discrete Log

```
int solve(int a, int b, int m) {
 a %= m, b %= m;
 int k = 1, add = 0, g;
 while ((g = gcd(a, m)) > 1) {
  if (b == k)
   return add;
  if (b % g)
   return -1;
  b /= g, m /= g, ++add;
  k = (k * 1ll * a / g) % m;
 }
 int n = sqrt(m) + 1;
 int an = 1;
 for (int i = 0; i < n; ++i)
  an = (an * 1ll * a) % m;
 unordered_map<int, int> vals;
 for (int q = 0, cur = b; q <= n; ++q) {
  vals[cur] = q;
  cur = (cur * 1ll * a) % m;
 }
 for (int p = 1, cur = k; p <= n; ++p) {
  cur = (cur * 1ll * an) % m;
  if (vals.count(cur)) {
   int ans = n * p - vals[cur] + add;
   return ans;
  }
 }
 return -1;
}
```

6.2.3 Modular Inverse (EGCD)

```
int gcdExtended(int a, int b, int* x, int* y) {
 if (a == 0) {
  *x = 0, *y = 1;
  return b;
 }
 int x1, y1;
 int gcd = gcdExtended(b % a, a, &x1, &y1);
 *x = y1 - (b / a) * x1;
 *y = x1;
 return gcd;
}
void modInverse(int a, int m) {
 int x, y;
 int g = gcdExtended(a, m, &x, &y);
 if (g != 1)
  printf("Inverse doesn't exist");
 else {
  int res = (x % m + m) % m;
  printf("Modular multiplicative inverse is
 ↪ %d\n", res);
 }
}
```

6.2.4 Modular Inverse

```
int gcdExtended(int a, int b, int* x, int* y) {
 if (a == 0) {
  *x = 0, *y = 1;
  return b;
 }
 int x1, y1;
 int gcd = gcdExtended(b % a, a, &x1, &y1);
 *x = y1 - (b / a) * x1;
 *y = x1;
 return gcd;
}
void modInverse(int a, int m) {
 int x, y;
 int g = gcdExtended(a, m, &x, &y);
 if (g != 1)
  printf("Inverse doesn't exist");
 else {
  int res = (x % m + m) % m;
  printf("Modular multiplicative inverse is
 ↪ %d\n", res);
 }
}
```

6.2.5 nCr Lucas

```
/*use this to calculate nCr modulo mod, when mod is smaller than n and m. define MOD
Complexity : O(mod + log mod n) */
ll fact[MOD];
ll bigmod(int x, int p) {
 ll res = 1;
 while (p) {
  if (p & 1) res = res * x % MOD;
  x = x * x % MOD;
  p >>= 1;
 }
 return res;
}
ll modinv(ll x) {
 return bigmod(x, MOD - 2);
}
void precalc() { //run this
 fact[0] = 1;
 for (int i = 1; i < MOD; i++) {
  fact[i] = fact[i - 1] * i % MOD;
 }
}
int C(int n, int m) {
 if (m > n) return 0;
 if (m == 0 or m == n) return 1;
 ll ret = fact[n] * modinv(fact[m]) % MOD;
 return ret * modinv(fact[n - m]) % MOD;
}
int nCr(int n, int m) {
 if (m > n) return 0;
 if (m == 0) return 1;
 return nCr(n / MOD, m / MOD) * C(n % MOD, m % MOD) % MOD;
}
```

6.3 Polynomial Multiplication

6.3.1 FFT

```
typedef cplx cd;
//define N as a power of two greater than the size
, of any possible polynomial
using cd = complex<double>;
const double PI = acosl(-1);
int rev[N]; cd w[N];
static cd f[N];
void prepare(int &n) {
 int sz = __builtin_ctz(n);
 for (int i = 1; i < n; i++) rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (sz - 1));
 w[0] = 0, w[1] = 1, sz = 1;
 while (1 << sz < n) {
  cd w_n = cd(cos(2 * PI / (1 << (sz + 1))), sin(2 * PI / (1 << (sz + 1))));
  for (int i = 1 << (sz - 1); i < (1 << sz); i++) {
   w[i << 1] = w[i], w[i << 1 | 1] = w[i] * w_n;
  }
  sz++;
 }
}
void fft(cd *a, int n) {
 for (int i = 1; i < n - 1; i++) {
  if (i < rev[i]) swap(a[i], a[rev[i]]);
 }
 for (int h = 1; h < n; h <<= 1) {
  for (int s = 0; s < n; s += h << 1) {
   for (int i = 0; i < h; i++) {
    cd &u = a[s + i], &v = a[s + i + h], t = v * w[h + i];
    v = u - t, u = u + t;
   }
  }
 }
}
vector<ll>multiply(vector<ll>a, vector<ll>b) {
 int n = a.size(), m = b.size(), sz = 1;
 if (!n or !m) return {};
 while (sz < n + m - 1) sz <<= 1;
 prepare(sz);
 for (int i = 0; i < sz; i++) f[i] = cd(i < n ? a[i] : 0, i < m ? b[i] : 0);
 fft(f, sz);
 for (int i = 0; i <= (sz >> 1); i++) {
  int j = (sz - i) & (sz - 1);
  cd x = (f[i] * f[i] - conj(f[j] * f[j])) * cd(0, -0.25);
  f[j] = x, f[i] = conj(x);
 }
 fft(f, sz);
 vector<ll>c(n + m - 1);
 for (int i = 0; i < n + m - 1; i++) c[i] = round(f[i].real() / sz);
 return c;
}
```

6.3.2 NTT

```
const int G = 3;
const int MOD = 998244353;
const int N = ?; // (1 << 20) + 5; greater than
, maximum possible degree of any polynomial
int rev[N], w[N], inv_n;
int bigMod(int a, int e, int mod) {
 if (e == -1) assert(false);
 if (e == -1) e = mod - 2;
 int ret = 1;
 while (e) {
  if (e & 1) ret = (ll) ret * a % mod;
  a = (ll) a * a % mod; e >>= 1;
 }
 return ret;
}
void prepare(int &n) {
 int sz = abs(31 - __builtin_clz(n));
 int r = bigMod(G, (MOD - 1) / n, MOD);
 inv_n = bigMod(n, MOD - 2, MOD), w[0] = w[n] = 1;
 for (int i = 1; i < n; ++i) w[i] = (ll) w[i - 1] * r % MOD;
 for (int i = 1; i < n; ++i) rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (sz - 1));
}
void ntt (int *a, int n, int dir) {
 for (int i = 1; i < n - 1; ++i) {
  if (i < rev[i]) swap(a[i], a[rev[i]]);
 }
 for (int m = 2; m <= n; m <<= 1) {
  for (int i = 0; i < n; i += m) {
   for (int j = 0; j < (m >> 1); ++j) {
    int &u = a[i + j], &v = a[i + j + (m >> 1)];
    int t = (ll) v * w[dir ? n - n / m * j : n / m * j] % MOD;
    v = u - t < 0 ? u - t + MOD : u - t;
    u = u + t >= MOD ? u + t - MOD : u + t;
   }
  }
 }
 if (dir) for (int i = 0; i < n; ++i) a[i] = (ll) a[i] * inv_n % MOD;
}
int f_a[N], f_b[N];
vector <int> multiply (vector <int> a, vector <int> b) {
 int sz = 1, n = a.size(), m = b.size();
 while (sz < n + m - 1) sz <<= 1; prepare(sz);
 for (int i = 0; i < sz; ++i) f_a[i] = i < n ? a[i] : 0;
 for (int i = 0; i < sz; ++i) f_b[i] = i < m ? b[i] : 0;
 ntt(f_a, sz, 0); ntt(f_b, sz, 0);
 for (int i = 0; i < sz; ++i) f_a[i] = (ll) f_a[i] * f_b[i] % MOD;
 ntt(f_a, sz, 1); return vector <int> (f_a, f_a + n + m - 1);
}
// G = primitive_root(MOD)
int primitive_root (int p) {
 vector <int> factor;
 int tmp = p - 1;
 for (int i = 2; i * i <= tmp; ++i) {
  if (tmp % i == 0) {
   factor.emplace_back(i);
   while (tmp % i == 0) tmp /= i;
  }
 }
 if (tmp != 1) factor.emplace_back(tmp);
 for (int root = 1; ; ++root) {
  bool flag = true;
  for (int i = 0; i < (int) factor.size(); ++i) {
```

```cpp
   if (bigMod(root, (p - 1) / factor[i], p) ==
    1) {
    flag = false; break;
   }
  }
  if (flag) return root;
 }
}
int main() {
// (x + 2)(x + 3) = x^2 + 5x + 6
 vector <int> a = {2, 1};
 vector <int> b = {3, 1};
 vector <int> c = multiply(a, b);
 for (int x : c) cout << x << " "; cout << endl;
 return 0;
}
```

6.4 Catalan Number

```cpp
unsigned long int binomialCoeff(unsigned int n,
    unsigned int k) {
 unsigned long int res = 1;
 if (k > n - k)
  k = n - k;
 for (int i = 0; i < k; ++i) {
  res *= (n - i); res /= (i + 1);
 }
 return res;
}
unsigned long int catalan(unsigned int n) {
 unsigned long int c = binomialCoeff(2 * n, n);
 return c / (n + 1);
}
```

6.5 Diophantine Equation

```cpp
int gcd_extend(int a, int b, int& x, int& y)
{
 if (b == 0) {
  x = 1;
  y = 0;
  return a;
 }
 else {
  int g = gcd_extend(b, a % b, x, y);
  int x1 = x, y1 = y;
  x = y1;
  y = x1 - (a / b) * y1;
  return g;
 }
}
void print_solution(int a, int b, int c) {
 int x, y;
 if (a == 0 && b == 0) {
  if (c == 0) {
   cout << "Infinite Solutions Exist" << endl;
  }
  else {
   cout << "No Solution exists" << endl;
  }
 }
 int gcd = gcd_extend(a, b, x, y);
 if (c % gcd != 0) {
  cout << "No Solution exists" << endl;
 }
 else {
  cout << "x = " << x * (c / gcd) << ", y = "
    << y * (c / gcd) << endl;
 }
}
```

6.6 Euler Totient

```cpp
int phi(int n) {
 int result = n;
 for (int i = 2; i * i <= n; i++) {
  if (n % i == 0) {
   while (n % i == 0) n /= i;
   result -= result / i;
  }
 }
 if (n > 1)
  result -= result / n;
 return result;
}
void phi_1_to_n(int n) {
 vector<int> phi(n + 1);
 phi[0] = 0;
 phi[1] = 1;
 for (int i = 2; i <= n; i++)
  phi[i] = i;
 for (int i = 2; i <= n; i++) {
  if (phi[i] == i) {
   for (int j = i; j <= n; j += i)
    phi[j] -= phi[j] / i;
  }
 }
}
```

6.7 FastSieve

```cpp
// primes up to 5e8 within 0.35 seconds
// primes up to 1e9 within 1 second
vector <int> fastSieve (const int N, const int
    Q = 17, const int L = 1 << 15) {
 const int M = (N + 29) / 30;
 const int two = sqrt(N), four = sqrt(two);
 static const int r[] = {1, 7, 11, 13, 17, 19,
    23, 29};
 struct P {
   P (int p) : p(p) {}
   int p, pos[8];
 };
 auto approxPrimeCount = [] (const int N) ->
    int {
    return N > 60184 ? N / (log(N) - 1.1) :
    max(1.0, N / (log(N) - 1.11)) + 1;
 };
 vector <bool> isPrime(two + 1, true);
 for (int i = 2; i <= four; ++i) if
    (isPrime[i]) {
     for (int j = i * i; j <= two; j += i)
    isPrime[j] = false;
    }
 const int r_size = approxPrimeCount(N + 30);
 int p_size = 3;
 vector <P> s_primes;
 vector <int> primes = {2, 3, 5};
 int p_beg = 0, prod = 1;
 primes.resize(r_size);
 for (int p = 7; p <= two; ++p) {
   if (!isPrime[p]) continue;
   if (p <= Q) prod *= p, ++p_beg,
    primes[p_size++] = p;
   auto cur = P(p);
   for (int t = 0; t < 8; ++t) {
     int j = (p <= Q) ? p : p * p;
     while (j % 30 != r[t]) j += p << 1;
     cur.pos[t] = j / 30;
   }
   s_primes.push_back(cur);
 }
 vector <unsigned char> pre(prod, 0xFF);
 for (size_t it = 0; it < p_beg; ++it) {
   auto cur = s_primes[it];
   const int p = cur.p;
   for (int t = 0; t < 8; ++t) {
     const unsigned char m = ~(1 << t);
     for (int i = cur.pos[t]; i < prod; i +=
    p) pre[i] &= m;
   }
 }
 const int block_size = (L + prod - 1) / prod
    * prod;
 vector <unsigned char> block(block_size);
 unsigned char *p_block = block.data();
 for (int beg = 0; beg < M; beg += block_size,
    p_block -= block_size) {
   int end = min(M, beg + block_size);
   for (int i = beg; i < end; i += prod) {
     copy(pre.begin(), pre.end(), p_block + i);
   }
   if (beg == 0) p_block[0] &= 0xFE;
   for (size_t it = p_beg; it <
    s_primes.size(); ++it) {
     auto &cur = s_primes[it];
     const int p = cur.p;
     for (int t = 0; t < 8; ++t) {
       int i = cur.pos[t];
       const unsigned char m = ~(1 << t);
       for (; i < end; i += p) p_block[i] &= m;
       cur.pos[t] = i;
     }
```

```
    }
    for (int i = beg; i < end; ++i) {
      for (int m = p_block[i]; m > 0; m &= m - 1) {
        primes[p_size++] = i * 30 + r[__builtin_ctz(m)];
      }
    }
  }
  assert(p_size <= r_size);
  while (p_size > 0 and primes[p_size - 1] > N) --p_size;
  primes.resize(p_size); return primes;
}
int main() {
  int LIM; cin >> LIM;
  auto primes = fastSieve(LIM);
}
```

6.8 Primes

```
const int N = 1000000 + 6;
vector<long long>primes;
bitset<N>flag;
vector<long long>v;
void siv() {
 flag[1] = 1;
 for ( int i = 2; i * i <= N; i++ ) {
  if ( flag[i] == 0 ) {
   for ( int j = i * i; j < N; j += i ) flag[j] = 1;
  }
 }
 for ( int i = 2; i < N; i++ ) {
  if ( flag[i] == 0 ) primes.push_back(i);
 }
}
long long mul(long long a, long long b, long long mod) {
 long long res = 0;
 a %= mod;
 while (b) {
  if (b & 1) res = (res + a) % mod;
  a = (2 * a) % mod;
  b >>= 1; // b = b / 2
 }
 return res;
}
long long mod_inverse( long long n, long long p ) {
 long long x, y, g;
 g = gcd_extended( n, p, x, y );
 if ( g < 0 ) x = -x;
 return (x % p + p) % p;
}
long long mpow( long long x, long long y, long long mod ) {
 long long ret = 1;
 while ( y ) {
  if ( y & 1 ) ret = mul(ret, x, mod);
  y >>= 1, x = mul(x, x, mod);
 }
 return ret % mod;
}
int isPrime( long long p ) {
 if ( p < 2 || !(p & 1) ) return 0;
 if ( p == 2 ) return 1;
 long long q = p - 1, a, t;
 int k = 0, b = 0;
 while ( !(q & 1) ) q >>= 1, k++;
 for ( int it = 0; it < 2; it++ ) {
  a = rand() % (p - 4) + 2;
  t = mpow( a, q, p );
  b = (t == 1) || ( t == p - 1 );
  for ( int i = 1; i < k && !b; i++ ) {
   t = mul(t, t, p);
   if ( t == p - 1 ) b = 1;
  }
  if ( b == 0 ) return 0;
 }
 return 1;
}
long long pollard_rho( long long n, long long c ) {
 long long x = 2, y = 2, i = 1, k = 2, d;
 while ( 1 ) {
  x = ( mul(x, x, n) + c );
  if ( x >= n ) x -= n;
  d = gcd(x - y, n);
  if ( d > 1 ) return d;
  if ( ++i == k ) y = x, k <<= 1;
 }
 return n;
}
map<long long, int>mp;
void factorize( long long n ) {
 int l = primes.size();
 for ( int i = 0; primes[i]*primes[i] <= n && i < l; i++ ) {
  if ( n % primes[i] == 0 ) {
   mp[primes[i]] = 1;
   while ( n % primes[i] == 0 ) n /= primes[i];
  }
 }
 if ( n != 1 ) mp[n] = 1;
}
void lfactorize( long long n ) {
 if ( n == 1 ) return;
 if ( n < 1e9 ) {
  factorize(n);
  return;
 }
 if ( isPrime(n) ) {
  mp[n] = 1;
  return;
 }
 long long d = n;
 for ( int i = 2; d == n; i++ ) d = pollard_rho(n, i);
 lfactorize(d);
 lfactorize(n / d);
}
long long f(long long r, vector<long long> v1) {
 int sz = v1.size();
 long long res = 0;
 for (long long i = 1; i < (1 << sz); i++) {
  int ct = 0;
  long long mul = 1;
  for (int j = 0; j < sz; j++ ) {
   if (i & (1 << j)) {
    ct++;
    mul *= v1[j];
   }
  }
  long long sign = -1;
  if (ct & 1)sign = 1;
  res += sign * (r / mul);
 }
 return r - res;
}
```

6.9 Striling Number of 2nd kind

```
long long p = 1e9 + 7;
long long fact[1000005];
int n, m, k;
long long s( long long N, long long R )
{
 if ( N == 0 && R == 0 ) return 1;
 if ( N == 0 || R == 0 ) return 0;
 long long ans = 0;
 for ( int i = 1; i <= R; i++ ) {
  long long par;
  if ( (R - i) % 2 == 0 ) par = 1;
  else par = -1;
  par = (par + p) % p;
  long long temp = (ncr(R, i) * bm(i, N)) % p;
  temp = (temp % p * par % p) % p;
  ans = (ans % p + temp % p) % p;
 }
 return (ans * bm( fact[R], p - 2 )) % p;
}
```

7 Misc

7.1 Build (Nafi)

```
{
"cmd" : ["g++ -std=c++14 $file_name -o $file_base_name && timeout 6s ./$file_base_name<in>out"],
"selector" : "source.c, source.cpp, source.Cc",
"shell": true,
"working_dir" : "$file_path"
}
```

7.2 Build files

```
//pragma
#pragma GCC optimize("O3")
#pragma GCC optimize("unroll-loops")
compile: g++ -std = c++17 - I . - Dakifpathan -
↪ o "%e" "%f"
                    build: g++ -std = c++17 -
↪ DHFTF - Wshadow - o "%e" "%f"
                                      - g -
↪ fsanitize = address - fsanitize = undefined
                                            -
↪ D_GLIBCXX_DEBUG
                                          run:
↪ "./%e"
//for sublime
{
"cmd" : ["g++ -std=c++14 $file_name -o
 $file_base_name && timeout 6s
 ./$file_base_name<in>out"],
"selector" : "source.c, source.cpp, source.Cc",
"shell": true,
"working_dir" : "$file_path"
}
//windows
{
"cmd": ["g++.exe", "-std=c++14", "${file}",
↪ "-o",
 "${file_base_name}.exe", "&&",
 "${file_base_name}.exe<in>out"],
"shell": true,
"working_dir": "$file_path",
"selector": "source.cpp, source.c, source.c++,
↪ source.cc"
}
```

7.3 Ternary Search

```
while (hi >= lo)
{
 int mid1 = lo + (hi - lo) / 3; int mid2 = hi -
↪ (hi - lo) / 3;
 if (f(mid1) > f(mid2)) { } //change
 else //change
 }//ittehad
double x1, why1, z1, x2, y2, z2, x, y , z;
double f( double t )
{
 double xt = x1 + (x2 - x1)t;
 double yt = why1 + (y2 - why1)t;
 double zt = z1 + (z2 - z1)t;
 return ((xt - x)(xt - x) + (yt - y)(yt - y) +
↪ (zt - z)(zt - z));
}
double Tsearch()
{
 double low = 0, high = 1, mid;
 int step = 64;
 while ( step-- ) {
  double t1 = (2low + high) / 3;
  double t2 = (low + 2high) / 3;
  double d1 = f(t1);
  double d2 = f(t2);
  if ( d1 < d2 ) high = t2;
  else low = t1;
 }
 return low;
}
```

7.4 fastIO

```
ios_base::sync_with_stdio(false);
↪ cin.tie(NULL); cout.tie(NULL)
```

8 String

8.1 Aho-Corasick

```
struct vartex {
 int next[30], endmark, link;
 vector<int>dlink;
 vartex() {
  memset(next, -1, sizeof(next));
  endmark = -1;
  link = 0;
 }
};
void addstring(string& s, vector<vartex>&trie) {
 int v = 0;
 for (auto x : s) {
  if (trie[v].next[x - 'a'] == -1) {
   trie[v].next[x - 'a'] = trie.size();
   trie.emplace_back();
  }
  v = trie[v].next[x - 'a'];
 }
 trie[v].endmark = 0;
}
void fail(vector<vartex>&trie) {
 int v = 0;
 trie[v].link = 0;
 queue<int>q;
 q.push(0);
 while (!q.empty()) {
  v = q.front();
  q.pop();
  for (int i = 0; i < 26; i++) {
   if (trie[v].next[i] != -1) {
    if (v == 0) {
     trie[trie[v].next[i]].link = 0;
    }
    else {
     int x = trie[v].link;
     while (x != 0 && trie[x].next[i] == -1) {
      x = trie[x].link;
     }
     if (trie[x].next[i] == -1) {
      trie[trie[v].next[i]].link = 0;
     }
     else {
      trie[trie[v].next[i]].link =
↪     trie[x].next[i];
     }
    }
    q.push(trie[v].next[i]);
   }
  }
 }
}
void dictionary_link(vector<vartex>&trie) {
 queue<int>q;
 q.push(0);
 while (!q.empty()) {
  int u = q.front();
  q.pop();
  for (int i = 0; i < 26; i++) {
   if (trie[u].next[i] != -1) {
    q.push(trie[u].next[i]);
   }
  }
  int k = u;
  while (k != 0) {
   if (trie[k].endmark != -1 && k != u) {
    trie[u].dlink.push_back(k);
   }
   k = trie[k].link;
  }
  debug(u, trie[u].dlink);
 }
}
int search(string& s, vector<vartex>&trie) {
 int v = 0;
 for (auto x : s) {
  v = trie[v].next[x - 'a'];
 }
 return trie[v].endmark;
}
```

8.2 Aho-Corasick_New

```
struct node {
    bool f = false;
    char c;
    int p = -1, link = -1, ex = -1;
    int to[26], go[26];
    vector <int> id;
    node() {
        memset(to, -1, sizeof(to));
        memset(go, -1, sizeof(go));
    }
};
int siz = 1;
vector <node> trie(1);
void insert(string &s, int len, int j) {
    int u, v = 0;
    for (int i = 0; i < len; i++) {
```

```cpp
        int c = s[i] - 97;
        if (trie[v].to[c] == -1) {
            trie.emplace_back();
            trie[v].to[c] = siz++;
        }
        u = v;
        v = trie[v].to[c];
        trie[v].p = u, trie[v].c = s[i];
    }
    trie[v].f = true;
    trie[v].id.push_back(j);
}
int go(int v, char c);
int get_link(int v) {
    if (trie[v].link == -1) {
        if (v == 0 || trie[v].p == 0)
 ↪  trie[v].link = 0;
        else trie[v].link =
 ↪  go(get_link(trie[v].p), trie[v].c);
    }
    return trie[v].link;
}
int get_exit_link(int v) {
    if (trie[v].ex == -1) {
        int u = get_link(v);
        if (u == 0 || trie[u].f) trie[v].ex = u;
        else  trie[v].ex = get_exit_link(u);
    }
    return trie[v].ex;
}
int go(int v, char c) {
    int x = c - 97;
    if (trie[v].go[x] == -1) {
        if (trie[v].to[x] != -1) trie[v].go[x]
 ↪  = trie[v].to[x];
        else trie[v].go[x] = v ?
 ↪  go(get_link(v), c) : 0;
    }
    return trie[v].go[x];
}
string s, t;
vector <int> a[100005];
int n, k[100005], len, ln[100005];
void get_id(int v, int i) {
    int sz = trie[v].id.size();
    for (int j = 0; j < sz; j++) {
        int p = trie[v].id[j];
        a[p].push_back(i - ln[p]);
    }
}
void fun(int i, int v) {
    if (trie[v].f) get_id(v, i);
    int u = get_exit_link(v);
    while (u > 0) {
        if (trie[u].f) get_id(u, i);
        u = get_exit_link(u);
    }
    if (i < s.size()) fun(i + 1, go(v, s[i]));
}
int query(int i) {
    int s = a[i].size(), ans = -1;
    for (int j = k[i] - 1, p = 0; j < s; j++,
 ↪  p++) {
        int x = a[i][j] + ln[i] - a[i][p];
        ans = (ans == -1) ? x : min(ans, x);
    }
    return ans;
}
int main() {
    cin >> s >> n;
    len = s.length();
    for (int i = 0; i < n; i++) {
        cin >> k[i] >> t;
        ln[i] = t.length();
        insert(t, ln[i], i);
    }
    fun(0, 0);
    for (int i = 0; i < n; i++) cout <<
 ↪  query(i) << endl;
}
```

8.3 Hashing without inv

```cpp
long long h[400005];
long long MOD[400005];
int L;
void pre_hash( string s ) {
 long long p = 31 , m = 1e9 + 9, power = 1,
 ↪  hash = 0;
 int z = 0;
 for ( int i = s.size() - 1; i >= 0; i-- ) {
  hash = ( hash * p + (s[i] - 'A' + 1) ) % m;
  h[i] = hash;
  MOD[z] = power;
  z++;
  power = (power * p) % m;
 }
}
long long f( int l, int r ) {
 long long val = h[r], m = 1e9 + 9;
 if ( l != L - 1 ) {
  long long val2 = (h[l + 1] % m * MOD[l - r +
 ↪  1] % m ) % m;
  val -= val2;
  val += m;
  val %= m;
 }
 if ( val < 0 ) val = (val + m) % m;
 return val;
}
```

8.4 KMP

```cpp
#define pii pair<int,int>
vector<int> prefix_function (string Z) {
 int n = (int) Z.length();
 vector<int> F (n);
 F[0] = 0;
 for (int i = 1; i < n; ++i) {
  int j = F[i - 1];
  while (j > 0 && Z[i] != Z[j])
   j = F[j - 1];
  if (Z[i] == Z[j]) ++j;
  F[i] = j;
 }
 return F;
}
```

8.5 Manacher

```cpp
/// When i is even, pal[i] = largest
 ↪  palindromic substring centered from str[i /
 ↪  2]
/// When i is odd, pal[i] = largest palindromic
 ↪  substring centered between str[i / 2] and
 ↪  str[i / 2] + 1
vector <int> manacher(char *str) {
 int i, j, k, l = strlen(str), n = l << 1;
 vector <int> pal(n);
 for (i = 0, j = 0, k = 0; i < n; j = max(0, j
 ↪  - k), i += k) {
  while (j <= i && (i + j + 1) < n && str[(i -
 ↪  j) >> 1] == str[(i + j + 1) >> 1])
   j++;
  for (k = 1, pal[i] = j; k <= i && k <= pal[i]
 ↪  && (pal[i] - k) != pal[i - k]; k++) {
   pal[i + k] = min(pal[i - k], pal[i] - k);
  }
 }
 pal.pop_back();
 return pal;
}
int main() {
 char str[100];
 while (scanf("%s", str)) {
  auto v = manacher(str);
  for (auto it : v) printf("%d ", it);
  puts("");
 }
 return 0;
}
```

8.6 Palindromic Tree

```cpp
#define CLR(a) memset(a,0,sizeof(a))
/***
* str is 1 based
Each node in the palindromic tree denotes a
 ↪  STRING
Node 1 denotes an imaginary string of size -1
Node 2 denotes a string of size 0
They are the two roots
There can be maximum of (string_length + 2)
 ↪  nodes in total
```

```
It's a directed tree. If we reverse the
direction of the suffix links, we get a dag. In
this DAG, if node v is reachable from node u
iff, u is a substring of v.
* if ( tree[A].next[x] == B )
then, B = xAx
* if ( tree[A].suffixLink == B )
Then B is the longest possible palindrome which
 ↪ is a proper suffix of A
(node 1 is an exception)
* occ[i] contains the number of occurrences of
 ↪ the corresponding palindrome
* st[i] denotes starting index of the first
 ↪ occurrence of the corresponding palindrome
* st[] or occ[] or both can be ignored if not
 ↪ needed
* If memory limit is compact, a map has to be
 ↪ used instead of
ed[MAXN][MAXC]. Swapping row and column of the
 ↪ matrix will
save more memory.
Example :
map <int,int> ed[MAXC];
ed[c][u] = v means, there is an edge from node
 ↪ u to
node v that is labeled character c.
***/
namespace pt {
const int MAXN = 100010; /// maximum possible
 ↪ string size
const int MAXC = 26; /// Size of the character
 ↪ set
int n; /// length of str
char str[MAXN];
int len[MAXN], link[MAXN], ed[MAXN][MAXC],
 ↪ occ[MAXN], st[MAXN];
int nc, suff, pos;
/// nc -> node count
/// suff -> Index of the node denoting the
 ↪ longest palindromic proper suffix of the
 ↪ current prefix
void init() {
 str[0] = -1;
 nc = 2; suff = 2;
 len[1] = -1, link[1] = 1;
 len[2] = 0, link[2] = 1;
 CLR(ed[1]), CLR(ed[2]);
 occ[1] = occ[2] = 0;
}
inline int scale(char c) { return c - 'a'; }
inline int nextLink(int cur) {
 while (str[pos - 1 - len[cur]] != str[pos])
 ↪ cur = link[cur];
 return cur;
}
inline bool addLetter(int p) {
 pos = p;
 int let = scale(str[pos]);
 int cur = nextLink(suff);
 if (ed[cur][let]) {
  suff = ed[cur][let];
  occ[suff]++;
  return false;
 }
 suff = ++nc;
 CLR(ed[nc]);
 len[nc] = len[cur] + 2;
 ed[cur][let] = nc;
 occ[nc] = 1;
 if (len[nc] == 1) {
  st[nc] = pos;
  link[nc] = 2;
  return true;
 }
 link[nc] = ed[nextLink(link[cur])][let];
 st[nc] = pos - len[nc] + 1;
 return true;
}
void build(int _n) {
 n = _n;
 init();
 for (int i = 1; i <= n; i++) addLetter(i);
 for (int i = nc; i >= 3; i--) occ[link[i]] +=
 ↪ occ[i];
 occ[2] = occ[1] = 0;
}
void printTree() {
 puts(str);
 cout << "Node\tStart\tLength\tOcc\n";
 for (int i = 3; i <= nc; i++) {
  cout << i << "\t" << st[i] << "\t" << len[i]
 ↪ << "\t" << occ[i] << "\n";
 }
}
}
int main() {
 scanf("%s", pt::str + 1);
 pt::build(strlen(pt::str + 1));
 return 0;
}
```

8.7 String Hashing

```
ll bigmod(ll x, ll p, ll md) {
 ll res = 1;
 while (p) {
  if (p & 1) res = res * x % md;
  x = x * x % md;
  p >>= 1;
 }
 return res;
}
ll modinv(ll x, ll md) {
 return bigmod(x, md - 2, md);
}
namespace Hash {
ll pw[M][2];
ll invpw[M][2];
const int pr[] = {37, 53};
const int md[] = {1000000007, 1000000009};
void precalc() {
 pw[0][0] = pw[0][1] = 1;
 for (int i = 1; i < M; i++) {
  pw[i][0] = pw[i - 1][0] * pr[0] % md[0];
  pw[i][1] = pw[i - 1][1] * pr[1] % md[1];
 }
 invpw[M - 1][0] = modinv(pw[M - 1][0], md[0]);
 invpw[M - 1][1] = modinv(pw[M - 1][1], md[1]);
 for (int i = M - 2; i >= 0; i--) {
  invpw[i][0] = invpw[i + 1][0] * pr[0] % md[0];
  invpw[i][1] = invpw[i + 1][1] * pr[1] % md[1];
 }
}
pii get_hash(const string &s) {
 pii ret = {0, 0};
 for (int i = 0; i < s.size(); i++) {
  ret.first += (s[i] - 'a' + 1) * pw[i][0] %
 ↪ md[0];
  ret.second += (s[i] - 'a' + 1) * pw[i][1] %
 ↪ md[1];
  if (ret.first >= md[0]) ret.first -= md[0];
  if (ret.second >= md[1]) ret.second -= md[1];
 }
 return ret;
}
void prefix(const string &s, pii *H) {
 H[0] = {0, 0};
 for (int i = 1; i <= s.size(); i++) {
  H[i].first = H[i - 1].first + (s[i - 1] - 'a'
 ↪ + 1) * pw[i - 1][0] % md[0];
  H[i].second = H[i - 1].second + (s[i - 1] -
 ↪ 'a' + 1) * pw[i - 1][1] % md[1];
  if (H[i].first >= md[0]) H[i].first -= md[0];
  if (H[i].second >= md[1]) H[i].second -=
 ↪ md[1];
 }
}
void reverse_prefix(const string &s, pii *H) {
 int n = s.size();
 for (int i = 1; i <= s.size(); i++) {
  H[i].first = H[i - 1].first + (s[i - 1] - 'a'
 ↪ + 1) * pw[n - i][0] % md[0];
  H[i].second = H[i - 1].second + (s[i - 1] -
 ↪ 'a' + 1) * pw[n - i][1] % md[1];
  if (H[i].first >= md[0]) H[i].first -= md[0];
  if (H[i].second >= md[1]) H[i].second -=
 ↪ md[1];
 }
}
pii range_hash(int L, int R, pii H[]) {
 pii ret;
 ret.first = (H[R].first - H[L - 1].first +
 ↪ md[0]) % md[0];
 ret.second = (H[R].second - H[L - 1].second +
 ↪ md[1]) % md[1];
 ret.first = ret.first * invpw[L - 1][0] %
 ↪ md[0];
```

```
 ret.second = ret.second * invpw[L - 1][1] % md[1];
 return ret;
}
pii reverse_hash(int L, int R, pii H[], int n) {
 pii ret;
 ret.first = (H[R].first - H[L - 1].first + md[0]) % md[0];
 ret.second = (H[R].second - H[L - 1].second + md[1]) % md[1];
 ret.first = ret.first * invpw[n - R][0] % md[0];
 ret.second = ret.second * invpw[n - R][1] % md[1];
 return ret;
}
}
```

8.8 Suffix Array

```
/*
  O(|S| + |alphabet|) Suffix Array
  LIM := max{s[i]} + 2
*/

void inducedSort (const vector <int> &vec, int val_range, vector <int> &SA, const vector <int> &sl, const vector <int> &lms_idx) {
 vector <int> l(val_range, 0), r(val_range, 0);
 for (int c : vec) {
  ++r[c]; if (c + 1 < val_range) ++l[c + 1];
 }
 partial_sum(l.begin(), l.end(), l.begin());
 partial_sum(r.begin(), r.end(), r.begin());
 fill(SA.begin(), SA.end(), -1);
 for (int i = lms_idx.size() - 1; i >= 0; --i) SA[--r[vec[lms_idx[i]]]] = lms_idx[i];
 for (int i : SA) if (i > 0 and sl[i - 1]) SA[l[vec[i - 1]]++] = i - 1;
 fill(r.begin(), r.end(), 0);
 for (int c : vec) ++r[c];
 partial_sum(r.begin(), r.end(), r.begin());
 for (int k = SA.size() - 1, i = SA[k]; k; --k, i = SA[k]) {
  if (i and !sl[i - 1]) SA[--r[vec[i - 1]]] = i - 1;
 }
}

vector <int> suffixArray (const vector <int> &vec, int val_range) {
 const int n = vec.size();
 vector <int> sl(n), SA(n), lms_idx;
 for (int i = n - 2; i >= 0; --i) {
  sl[i] = vec[i] > vec[i + 1] or (vec[i] == vec[i + 1] and sl[i + 1]);
  if (sl[i] and !sl[i + 1]) lms_idx.emplace_back(i + 1);
 }
 reverse(lms_idx.begin(), lms_idx.end());
 inducedSort(vec, val_range, SA, sl, lms_idx);
 vector <int> new_lms_idx(lms_idx.size()), lms_vec(lms_idx.size());
 for (int i = 0, k = 0; i < n; ++i) {
  if (SA[i] > 0 and !sl[SA[i]] and sl[SA[i] - 1]) new_lms_idx[k++] = SA[i];
 }
 int cur = 0; SA[n - 1] = 0;
 for (int k = 1; k < new_lms_idx.size(); ++k) {
  int i = new_lms_idx[k - 1], j = new_lms_idx[k];
  if (vec[i] ^ vec[j]) {
   SA[j] = ++cur; continue;
  }
  bool flag = 0;
  for (int a = i + 1, b = j + 1; ; ++a, ++b) {
   if (vec[a] ^ vec[b]) {
    flag = 1; break;
   }
   if ((!sl[a] and sl[a - 1]) or (!sl[b] and sl[b - 1])) {
    flag = !(!sl[a] and sl[a - 1] and !sl[b] and sl[b - 1]); break;
   }
  }
  SA[j] = flag ? ++cur : cur;
 }
 for (int i = 0; i < lms_idx.size(); ++i) lms_vec[i] = SA[lms_idx[i]];
 if (cur + 1 < lms_idx.size()) {
  auto lms_SA = suffixArray(lms_vec, cur + 1);
  for (int i = 0; i < lms_idx.size(); ++i) new_lms_idx[i] = lms_idx[lms_SA[i]];
 }
 inducedSort(vec, val_range, SA, sl, new_lms_idx); return SA;
}

vector <int> getSuffixArray (const string &s, const int LIM = 128) {
 vector <int> vec(s.size() + 1);
 copy(begin(s), end(s), begin(vec)); vec.back() = '$';
 auto ret = suffixArray(vec, LIM);
 ret.erase(ret.begin()); return ret;
}

// build RMQ on it to get LCP of any two suffix
vector <int> getLCParray (const string &s, const vector <int> &SA) {
 int n = s.size(), k = 0;
 vector <int> lcp(n), rank(n);
 for (int i = 0; i < n; ++i) rank[SA[i]] = i;
 for (int i = 0; i < n; ++i, k ? --k : 0) {
  if (rank[i] == n - 1) {
   k = 0; continue;
  }
  int j = SA[rank[i] + 1];
  while (i + k < n and j + k < n and s[i + k] == s[j + k]) ++k;
  lcp[rank[i]] = k;
 }
 lcp[n - 1] = 0; return lcp;
}
```

8.9 Suffix Automaton

```
// collected from cp algorithm
struct state {
 int len, link, cnt, firstpos; // cnt -> endpos set size, link -> suffix link
 map <char, int> next;
};
const int MAXLEN = 100002;
state st[MAXLEN * 2];
struct SuffixAutomata { // 0-based
 int sz, last;
 SuffixAutomata() { // init
  st[0].cnt = st[0].len = 0;
  st[0].link = -1;
  sz = 1, last = 0;
 }
 void add(char c) { // add new char in automata
  int cur = sz++;
  st[cur].len = st[last].len + 1;
  st[cur].firstpos = st[cur].len - 1;
  st[cur].cnt = 1;
  int p = last;
  while (p != -1 && !st[p].next.count(c)) {
   st[p].next[c] = cur;
   p = st[p].link;
  }
  if (p == -1) {
   st[cur].link = 0;
  }
  else {
   int q = st[p].next[c];
   if (st[p].len + 1 == st[q].len) {
    st[cur].link = q;
   }
   else { // clone state
    int clone = sz++;
    st[clone].len = st[p].len + 1;
    st[clone].next = st[q].next;
    st[clone].link = st[q].link;
    st[clone].firstpos = st[q].firstpos;
    st[clone].cnt = 0;
    while (p != -1 && st[p].next[c] == q) {
     st[p].next[c] = clone;
     p = st[p].link;
    }
    st[q].link = st[cur].link = clone;
   }
  }
```

```
  last = cur;
 }
 void occurrence() { // calculate number of
 ↪  occurrences of all possible substring
  vector <int> rank(sz);
  iota(all(rank), 0);
  sort(all(rank), [&](int i, int j) {
   return st[i].len > st[j].len;
  });
  for (int ii : rank) if (st[ii].link != -1)
    st[st[ii].link].cnt += st[ii].cnt;
 }
 int count(string s) { // number of occurrences
 ↪  of string s. #prerequisite -> call
 ↪  occurrence()
  int node = 0;
  for (char ch : s) {
   if (!st[node].next.count(ch)) return 0;
   node = st[node].next[ch];
  }
  return st[node].cnt;
 }
 int firstOcc(string s) { // first
 ↪  position(occurence) of string s
  int node = 0;
  for (char ch : s) {
   if (!st[node].next.count(ch)) return -1;
   node = st[node].next[ch];
  }
  return st[node].firstpos + 2 - (int)s.size();
 }
 void build(string S) { // build suffix automata
  for (char ch : S) add(ch);
 }
 bool find(string s) { // find string s in
 ↪  automata
  int node = 0;
  for (char ch : s) {
   if (!st[node].next.count(ch)) return false;
   node = st[node].next[ch];
  }
  return true;
 }
};
```

8.10 Trie

```
//define M, K = alphabet size
int trie[M][K], word[M * K + 3], cnt[M * K +
 ↪  3], sz;
void Insert(string s) {
 int node = 0;
 for (int i = 0; i < s.size(); i++) {
  int c = s[i] - 'a';
  if (!trie[node][c]) {
   trie[node][c] = ++sz;
  }
  node = trie[node][c];
  cnt[node]++;
 }
 word[node]++;
}
bool Search(string s) {
 int node = 0, ret = 0;
 for (int i = 0; i < s.size(); i++) {
  int c = s[i] - 'a';
  if (!trie[node][c]) return false;
  node = trie[node][c];
 }
 return (word[node] > 0);
}
void Delete(string s) {
 int node = 0;
 vector<int>v(1, 0);
 for (int i = 0; i < s.size(); i++) {
  int c = s[i] - 'a';
  node = trie[node][c];
  cnt[node]--;
  v.push_back(node);
 }
 word[node]--;
 for (int i = 1; i < v.size(); i++) {
  int c = s[i - 1] - 'a';
  if (!cnt[v[i]]) {
   trie[v[i - 1]][c] = 0;
  }
 }
}
```

8.11 Z algo

```
/* z[i] denotes the maximum length of substring
* starting from position(i) which is also a
 ↪  prefix
* of the string
* call with Z zf(x) where x is the desired
 ↪  string*/
struct Z {
 int n; string s;
 vector<int>z;
 Z(const string &a) {
  n = a.size(); s = a; z.assign(n, 0);
 }
 void z_function() {
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
   if (i <= r) z[i] = min (r - i + 1, z[i - l]);
   while (i + z[i] < n && s[z[i]] == s[i +
 ↪  z[i]]) ++z[i];
   if (i + z[i] - 1 > r) l = i, r = i + z[i] -
 ↪  1;
  }
 }
};
```