Contents	
Topic	***Page***

Combinatorics

Bishop

```
int squares (int i) {
    if (i & 1)     return i / 4 * 2 + 1;
    else     return (i - 1) / 4 * 2 + 2;
}
int bishop_placements(int N, int K){
    if (K > 2 * N - 1)   return 0;
    vector<vector<int>> D(N * 2, vector<int>(K + 1));
    for (int i = 0; i < N * 2; ++i)  D[i][0] = 1;
    D[1][1] = 1;
    for (int i = 2; i < N * 2; ++i)
        for (int j = 1; j <= K; ++j)
            D[i][j] = D[i-2][j] + D[i-2][j-1] * (squares(i) - j + 1);
    int ans = 0;
    for (int i = 0; i <= K; ++i)      ans += D[N*2-1][i] * D[N*2-2][K-i];
    return ans;
}
```

Bracket Sequence

```
bool next_balanced_sequence(string & s) {
    int n = s.size(), depth = 0;
    for (int i = n - 1; i >= 0; i--) {
        if (s[i] == '(')       depth--;
        else                   depth++;
        if (s[i] == '(' && depth > 0) {
            depth--;
            int open = (n - i - 1 - depth) / 2, close = n - i - 1 - open;
            string next = s.substr(0, i) + ')' + string(open, '(') + string(close, ')');
            s.swap(next);
            return true;
        }
    }
    return false;
```

```
}
string kth_balanced(int n, int k) {
    vector<vector<int>> d(2*n+1, vector<int>(n+1, 0));
    d[0][0] = 1;
    for (int i = 1; i <= 2*n; i++) {
        d[i][0] = d[i-1][1];
        for (int j = 1; j < n; j++)  d[i][j] = d[i-1][j-1] + d[i-1][j+1];
        d[i][n] = d[i-1][n-1];
    }
    string ans;
    int depth = 0;
    for (int i = 0; i < 2*n; i++) {
        if (depth + 1 <= n && d[2*n-i-1][depth+1] >= k) {
            ans += '(';   depth++;
        } else {
            ans += ')';
            if (depth + 1 <= n)        k -= d[2*n-i-1][depth+1];
            depth--;
        }
    }
    return ans;
}
string kth_balanced2(int n, int k) {
    vector<vector<int>> d(2*n+1, vector<int>(n+1, 0));
    d[0][0] = 1;
    for (int i = 1; i <= 2*n; i++) {
        d[i][0] = d[i-1][1];
        for (int j = 1; j < n; j++)  d[i][j] = d[i-1][j-1] + d[i-1][j+1];
        d[i][n] = d[i-1][n-1];
    }
    string ans;
    int depth = 0;
    stack<char> st;
    for (int i = 0; i < 2*n; i++) {
        // '('
        if (depth + 1 <= n) {
            int cnt = d[2*n-i-1][depth+1] << ((2*n-i-1-depth-1) / 2);
            if (cnt >= k) {
                ans += '(';    st.push('(');      depth++;    continue;
            }
            k -= cnt;
        }
        // ')'
        if (depth && st.top() == '(') {
            int cnt = d[2*n-i-1][depth-1] << ((2*n-i-1-depth+1) / 2);
            if (cnt >= k) {
                ans += ')';     st.pop();   depth--;    continue;
            }
            k -= cnt;
        }
        // '['
        if (depth + 1 <= n) {
            int cnt = d[2*n-i-1][depth+1] << ((2*n-i-1-depth-1) / 2);
            if (cnt >= k) {
                ans += '[';    st.push('[');      depth++;    continue;
            }
```

```
            k -= cnt;
        }
        // ']'
        ans += ']';      st.pop();    depth--;
    }
    return ans;
}
```

Burnside's Lemma

```
using Permutation = vector<int>;
void operator*=(Permutation& p, Permutation const& q) {
    Permutation copy = p;
    for (int i = 0; i < p.size(); i++)      p[i] = copy[q[i]];
}
int count_cycles(Permutation p) {
    int cnt = 0;
    for (int i = 0; i < p.size(); i++) {
        if (p[i] != -1) {
            cnt++;
            for (int j = i; p[j] != -1;) {
                int next = p[j];      p[j] = -1;   j = next;
            }
        }
    }
    return cnt;
}
int solve(int n, int m) {
    Permutation p(n*m), p1(n*m), p2(n*m), p3(n*m);
    for (int i = 0; i < n*m; i++) {
        p[i] = i;
        p1[i] = (i % n + 1) % n + i / n * n;
        p2[i] = (i / n + 1) % m * n + i % n;
        p3[i] = (m - 1 - i / n) * n + (n - 1 - i % n);
    }
    set<Permutation> s;
    for (int i1 = 0; i1 < n; i1++) {
        for (int i2 = 0; i2 < m; i2++) {
            for (int i3 = 0; i3 < 2; i3++) {
                s.insert(p);    p *= p3;
            }
            p *= p2;
        }
        p *= p1;
    }
    int sum = 0;
    for (Permutation const& p : s)    sum += 1 << count_cycles(p);
    return sum / s.size();
}
```

NCR

```
int ncr[1000][1000];
ll fac[MX];
int calcNCR() {
    for (int i = 1; i <= 1000; i++) {
        ncr[i][i] = ncr[i][0] = 1;
```

```
        for (int j = 1; j < i; j++)   ncr[i][j] = ncr[i-1][j] + ncr[i-1][j-1];
    }
}
void factorial() {
    fac[1] = 1;
    for (int i = 2; i < MX; i++)      fac[i] = (fac[i-1] * i) % MOD;
}
ll getNCR(int n, int r, int MOD) {
    ll res = fac[n];      res = (res * modInv(fac[n-r], MOD)) % MOD;
    res = (res * modInv(fac[r], MOD)) % MOD;
    return res;
}
bool next_combination(vector<int>& a, int n) {
    int k = (int)a.size();
    for (int i = k - 1; i >= 0; i--) {
        if (a[i] < n - k + i + 1) {
            a[i]++;
            for (int j = i + 1; j < k; j++)       a[j] = a[j - 1] + 1;
            return true;
        }
    }
    return false;
}
//  next permutation such that 1 digit differ
int gray_code (int n) {  return n ^ (n >> 1);      }
int count_bits (int n) {
    int res = 0;
    for (; n; n >>= 1)   res += n & 1;
    return res;
}
void all_combinations (int n, int k) {
    for (int i = 0; i < (1 << n); i++) {
        int cur = gray_code (i);
        if (count_bits(cur) == k) {
            for (int j = 0; j < n; j++)
                if (cur & (1 << j))   cout << j + 1;
            cout << "\n";
        }
    }
}
vector<int> ans;
void gen(int n, int k, int idx, bool rev) {
    if (k > n || k < 0)  return;
    if (!n) {
        for (int i = 0; i < idx; ++i)
            if (ans[i])  cout << i + 1;
        cout << "\n";
        return;
    }
    ans[idx] = rev;       gen(n - 1, k - rev, idx + 1, false);
    ans[idx] = !rev;      gen(n - 1, k - !rev, idx + 1, true);
}
void all_combinations(int n, int k) {
    ans.resize(n); gen(n, k, 0, false);
}
```

Prufer Code

```
//  node n occurs (d - 1) times in arra if node n has degree of d Tree to Prufer Code
//  Complexity: O(VlogV)
vector<int> treeToPrufercode(int nodes, vector<pair<int, int>> &edges) {
    unordered_set<int> neighbors[nodes + 1];
    for (int i = 0; i < edges.size(); i++) {
        pair<int, int> edge = edges[i];
        int u = edges[i].first, v = edges[i].second;
        neighbors[u].insert(v);       neighbors[v].insert(u);
    }
    priority_queue<int> leaves;
    for (int i = 0; i <= nodes; i++)
        if (neighbors[i].size() == 1)       leaves.push(-i);
    vector<int> pruferCode;
    int need = nodes - 2;
    while (need--) {
        int leaf = -leaves.top();     leaves.pop();
        int neighborOfLeaf = *(neighbors[leaf].begin());
        pruferCode.push_back(neighborOfLeaf);
        neighbors[neighborOfLeaf].erase(leaf);
        if (neighbors[neighborOfLeaf].size() == 1)      leaves.push(-neighborOfLeaf);
    }
    return pruferCode;
}
//  Prufer Code to Tree
//  Complexity: O(VlogV)
vector<pair<int, int>> pruferCodeToTree(vector<int> &pruferCode) {
    unordered_map<int, int> nodeCount;
    set<int> leaves;
    int len = pruferCode.size(), node = len + 2;
    for (int i = 0; i < len; i++) {
        int t = pruferCode[i];        nodeCount[t]++;
    }
    for (int i = 1; i <= node; i++)
        if (nodeCount.find(i) == nodeCount.end())       leaves.insert(i);
    vector<pair<int, int>> edges;
    for (int i = 0; i < len; i++) {
        int a = pruferCode[i], b = *leaves.begin();
        edges.push_back({a, b});      leaves.erase(b);
        nodeCount[a]--;
        if (nodeCount[a] == 0)        leaves.insert(a);
    }
    edges.push_back({*leaves.begin(), *leaves.rbegin()});
    return edges;
}
```

The Inclusion-Exclusion Principle

```
//  complexity : O(sqrt(n))
int solve (int n, int r) {
    vector<int> p;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            p.push_back (i);
            while (n % i == 0)        n /= i;
```

```
        }
    if (n > 1)     p.push_back (n);
    int sum = 0;
    for (int msk=1; msk<(1<<p.size()); ++msk) {
        int mult = 1, bits = 0;
        for (int i=0; i<(int)p.size(); ++i)
            if (msk & (1<<i)) {
                ++bits;         mult *= p[i];
            }
        int cur = r / mult;
        if (bits % 2 == 1)      sum += cur;
        else                    sum -= cur;
    }
    return r - sum;
}
int n;
bool good[MAXN], deg[MAXN], cnt[MAXN];
long long solve() {
    memset (good, 1, sizeof good);    memset (deg, 0, sizeof deg);
    memset (cnt, 0, sizeof cnt);
    long long ans_bad = 0;
    for (int i=2; i<=n; ++i) {
        if (good[i]) {
            if (deg[i] == 0)          deg[i] = 1;
            for (int j=1; i*j<=n; ++j) {
                if (j > 1 && deg[i] == 1)
                    if (j % i == 0)   good[i*j] = false;
                    else              ++deg[i*j];
                cnt[i*j] += (n / i) * (deg[i]%2==1 ? +1 : -1);
            }
        }
        ans_bad += (cnt[i] - 1) * 1ll * (n-1 - cnt[i]);
    }
    return (n-1) * 1ll * (n-2) * (n-3) / 6 - ans_bad / 2;
}
```

Theorem

Stars and Bars Theorem :

if [n,k>=0], the number of K-tuples of non-negative integers whose sum is N => (N+K-1)C(N) => (N+K-1)C(K-1). if (n>0 && k >0) : (N-1)C(K-1).

Theorem:

If we have K distinguishable containers and N indistinguishable balls, then we can distribute them in (N+K-1)C(N) ways.

Theorem:

if (n>0 && k >0), the number of K-tuples of positive integers whose sum is N => (N-1)C(K-1).

Theorem:

if (n>0 && k >0), the number of K-tuples of non-negative integers whose sum = N => (N+K)C(K).

Application of Prufer Code:

Random Tree Generation, Cayley's Formula, Building Tree from Degree Count number of spanning trees with N node is N^(N-2).

Data Structure

BIT

```
#define MAX 100005
int bit[MAX];
void update(int x, int v){
    while (x < MAX){
        bit[x] += v;     x += x & (-x);
    }
}
int query(int x){
    int res = 0;
    while (x){
        res += bit[x];          x -= x & (-x);
    }
    return res;
}
// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x){
    int LOGSZ = 17, N = (1 << LOGSZ), idx = 0, mask = N;
    while (mask && idx < N)    {
        int t = idx + mask;
        if (x >= bit[t]){
            idx = t;      x -= bit[t];
        }
        mask >>= 1;
    }
    return idx;
}
///** 2d BIT
ll bit[1030][1030];
int arr[1030][1030];
void update(int x, int y, int v){
    while (x <= n){
        int tmp = y;
        while (tmp <= n){
            bit[x][tmp] += v;    tmp += tmp & (-tmp);
        }
        x += x & (-x);
    }
}
ll query(int x, int y){
    ll res = 0;
    while (x){
        int tmp = y;
        while (tmp){
            res += bit[x][tmp];       tmp -= tmp & (-tmp);
        }
        x -= x & (-x);
    }
    return res;
```

```
}
```

Disjoint Set Union Find

```
class DisjointSet{
public:
    PII a[MX]; //index for value, first value for parent, second value for rank
    DisjointSet(int sz = MX){
        for (int i = 0; i < sz; i++){
            a[i].first = i;          a[i].second = 0;
        }
    }
    int FindSet(int n){
        int m = n;
        while (a[m].first != m)      m = a[m].first;
        a[n].first = m;
        return m;
    }
    void Union(int n1, int n2){
        int x = FindSet(n1), y = FindSet(n2);
        if (a[x] == a[y])                          return;
        else if (a[x].second < a[y].second)        swap(x, y);
        a[y].first = x;         a[x].second++;
    }
};
```

Dynamic Connectivity

```
// complexity : O(T(n)logn)
struct dsu_save {
    int v, rnkv, u, rnku;
    dsu_save() {   }
    dsu_save(int _v, int _rnkv, int _u, int _rnku) {
        v = _v; rnkv = _rnkv;   u = _u; rnku = _rnku;
    }
};
struct dsu_with_rollbacks {
    vector<int> p, rnk;
    int comps;
    stack<dsu_save> op;
    dsu_with_rollbacks() {     }
    dsu_with_rollbacks(int n) {
        p.resize(n);            rnk.resize(n);
        for (int i = 0; i < n; i++) {
            p[i] = i;    rnk[i] = 0;
        }
        comps = n;
    }
    int find_set(int v)  {     return (v == p[v]) ? v : find_set(p[v]);    }
    bool unite(int v, int u) {
        v = find_set(v);        u = find_set(u);
        if (v == u)      return false;
        comps--;
        if (rnk[v] > rnk[u])   swap(v, u);
        op.push(dsu_save(v, rnk[v], u, rnk[u]));
        p[v] = u;
        if (rnk[u] == rnk[v])         rnk[u]++;
```

```
        return true;
    }
    void rollback() {
        if (op.empty())        return;
        dsu_save x = op.top();        op.pop();
        comps++;
        p[x.v] = x.v;    rnk[x.v] = x.rnkv;
        p[x.u] = x.u;    rnk[x.u] = x.rnku;
    }
};
struct query {
    int v, u;      bool united;
    query(int _v, int _u)      {     v = _v; u = _u;    }
};
struct QueryTree {
    vector<vector<query>> t;
    dsu_with_rollbacks dsu;
    int T;
    QueryTree()    {     }
    QueryTree(int _T, int n) {
        T = _T;
        dsu = dsu_with_rollbacks(n);
        t.resize(4 * T + 4);
    }
    void add_to_tree(int v, int l, int r, int ul, int ur, query& q) {
        if (ul > ur)     return;
        if (l == ul && r == ur) {
            t[v].push_back(q);        return;
        }
        int mid = (l + r) / 2;
        add_to_tree(2 * v, l, mid, ul, min(ur, mid), q);
        add_to_tree(2 * v + 1, mid + 1, r, max(ul, mid + 1), ur, q);
    }
    void add_query(query q, int l, int r)   {     add_to_tree(1, 0, T - 1, l, r, q);    }
    void dfs(int v, int l, int r, vector<int>& ans) {
        for (query& q : t[v])  q.united = dsu.unite(q.v, q.u);
        if (l == r)      ans[l] = dsu.comps;
        else {
            int mid = (l + r) / 2;
            dfs(2 * v, l, mid, ans);
            dfs(2 * v + 1, mid + 1, r, ans);
        }
        for (query q : t[v])   if (q.united)      dsu.rollback();
    }
    vector<int> solve() {
        vector<int> ans(T);
        dfs(1, 0, T - 1, ans);
        return ans;
    }
};
```

Fenwick Tree

```
struct FenwickTree {
    vector<int> bit;  // binary indexed tree
    int n;
```

```
    FenwickTree(int n) {
        this->n = n;     bit.assign(n, 0);
    }
    FenwickTree(vector<int> a) : FenwickTree(a.size()) {
        for (size_t i = 0; i < a.size(); i++)     add(i, a[i]);
    }
    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)     ret += bit[r];
        return ret;
    }
    int sum(int l, int r)      {     return sum(r) - sum(l - 1);     }
    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))    bit[idx] += delta;
    }
};
struct FenwickTreeMin {
    vector<int> bit;
    int n;
    const int INF = (int)1e9;
    FenwickTreeMin(int n) {
        this->n = n;     bit.assign(n, INF);
    }
    FenwickTreeMin(vector<int> a) : FenwickTreeMin(a.size()) {
        for (size_t i = 0; i < a.size(); i++)     update(i, a[i]);
    }
    int getmin(int r) {
        int ret = INF;
        for (; r >= 0; r = (r & (r + 1)) - 1)     ret = min(ret, bit[r]);
        return ret;
    }
    void update(int idx, int val) {
        for (; idx < n; idx = idx | (idx + 1))    bit[idx] = min(bit[idx], val);
    }
};
struct FenwickTree2D {
    vector<vector<int>> bit;
    int n, m;
    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;
    }
};
struct FenwickTreeOneBasedIndexing {
    vector<int> bit;  // binary indexed tree
    int n;
    FenwickTreeOneBasedIndexing(int n) {
        this->n = n + 1;       bit.assign(n + 1, 0);
```

```
    }
    FenwickTreeOneBasedIndexing(vector<int> a)
        : FenwickTreeOneBasedIndexing(a.size()) {
        init(a.size());
        for (size_t i = 0; i < a.size(); i++)     add(i, a[i]);
    }
    int sum(int idx) {
        int ret = 0;
        for (++idx; idx > 0; idx -= idx & -idx)   ret += bit[idx];
        return ret;
    }
    int sum(int l, int r)      {     return sum(r) - sum(l - 1);    }
    void add(int idx, int delta) {
        for (++idx; idx < n; idx += idx & -idx)   bit[idx] += delta;
    }
};
//  range update point query
void add(int idx, int val) {
    for (++idx; idx < n; idx += idx & -idx)       bit[idx] += val;
}
void range_add(int l, int r, int val) {
    add(l, val);   add(r + 1, -val);
}
int point_query(int idx) {
    int ret = 0;
    for (++idx; idx > 0; idx -= idx & -idx)       ret += bit[idx];
    return ret;
}
```

LCA

```
#define MX 100005
vector<int> g[MX];
int level[MX], height, sparse[MX][22], parent[MX], visited[MX];
void dfs(int v, int u = -1, int lvl = 0) //for defining parent{
    visited[v] = 1;      parent[v] = u;
    level[v] = lvl;      height = max(height, lvl);
    for (int i = 0; i < g[v].size(); i++)
        if (!visited[g[v][i]])       dfs(g[v][i], v, lvl + 1);
}
void SparseTable(int n){
    for (int i = 0; i < n; i++)      sparse[i][0] = parent[i];
    for (int i = 1; (1 << i) < n; i++)
        for (int j = 0; j < n; j++)
            if (sparse[j][i - 1] != -1)     sparse[j][i] = sparse[sparse[j][i - 1]][i - 1];
            else                            sparse[j][i] = -1;
}
int findLCA(int u, int v, int n){
    if (level[u] > level[v])   swap(u, v); //so that level[u] is alawys smaller
    while (level[v] > level[u]){
        int k = log2(level[v] - level[u]);       v = sparse[v][k];
    }
    if (u == v)    return u;
    for (int i = height; i >= 0; i--)    {
        if (sparse[u][i] == sparse[v][i])   continue;
        u = sparse[u][i];      v = sparse[v][i];
```

```
    }
    return sparse[u][0];
}
int main(){
    for (int i = 0; i < n; i++)    {
        parent[i] = -1;  visited[i] = 0;
    }
    dfs(0);
    SparseTable(n);
    cout << findLCA(u, v, n) << endl;
}
```

MO

```
// *not fully ready
struct query {
    int l, r, t, id;
}q[MX];
struct update {
    int x, pre, now;
}u[MX];
const int k = 320;//    ceil(sqrt(MX));
bool cmp(query &a, query &b) {
    int l1 = a.l / k, l2 = b.l / k,
    r1 = a.r / k, r2 = b.r / k;
    if(l1 != l2)   return l1 < l2;
    if(r1 != r2)   return r1 < r2;
    return a.t < b.t;
}
int l = 0, r = -1, sum = 0, ans[MX], a[MX];
void apply(int x, int t) {
  if(l <= x && x <= r) {  // l, r is the l, r from MO's algo
    remove(x);     a[x] = y;    add(x);
  } else           a[x] = y;
}
void add(int x)    {     sum += a[x];       }
void remove(int x) {     sum += a[x];       }
int main(){
    int Q;   cin >> Q;
    for (int i = 0; i < Q; i++) {
        cin >> q[i].l >> q[i].r;     q[i].id = i;
    }
    sort(q, q+Q, cmp);
    int l = 0, r = -1, t = 0;
    int last[N];
    for(int i = 0; i < N; i++)       last[i] = a[i];
    for(int i = 0; i < Q; i++) {
    if( this is a query )
      store query {l, r, idx, id++} // idx is number of updates before, id is this query's i
d
        if( this is an update ) {
            u[++idx] = {x, last[x], y};     last[x] = y;
        }
    }
    for(int i = 0; i < Q; i++) {
        while(t < q[i].t)      t++, apply(u[t].x, u[t].now);
```

```
        while(t > q[i].t)       apply(u[t].x, u[y].pre), t--;
        while(l > q[i].l)       add(--l);
        while(r < q[i].r)       add(++r);
        while(l < q[i].l)       remove(l++);
        while(r > q[i].r)       remove(r--);
        ans[q[i].id] = some_variable;
    }
}
```

Ordered Set

```
#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp> // Including tree_order_statistics_node_update
using namespace __gnu_pbds;
typedef tree<
  int,
  null_type,
  less<int>,
  rb_tree_tag,
  tree_order_statistics_node_update>
ordered_set;
int main(){
    ordered_set X;
    X.insert(1);
    X.erase(8); /// delete 8 from where 8 is located
    cout<<"0 : "<<*X.find_by_order(0)<<endl; /// 2 same as X[1]
    cout<<"end@4 : "<<(X.end()==X.find_by_order(4))<<endl<<endl; // true
    cout<<"-5 : "<<X.order_of_key(-5)<<endl;   // 0 = lower bound
}
```

Palindrome Tree

```
//    Palindrome tree. Useful structure to deal with palindromes in strings. O(N)
//    This code counts number of palindrome substrings of the string.
const int MAXN = 105000;
struct node {
    int next[26], len, sufflink, num;
};
int len;
char s[MAXN];
node tree[MAXN];
int num;            // node 1 - root with len -1, node 2 - root with len 0
int suff;           // max suffix palindrome
long long ans;
bool addLetter(int pos) {
    int cur = suff, curlen = 0;
    int let = s[pos] - 'a';
    while (true) {
        curlen = tree[cur].len;
        if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos])  break;
        cur = tree[cur].sufflink;
    }
    if (tree[cur].next[let]) {
        suff = tree[cur].next[let];   return false;
    }
    num++;   suff = num;
```

```
    tree[num].len = tree[cur].len + 2;
    tree[cur].next[let] = num;
    if (tree[num].len == 1) {
        tree[num].sufflink = 2;       tree[num].num = 1;       return true;
    }
    while (true) {
        cur = tree[cur].sufflink;
        curlen = tree[cur].len;
        if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
            tree[num].sufflink = tree[cur].next[let];   break;
        }
    }
    tree[num].num = 1 + tree[tree[num].sufflink].num;
    return true;
}
void initTree() {
    num = 2;       suff = 2;
    tree[1].len = -1;          tree[1].sufflink = 1;
    tree[2].len = 0;           tree[2].sufflink = 1;
}
int main() {
    gets(s);
    len = strlen(s);
    initTree();
    for (int i = 0; i < len; i++) {
        addLetter(i);    ans += tree[suff].num;
    }
    cout << ans << endl;
}
```

persistent Segment Tree

```
#include <bits/stdc++.h>
using namespace std;
#define IN freopen("in.txt", "r", stdin);
#define OUT freopen("out.txt", "w", stdout);
#define ll long long int
#define PII pair <int, int>
#define MX 100001
#define EPS 1e-9
#define MOD 1000000007
#define PI 2.0 * acos(0.0)
struct node {
    node *left, *right;
    int val;
    node(int a = 0, node *b = NULL, node *c = NULL) {
        val = a;    left = b;   right = c;
    }
    void build(int l, int r) {
        if (l == r)      return;
        left = new node();      right = new node();
        int mid = (l + r) >> 1;
        left->build(l, mid);   right->build(mid + 1, r);
    }
    node* update(int l, int r, int idx, int v) {
        if (r < idx || l > idx)      return this;
```

```
        else if (l == r)            return new node(val + v, left, right);
        int mid = (l + r) >> 1;
        node *ret = new node(val);
        ret->left = left->update(l, mid, idx, v);
        ret->right = right->update(mid+1, r, idx, v);
        ret->val = ret->left->val + ret->right->val;
        return ret;
    }
    //  [l, r] node range & [i, j] query range
    int query(int l, int r, int i, int j) {
        if (r < i || l > j)    return 0;
        else if (i <= l && r <= j)    return val;
        int mid = (l + r) >> 1;
        int ret = left->query(l, mid, i, j) + right->query(mid+1, r, i, j);
        return ret;
    }
} *root[MX];
int main(){
    int n = MX:
    root[0] = new node();       root[0]->build(0, n - 1);
    root[1] = root[0]->update(0, n-1, 4, 6); //update value of 4th index with 6
}
```

RMQ-1D

```
#define SIZE 8
int a[SIZE] = {3, 6, 2, -1, 0, 3, 1, 5}, sparse[SIZE][22], height;
int buildTable(int a[], int n) //time : o(nlogn){
    for (int i = 0; i < n; i++)       sparse[i][0] = a[i];
    for (int i = 1; (1 << i) <= n; i++){
        height = i;
        for (int j = 0; j < n; j++){
            int k = j + (1 << (i - 1));
            if (k >= n)         k = n - 1;
            sparse[j][i] = min(sparse[j][i - 1], sparse[k][i - 1]);
        }
    }
}
int rmq(int i, int j) //0 indexed & time : O(1){
    int len = j - i + 1, l = -1;
    while (len){
        len = len >> 1;         l++;
    }
    int minn = min(sparse[i][l], sparse[j - (1 << l) + 1][l]);
    return minn;
}
```

RMQ-2D

```
//    not done
int a[SIZE][SIZE], sparse[SIZE][SIZE][22], height;
int buildTable(int n) //time : o(nlogn){
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)   sparse[i][j][0] = a[i][j];
    for (int i = 1; (1 << i) <= n; i++)    {
        height = i;
        for (int j = 0; j < n; j++)
```

```
            for (int k = 0; k < n; k++){
                int l = j + (1 << (i - 1));
                if (l >= n)           l = n - 1;
                sparse[j][i] = min(sparse[j][i - 1], sparse[l][i - 1]);
            }
    }
}
int rmq(int i, int j) //0 indexed & time : O(1){
    int len = j - i + 1, l = -1;
    while (len){
        len = len >> 1;         l++;
    }
    int minn = min(sparse[i][l], sparse[j - (1 << l) + 1][l]);
    return minn;
}
```

segmented tree

```
int a[MX], SegTree[4 * MX], Lazy[4 * MX];
void init(int low, int high, int pos = 0) //    O(n){
    if (low == high){
        SegTree[pos] = a[low];        return;
    }
    int mid = (low + high) / 2;
    init(low, mid, 2 * pos + 1);      init(mid + 1, high, 2 * pos + 2);
    SegTree[pos] = min(SegTree[2 * pos + 1], SegTree[2 * pos + 2]);
}
int Query(int low, int high, int Qlow, int Qhigh, int pos = 0) //O(logn){
    if (Lazy[pos]){
        SegTree[pos] += Lazy[pos];
        if (low != high) //not a leaf node        {
            Lazy[2 * pos + 1] += Lazy[pos];       Lazy[2 * pos + 2] += Lazy[pos];
        }
        Lazy[pos] = 0;
    }
    if (Qlow > high || Qhigh < low || low > high)       return INT_MAX;
    if (Qlow <= low && Qhigh >= high)                   return SegTree[pos];
    int mid = (low + high) / 2;
    int x = Query(low, mid, Qlow, Qhigh, 2 * pos + 1);
    int y = Query(mid + 1, high, Qlow, Qhigh, 2 * pos + 2);
    return min(x, y);
}
void Update(int low, int high, int Qlow, int Qhigh, int val, int pos = 0) //O(logn){
    if (Lazy[pos]){
        SegTree[pos] += Lazy[pos];
        if (low != high) //not a leaf node{
            Lazy[2 * pos + 1] += Lazy[pos];       Lazy[2 * pos + 2] += Lazy[pos];
        }
        Lazy[pos] = 0;
    }
    if (Qlow > high || Qhigh < low || low > high)       return;
    if (Qlow <= low && Qhigh >= high){
        SegTree[pos] += val;
        if (low != high) //not a leaf node{
            Lazy[2 * pos + 1] += val;       Lazy[2 * pos + 2] += val;
        }
```

```
        return;
    }
    int mid = (low + high) / 2;
    Update(low, mid, Qlow, Qhigh, val, 2 * pos + 1);
    Update(mid + 1, high, Qlow, Qhigh, val, 2 * pos + 2);
    SegTree[pos] = min(SegTree[2 * pos + 1], SegTree[2 * pos + 2]);
}
```

Sqrt Decomposition

```
int main() {
    int n;
    vector<int> a (n);
    int len = (int) sqrt (n + .0) + 1; // size of the block and the number of blocks
    vector<int> b (len);
    for (int i=0; i<n; ++i) b[i / len] += a[i];
    for (;;) {
        int l, r, sum = 0;
        for (int i=l; i<=r; )
            if (i % len == 0 && i + len - 1 <= r) {
                sum += b[i / len];          i += len;
            }
            else {
                sum += a[i];                ++i;
            }
    }
    int sum = 0, c_l = l / len,   c_r = r / len;
    if (c_l == c_r)
        for (int i=l; i<=r; ++i)    sum += a[i];
    else {
        for (int i=l, end=(c_l+1)*len-1; i<=end; ++i)   sum += a[i];
        for (int i=c_l+1; i<=c_r-1; ++i)                sum += b[i];
        for (int i=c_r*len; i<=r; ++i)                  sum += a[i];
    }
}
```

sqrt Tree

```
SqrtTreeItem op(const SqrtTreeItem &a, const SqrtTreeItem &b);
inline int log2Up(int n) {
    int res = 0;
    while ((1 << res) < n)     res++;
    return res;
}
class SqrtTree {
private:
    int n, lg, indexSz;
    vector<SqrtTreeItem> v;
    vector<int> clz, layers, onLayer;
    vector< vector<SqrtTreeItem> > pref, suf, between;
    inline void buildBlock(int layer, int l, int r) {
        pref[layer][l] = v[l];
        for (int i = l+1; i < r; i++)       pref[layer][i] = op(pref[layer][i-1], v[i]);
        suf[layer][r-1] = v[r-1];
        for (int i = r-2; i >= l; i--)      suf[layer][i] = op(v[i], suf[layer][i+1]);
    }
    inline void buildBetween(int layer, int lBound, int rBound, int betweenOffs) {
```

```
    int bSzLog = (layers[layer]+1) >> 1,  bCntLog = layers[layer] >> 1;
    int bSz = 1 << bSzLog, bCnt = (rBound - lBound + bSz - 1) >> bSzLog;
    for (int i = 0; i < bCnt; i++) {
        SqrtTreeItem ans;
        for (int j = i; j < bCnt; j++) {
            SqrtTreeItem add = suf[layer][lBound + (j << bSzLog)];
            ans = (i == j) ? add : op(ans, add);
            between[layer-1][betweenOffs + lBound + (i << bCntLog) + j] = ans;
        }
    }
}
inline void buildBetweenZero() {
    int bSzLog = (lg+1) >> 1;
    for (int i = 0; i < indexSz; i++)   v[n+i] = suf[0][i << bSzLog];
    build(1, n, n + indexSz, (1 << lg) - n);
}
inline void updateBetweenZero(int bid) {
    int bSzLog = (lg+1) >> 1;     v[n+bid] = suf[0][bid << bSzLog];
    update(1, n, n + indexSz, (1 << lg) - n, n+bid);
}
void build(int layer, int lBound, int rBound, int betweenOffs) {
    if (layer >= (int)layers.size())    return;
    int bSz = 1 << ((layers[layer]+1) >> 1);
    for (int l = lBound; l < rBound; l += bSz) {
        int r = min(l + bSz, rBound);
        buildBlock(layer, l, r);        build(layer+1, l, r, betweenOffs);
    }
    if (layer == 0)         buildBetweenZero();
    else                    buildBetween(layer, lBound, rBound, betweenOffs);
}
void update(int layer, int lBound, int rBound, int betweenOffs, int x) {
    if (layer >= (int)layers.size())    return;
    int bSzLog = (layers[layer]+1) >> 1, bSz = 1 << bSzLog;
    int blockIdx = (x - lBound) >> bSzLog;
    int l = lBound + (blockIdx << bSzLog), r = min(l + bSz, rBound);
    buildBlock(layer, l, r);
    if (layer == 0)  updateBetweenZero(blockIdx);
    else             buildBetween(layer, lBound, rBound, betweenOffs);
    update(layer+1, l, r, betweenOffs, x);
}
inline SqrtTreeItem query(int l, int r, int betweenOffs, int base) {
    if (l == r)             return v[l];
    if (l + 1 == r)         return op(v[l], v[r]);
    int layer = onLayer[clz[(l - base) ^ (r - base)]];
    int bSzLog = (layers[layer]+1) >> 1, bCntLog = layers[layer] >> 1;
    int lBound = (((l - base) >> layers[layer]) << layers[layer]) + base;
    int lBlock = ((l - lBound) >> bSzLog) + 1, rBlock = ((r - lBound) >> bSzLog) - 1;
    SqrtTreeItem ans = suf[layer][l];
    if (lBlock <= rBlock) {
        SqrtTreeItem add = (layer == 0) ? ((n + lBlock, n + rBlock, (1 << lg) - n, n))
         :(between[layer-1][betweenOffs + lBound + (lBlock << bCntLog) + rBlock]);
        ans = op(ans, add);
    }
    ans = op(ans, pref[layer][r]);
    return ans;
}
```

```
public:
    inline SqrtTreeItem query(int l, int r){      return query(l, r, 0, 0);      }
    inline void update(int x, const SqrtTreeItem &item) {
        v[x] = item;     update(0, 0, n, 0, x);
    }
    SqrtTree(const vector<SqrtTreeItem>& a)
        : n((int)a.size()), lg(log2Up(n)), v(a), clz(1 << lg), onLayer(lg+1) {
        clz[0] = 0;
        for (int i = 1; i < (int)clz.size(); i++)       clz[i] = clz[i >> 1] + 1;
        int tlg = lg;
        while (tlg > 1) {
            onLayer[tlg] = (int)layers.size();
            layers.push_back(tlg);          tlg = (tlg+1) >> 1;
        }
        for (int i = lg-1; i >= 0; i--)     onLayer[i] = max(onLayer[i], onLayer[i+1]);
        int betweenLayers = max(0, (int)layers.size() - 1);
        int bSzLog = (lg+1) >> 1, bSz = 1 << bSzLog;
        indexSz = (n + bSz - 1) >> bSzLog;
        v.resize(n + indexSz);
        pref.assign(layers.size(), vector<SqrtTreeItem>(n + indexSz));
        suf.assign(layers.size(), vector<SqrtTreeItem>(n + indexSz));
        between.assign(betweenLayers, vector<SqrtTreeItem>((1 << lg) + bSz));
        build(0, 0, n, 0);
    }
};
```

Treap (Cartesian tree)

```
struct item {
    int key, prior;
    item * l, * r;
    item() { }
    item (int key, int prior) : key(key), prior(prior), l(NULL), r(NULL) { }
};
typedef item * pitem;
void split (pitem t, int key, pitem & l, pitem & r) {
    if (!t)                     l = r = NULL;
    else if (key < t->key)     split (t->l, key, l, t->l),  r = t;
    else                       split (t->r, key, t->r, r),  l = t;
}
void insert (pitem & t, pitem it) {
    if (!t)                            t = it;
    else if (it->prior > t->prior)    split (t, it->key, it->l, it->r),  t = it;
    else                               insert (it->key < t->key ? t->l : t->r, it);
}
void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)                      t = l ? l : r;
    else if (l->prior > r->prior)     merge (l->r, l->r, r),  t = l;
    else                               merge (r->l, l, r->l),  t = r;
}
void erase (pitem & t, int key) {
    if (t->key == key)   merge (t, t->l, t->r);
    else                 erase (key < t->key ? t->l : t->r, key);
}
pitem unite (pitem l, pitem r) {
    if (!l || !r)              return l ? l : r;
```

```
    if (l->prior < r->prior)   swap (l, r);
    pitem lt, rt;
    split (r, l->key, lt, rt);
    l->l = unite (l->l, lt);   l->r = unite (l->r, rt);
    return l;
}
int cnt (pitem t){ return t ? t->cnt : 0;   }
void upd_cnt (pitem t) {       if (t)      t->cnt = 1 + cnt(t->l) + cnt (t->r);  }
void heapify (pitem t) {//  O(n) offline
    if (!t)        return;
    pitem max = t;
    if (t->l != NULL && t->l->prior > max->prior)       max = t->l;
    if (t->r != NULL && t->r->prior > max->prior)       max = t->r;
    if (max != t) {
        swap (t->prior, max->prior);        heapify (max);
    }
}
pitem build (int * a, int n) { // Construct a treap on values {a[0], a[1], ..., a[n - 1]}
    if (n == 0)          return NULL;
    int mid = n / 2;
    pitem t = new item (a[mid], rand ());
    t->l = build (a, mid);     t->r = build (a + mid + 1, n - mid - 1);
    heapify (t);
    return t;
}
```

implicit treap

```
void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)                    t = l ? l : r;
    else if (l->prior > r->prior)    merge (l->r, l->r, r),  t = l;
    else                             merge (r->l, l, r->l),  t = r;
    upd_cnt (t);
}
void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)        return void( l = r = 0 );
    int cur_key = add + cnt(t->l); //implicit key
    if (key <= cur_key)  split (t->l, l, t->l, key, add),  r = t;
    else                 split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
    upd_cnt (t);
}
typedef struct item * pitem;
struct item {
    int prior, value, cnt;          bool rev;          pitem l, r;
};
int cnt (pitem it) {     return it ? it->cnt : 0; }
void upd_cnt (pitem it) {      if (it)      it->cnt = cnt(it->l) + cnt(it->r) + 1;      }
void push (pitem it) {
    if (it && it->rev) {
        it->rev = false;       swap (it->l, it->r);
        if (it->l)       it->l->rev ^= true;
        if (it->r)       it->r->rev ^= true;
    }
}
void merge (pitem & t, pitem l, pitem r) {
    push (l);            push (r);
    if (!l || !r)                    t = l ? l : r;
    else if (l->prior > r->prior)    merge (l->r, l->r, r),  t = l;
```

```
    else                            merge (r->l, l, r->l),  t = r;
    upd_cnt (t);
}
void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)         return void( l = r = 0 );
    push (t);
    int cur_key = add + cnt(t->l);
    if (key <= cur_key)  split (t->l, l, t->l, key, add),  r = t;
    else                 split (t->r, t->r, r, key, add + 1 + cnt(t->l)),  l = t;
    upd_cnt (t);
}
void reverse (pitem t, int l, int r) {
    pitem t1, t2, t3;
    split (t, t1, t2, l);       split (t2, t2, t3, r-l+1);
    t2->rev ^= true;
    merge (t, t1, t2);          merge (t, t, t3);
}
void output (pitem t) {
    if (!t)         return;
    push (t);       output (t->l);
    printf ("%d ", t->value);  output (t->r);
}
```

Trie

```
#define MX 26
struct node{
    bool end;             node *next[MX];
    node(){
        end = 0;
        for (int i = 0; i < MX; i++)        next[i] = NULL;
    }
};
class trie{
public:
    node *root;
    trie() {        root = new node(); }
    void Insert(string s){
        node *cur = root;
        for (int i = 0; i < s.length(); i++){
            int id = s[i] - 'a';
            if (cur->next[id] == NULL)       cur->next[id] = new node();
            cur = cur->next[id];
        }
        cur->end = 1;
    }
    bool Find(string s){
        node *cur = root;
        for (int i = 0; i < s.length(); i++){
            int id = s[i] - 'a';
            if (cur->next[id] == NULL)      return 0;
            cur = cur->next[id];
        }
        return cur->end;
    }
    void del(node *cur) {
```

```
        for (int i = 0; i < MX; i++)
            if (cur->next[i])  del(cur->next[i]);
        delete (cur);
    }
};
```

Dynamic Programming

DP

Riaz Vai

g one high low

```
int A[] = {2, 0}, B[] = {2, 9}, n = 2, dp[20][2][2][220], visited[20][2][2][220];
bool isprime(int n) {
    for (int i = 2; i * i <= n; i++)
        if (n % i == 0)        return false;
    return true;
}
int func(int pos, int high_flag, int low_flag, int sum) {
    if (pos == n)        return isprime(sum);
    if (visited[pos][high_flag][low_flag][sum])   return dp[pos][high_flag][low_flag][sum];
    visited[pos][high_flag][low_flag][sum] = 1;
    int lo = low_flag ? A[pos] : 0,  hi = high_flag ? B[pos] : 9, cnt = 0;
    for (int i = lo; i <= hi; i++)
      cnt += func(pos+1, high_flag&(i==hi), low_flag&(i==lo), sum+i);

    return dp[pos][high_flag][low_flag][sum] = cnt;
}
func(0,1,1,0);
```

one high

```
int A[] = {2, 5, 9}, n = 3;
int dp[20][2][220], visited[20][2][220];
bool isprime(int n) {
    for (int i = 2; i * i <= n; i++)
        if (n % i == 0)        return false;
    return true;
}
int func(int pos, int high_flag, int sum) {
    if (pos == n)        return isprime(sum);
    if (visited[pos][high_flag][sum])      return dp[pos][high_flag][sum];
    visited[pos][high_flag][sum] = 1;
    int lo = 0, hi = high_flag ? A[pos] : 9, cnt = 0;
    for (int i = lo; i <= hi; i++)    cnt += func(pos+1, high_flag&(i==hi), sum+i);
    return dp[pos][high_flag][sum] = cnt;
}
func(0,1,0)
```

cheapest palindrome

```
const int MAX=2010;
char A[MAX];
int insrt[300], dlt[300], dp[MAX][MAX];
bool visited[MAX][MAX];
```

```
int call(int l, int r){
    if (l>=r)           return 0;
    if (visited[l][r])  return dp[l][r];
    visited[l][r] = 1;
    int a=inf, b=inf, c=inf, d=inf;
    if (A[l]==A[r])     a=call(l+1, r-1);
    else{
        a=dlt[A[l]]+call(l+1, r);          b=dlt[A[r]]+call(l, r-1);
        c=insrt[A[r]]+call(l, r-1);        d=insrt[A[l]]+call(l+1, r);
    }
    return dp[l][r]=min(min(a, b), min(c, d));
}
```

Divide and Conquer DP

```
int n;      long long C(int i, int j);
vector<long long> dp_before(n), dp_cur(n);
void compute(int l, int r, int optl, int optr){//compute dp_cur[l], ... dp_cur[r] (inclusive)
    if (l > r)     return;
    int mid = (l + r) >> 1;
    pair<long long, int> best = {INF, -1};
    for (int k = optl; k <= min(mid, optr); k++)
      best = min(best, {dp_before[k] + C(k, mid), k});
    dp_cur[mid] = best.first;
    int opt = best.second;
    compute(l, mid - 1, optl, opt);        compute(mid + 1, r, opt, optr);
}
```

dp broken profile Parquet

```
// Problem description: Given a grid of size N×M. Find number of ways to fill the grid with
// figures of size 2×1 (no cell should be left unfilled, and figures should not overlap each
// other).
int n, m;
vector < vector<long long> > d;
void calc (int x = 0, int y = 0, int mask = 0, int next_mask = 0){
    if (x == n)    return;
    if (y >= m)    d[x+1][next_mask] += d[x][mask];
    else{
        int my_mask = 1 << y;
        if (mask & my_mask)    calc (x, y+1, mask, next_mask);
        else{
            calc (x, y+1, mask, next_mask | my_mask);
            if (y+1 < m && ! (mask & my_mask) && ! (mask & (my_mask << 1)))
                calc (x, y+2, mask, next_mask);
        }
    }
}
int main(){
    d.resize (n+1, vector<long long> (1<<m));
    d[0][0] = 1;
    for (int x=0; x<n; ++x)
        for (int mask=0; mask<(1<<m); ++mask)      calc (x, 0, mask, 0);
    cout << d[n][0];
}
```

Kadane's algorithm

```
int MS(){
    int maxi = INT_MIN, s = 0;
    for (int ii = 0; ii < n; ii++){
        maxi = max(maxi, s + dp[ii]);        s = max(0, s + dp[ii]);
    }
    return maxi;
}
PII MS(){
    int maxi = INT_MIN, s = 0;         PII answer;
    for (int ii = 0; ii < n; ii++){
        cin >> a[ii];    s += a[ii];
        if (maxi <= s) {
            maxi = s;    answer.first = l;        answer.second = ii;
        }
        if (s < 0) {
            s = 0;       l = ii + 1;
        }
    }
    return answer;
}
int MS(int n, int m) {
    int maxi = INT_MIN, s = 0, l = 0;
    PII p;         p.first = 0;       p.second = 0;
    FOR(i, 0, n - 1) {
        cin >> a[i];     s += a[i];
        while (p.first <= i && s > m) {
            s -= a[l];         l++;
        }
        if (s <= m && maxi <= s) {
            maxi = s;    p.first = l;       p.second = i;
        }
        if (s < 0) {
            s = 0;       l = i + 1;
        }
    }
    return maxi;
}
```

largest zero submatrix

```
// You are given a matrix with n rows and m columns.  Find the largest submatrix consisting
// of only zeros (a submatrix is a rectangular area of the matrix).
int zero_matrix(vector<vector<int>> a) {
    int n = a.size(), m = a[0].size(), ans = 0;
    vector<int> d(m, -1), d1(m), d2(m);
    stack<int> st;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j)
            if (a[i][j] == 1)        d[j] = i;
        for (int j = 0; j < m; ++j) {
            while (!st.empty() && d[st.top()] <= d[j])  st.pop();
            d1[j] = st.empty() ? -1 : st.top();
            st.push(j);
        }
```

```
        while (!st.empty())    st.pop();
        for (int j = m - 1; j >= 0; --j) {
            while (!st.empty() && d[st.top()] <= d[j])  st.pop();
            d2[j] = st.empty() ? m : st.top();
            st.push(j);
        }
        while (!st.empty())    st.pop();
        for (int j = 0; j < m; ++j)  ans = max(ans, (i - d[j]) * (d2[j] - d1[j] - 1));
    }
    return ans;
}
```

Geometry

Circle

```
#define D double
#define PDD pair<double, double>
#define PI 2.0 * acos(0.0)
struct point{ //(x, y)
    D x, y;
    point(D _x = 0, D _y = 0){
        x = _x;    y = _y;
    }
};
struct QEQ{ //1 variable        ax^2+bx+c=0
    D a, b, c;
    EQ(D _a = 0, D _b = 0, D _c = 0){
        a = _a;    b = _b;      c = _c;
    }
};
struct EQ3{ //3 variable        ax+by+cz+d=0
    D a, b, c, d;
    EQ3(D _a = 0, D _b = 0, D _c = 0, D _d = 0){
        a = _a;    b = _b;      c = _c;      d = _d;
    }
}
struct circleEQ{ //x^2+y^2+2gx+2fy+c=0
    D g, f, c;
    circleEQ(D _g = 0, D _f = 0, D _c = 0){
        g = _g;    f = _f;      c = _c;
    }
    D radius(){    return sqrt(g * g + f * f - c);}
    D area() {
        D r = radius();
        return PI * r * r;
    }
    D AngleInCenter(point p1, point p2) {
        p1.x += g;       p2.x += g;         p1.y += f;         p2.y += f;
        D dot = p1.x *p2.x + p1.y *p2.y //a.b
        D val = sqrt(p1.x * p1.x + p1.y * p1.y) * sqrt(p2.x * p2.x + p2.y * p2.y);//|a| |b|
        return acos(dot / val); //theta=cos-1 ( a.b / |a||b|)
    }
    void CreateEQ(point p1, point p2, point p3) {}
```

```
    int PositionOfPoint(point p) { //1 for outside, 0 for on the line, -1 for inside circle
        int v = p.x * p.x + p.y * p.y + 2 * g * p.x + 2 * f * p.y + c;//EQ of circlr @point p
        if (v > 0)       return 1;
        else if (v == 0) return 0;
        else             return -1;
    }
};
```

convex hull

```
// construction using graham's scan

struct pt {  double x, y; };
bool cmp(pt a, pt b) {   return a.x < b.x || (a.x == b.x && a.y < b.y);    }
bool cw(pt a, pt b, pt c) {    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) < 0;    }
bool ccw(pt a, pt b, pt c) {   return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) > 0;    }
void convex_hull(vector<pt>& a) {
    if (a.size() == 1)          return;
    sort(a.begin(), a.end(), &cmp);
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;        up.push_back(p1);        down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2)) {
            while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1], a[i]))
                up.pop_back();
            up.push_back(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i], p2)) {
            while(down.size() >= 2 && !ccw(down[down.size()-2], down[down.size()-1], a[i]))
                down.pop_back();
            down.push_back(a[i]);
        }
    }
    a.clear();
    for (int i = 0; i < (int)up.size(); i++)      a.push_back(up[i]);
    for (int i = down.size() - 2; i > 0; i--)     a.push_back(down[i]);
}
//  trick & li chao tree
typedef int ftype;
typedef complex<ftype> point;
#define x real
#define y imag
ftype dot(point a, point b) {  return (conj(a) * b).x();      }
ftype cross(point a, point b) {      return (conj(a) * b).y();      }
vector<point> hull, vecs;
void add_line(ftype k, ftype b) {
    point nw = {k, b};
    while(!vecs.empty() && dot(vecs.back(), nw - hull.back()) < 0) {
        hull.pop_back();        vecs.pop_back();
    }
    if(!hull.empty())    vecs.push_back(1i * (nw - hull.back()));
    hull.push_back(nw);
}
int get(ftype x) {
    point query = {x, 1};
    auto it = lower_bound(vecs.begin(), vecs.end(), query, [](point a, point b) {
```

```
        return cross(a, b) > 0;
    });
    return dot(query, hull[it - vecs.begin()]);
}
//  li chao tree
typedef int ftype;
typedef complex<ftype> point;
#define x real     #define y imag
ftype dot(point a, point b) {   return (conj(a) * b).x();       }
ftype f(point a,  ftype x) {    return dot(a, {x, 1});    }
const int maxn = 2e5;
point line[4 * maxn];
void add_line(point nw, int v = 1, int l = 0, int r = maxn) {
    int m = (l + r) / 2;
    bool lef = f(nw, l) < f(line[v], l),  mid = f(nw, m) < f(line[v], m);
    if(mid)              swap(line[v], nw);
    if(r - l == 1)             return;
    else if(lef != mid)        add_line(nw, 2 * v, l, m);
    else                       add_line(nw, 2 * v + 1, m, r);
}
int get(int x, int v = 1, int l = 0, int r = maxn) {
    int m = (l + r) / 2;
    if(r - l == 1)       return f(line[v], x);
    else if(x < m)       return min(f(line[v], x), get(x, 2 * v, l, m));
    else                 return min(f(line[v], x), get(x, 2 * v + 1, m, r));
}
```

Delaunay triangulation and Voronoi diagram

```
//  Delaunay triangulation and Voronoi diagram
typedef long long ll;
bool ge(const ll& a, const ll& b) { return a >= b; }
bool le(const ll& a, const ll& b) { return a <= b; }
bool eq(const ll& a, const ll& b) { return a == b; }
bool gt(const ll& a, const ll& b) { return a > b; }
bool lt(const ll& a, const ll& b) { return a < b; }
int sgn(const ll& a) { return a >= 0 ? a ? 1 : 0 : -1; }
struct pt {
    ll x, y;
    pt() { }
    pt(ll _x, ll _y) : x(_x), y(_y) { }
    pt operator-(const pt& p) const { return pt(x - p.x, y - p.y);    }
    ll cross(const pt& p) const {     return x * p.y - y * p.x;       }
    ll cross(const pt& a, const pt& b) const {    return (a - *this).cross(b - *this);  }
    ll dot(const pt& p) const {       return x * p.x + y * p.y;       }
    ll dot(const pt& a, const pt& b) const {      return (a - *this).dot(b - *this);    }
    ll sqrLength() const {     return this->dot(*this); }
    bool operator==(const pt& p) const {    return eq(x, p.x) && eq(y, p.y);      }
};
const pt inf_pt = pt(1e18, 1e18);
struct QuadEdge {
    pt origin;
    QuadEdge* rot = nullptr, *onext = nullptr;
    bool used = false;
    QuadEdge* rev() const {     return rot->rot;   }
    QuadEdge* lnext() const {   return rot->rev()->onext->rot; }
```

```
    QuadEdge* oprev() const {  return rot->onext->rot;  }
    pt dest() const {    return rev()->origin;    }
};
QuadEdge* make_edge(pt from, pt to) {
    QuadEdge* e1 = new QuadEdge, *e2 = new QuadEdge, *e3 = new QuadEdge, *e4 = new QuadEdge;
    e1->origin = from;   e2->origin = to;   e3->origin = e4->origin = inf_pt;
    e1->rot = e3;  e2->rot = e4;      e3->rot = e2;      e4->rot = e1;
    e1->onext = e1;      e2->onext = e2;    e3->onext = e4;    e4->onext = e3;
    return e1;
}
void splice(QuadEdge* a, QuadEdge* b) {
    swap(a->onext->rot->onext, b->onext->rot->onext);   swap(a->onext, b->onext);
}
void delete_edge(QuadEdge* e) {
    splice(e, e->oprev());     splice(e->rev(), e->rev()->oprev());
    delete e->rot;       delete e->rev()->rot;    delete e;    delete e->rev();
}
QuadEdge* connect(QuadEdge* a, QuadEdge* b) {
    QuadEdge* e = make_edge(a->dest(), b->origin);
    splice(e, a->lnext());     splice(e->rev(), b);
    return e;
}
bool left_of(pt p, QuadEdge* e) {    return gt(p.cross(e->origin, e->dest()), 0);      }
bool right_of(pt p, QuadEdge* e) {   return lt(p.cross(e->origin, e->dest()), 0);      }
template <class T>
T det3(T a1, T a2, T a3, T b1, T b2, T b3, T c1, T c2, T c3) {
    return a1 * (b2 * c3 - c2 * b3) - a2 * (b1 * c3 - c1 * b3) + a3 * (b1 * c2 - c1 * b2);
}
bool in_circle(pt a, pt b, pt c, pt d) {
// If there is __int128, calculate directly. Otherwise, calculate angles.
#if defined(__LP64__) || defined(_WIN64)
    __int128 det = -det3<__int128>(b.x, b.y, b.sqrLength(), c.x, c.y,
                                   c.sqrLength(), d.x, d.y, d.sqrLength());
    det += det3<__int128>(a.x, a.y, a.sqrLength(), c.x, c.y, c.sqrLength(), d.x,
                          d.y, d.sqrLength());
    det -= det3<__int128>(a.x, a.y, a.sqrLength(), b.x, b.y, b.sqrLength(), d.x,
                          d.y, d.sqrLength());
    det += det3<__int128>(a.x, a.y, a.sqrLength(), b.x, b.y, b.sqrLength(), c.x,
                          c.y, c.sqrLength());
    return det > 0;
#else
    auto ang = [](pt l, pt mid, pt r) {
        ll x = mid.dot(l, r), y = mid.cross(l, r);
        long double res = atan2((long double)x, (long double)y);
        return res;
    };
    long double kek = ang(a, b, c) + ang(c, d, a) - ang(b, c, d) - ang(d, a, b);
    if (kek > 1e-8)      return true;
    else                 return false;
#endif
}
pair<QuadEdge*, QuadEdge*> build_tr(int l, int r, vector<pt>& p) {
    if (r - l + 1 == 2) {
        QuadEdge* res = make_edge(p[l], p[r]);    return make_pair(res, res->rev());
    }
    if (r - l + 1 == 3) {
```

```
        QuadEdge *a = make_edge(p[l], p[l + 1]), *b = make_edge(p[l + 1], p[r]);
        splice(a->rev(), b);
        int sg = sgn(p[l].cross(p[l + 1], p[r]));
        if (sg == 0)     return make_pair(a, b->rev());
        QuadEdge* c = connect(b, a);
        if (sg == 1)     return make_pair(a, b->rev());
        else             return make_pair(c->rev(), c);
    }
    int mid = (l + r) / 2;
    QuadEdge *ldo, *ldi, *rdo, *rdi;
    tie(ldo, ldi) = build_tr(l, mid, p);    tie(rdi, rdo) = build_tr(mid + 1, r, p);
    while (true) {
        if (left_of(rdi->origin, ldi)) {
            ldi = ldi->lnext();      continue;
        }
        if (right_of(ldi->origin, rdi)) {
            rdi = rdi->rev()->onext;        continue;
        }
        break;
    }
    QuadEdge* basel = connect(rdi->rev(), ldi);
    auto valid = [&basel](QuadEdge* e) { return right_of(e->dest(), basel); };
    if (ldi->origin == ldo->origin)         ldo = basel->rev();
    if (rdi->origin == rdo->origin)         rdo = basel;
    while (true) {
        QuadEdge* lcand = basel->rev()->onext;
        if (valid(lcand)) {
            while (in_circle(basel->dest(), basel->origin, lcand->dest(),
                             lcand->onext->dest())) {
                QuadEdge* t = lcand->onext;
                delete_edge(lcand);         lcand = t;
            }
        }
        QuadEdge* rcand = basel->oprev();
        if (valid(rcand)) {
            while (in_circle(basel->dest(), basel->origin, rcand->dest(),
                             rcand->oprev()->dest())) {
                QuadEdge* t = rcand->oprev();
                delete_edge(rcand);         rcand = t;
            }
        }
        if (!valid(lcand) && !valid(rcand))       break;
        if (!valid(lcand) ||    (valid(rcand) && in_circle(lcand->dest(), lcand->origin,
                                    rcand->origin, rcand->dest())))
            basel = connect(rcand, basel->rev());
        else       basel = connect(basel->rev(), lcand->rev());
    }
    return make_pair(ldo, rdo);
}
vector<tuple<pt, pt, pt>> delaunay(vector<pt> p) {
    sort(p.begin(), p.end(), [](const pt& a, const pt& b) {
        return lt(a.x, b.x) || (eq(a.x, b.x) && lt(a.y, b.y));
    });
    auto res = build_tr(0, (int)p.size() - 1, p);
    QuadEdge* e = res.first;
    vector<QuadEdge*> edges = {e};
```

```
    while (lt(e->onext->dest().cross(e->dest(), e->origin), 0))      e = e->onext;
    auto add = [&p, &e, &edges]() {
        QuadEdge* curr = e;
        do {
            curr->used = true;
            p.push_back(curr->origin);            edges.push_back(curr->rev());
            curr = curr->lnext();
        } while (curr != e);
    };
    add();         p.clear();
    int kek = 0;
    while (kek < (int)edges.size())
        if (!(e = edges[kek++])->used)      add();
    vector<tuple<pt, pt, pt>> ans;
    for (int i = 0; i < (int)p.size(); i += 3)
      ans.push_back(make_tuple(p[i], p[i + 1], p[i + 2]));
    return ans;
}
```

Div

```
PII add_div(PII x, PII y) {
    PII temp;
    INT GCD = __gcd(x.second, y.second);    INT LCM = (x.second / GCD) * y.second;
    temp.second = LCM;
    temp.first = (LCM / x.second) * x.first + (LCM / y.second) * y.first;
    GCD = __gcd(temp.first, temp.second);
    temp.first /= GCD;   temp.second /= GCD;
    return temp;
}
PII subs_div(PII x, PII y){
    PII temp;
    INT GCD = __gcd(x.second, y.second);    INT LCM = (x.second / GCD) * y.second;
    temp.second = LCM;
    temp.first = (LCM / x.second) * x.first - (LCM / y.second) * y.first;
    GCD = __gcd(temp.first, temp.second);
    temp.first /= GCD;    temp.second /= GCD;=
    return temp;
}
PII mult_div(PII x, PII y) {
    PII temp;
    temp.first = x.first * y.first;   temp.second = x.second * y.second;
    INT GCD = __gcd(temp.first, temp.second);
    temp.first /= GCD;   temp.second /= GCD;
}
PII div_div(PII x, PII y) {
    PII temp;
    temp.first = x.first * y.second;  temp.second = x.second * y.first;
    INT GCD = __gcd(temp.first, temp.second);
    temp.first /= GCD;   temp.second /= GCD;
}
```

Find Nearest Pair of Point

```
// time complexity : O(nlogn)
struct Point {     double x, y;       };
bool cmp(Point a, Point b) {    return a.x < b.x;  };
```

```
double dis(Point a, Point b) { return (a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y); }
int main() {
    int n;  scanf("%d", &n);
    Point p[10000];
    for (int i = 0; i < n; i++)      scanf("%lf%lf", &p[i].x, &p[i].y);
    sort(p, p + n, cmp);
    double mini = dis(p[0], p[1]);
    for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n && (p[j].x - p[i].x)*(p[j].x - p[i].x) < mini; j++)
        if (dis(p[j], p[i]) < mini)  mini = dis(p[j], p[i]);
    mini = sqrt(mini);
    printf("%.4lf\n", mini);
}
```

Functions

```
struct POINT{      INT xx, yy;  };
struct EQUATION{   INT a, b, c;       }; //
bool onSegment(POINT pp, POINT qq, POINT rr) {
    if (qq.xx <= max(pp.xx, rr.xx) && qq.xx >= min(pp.xx, rr.xx) && qq.yy <= max(pp.yy, rr.yy
) && qq.yy >= min(pp.yy, rr.yy))      return true;
    else                              return false;
}
INT orientation(POINT p1, POINT p2, POINT p3) {
    INT val = (p2.yy - p1.yy) * (p3.xx - p2.xx) - (p2.xx - p1.xx) * (p3.yy - p2.yy);
    if (val == 0)          return 0; // colinear
    return (val > 0) ? 1 : 2; // clock or counterclock wise
}
bool doIntersect(POINT p1, POINT q1, POINT p2, POINT q2) {
    INT o1 = orientation(p1, q1, p2), o2 = orientation(p1, q1, q2);
    INT o3 = orientation(p2, q2, p1), o4 = orientation(p2, q2, q1);
    if (o1 != o2 && o3 != o4)   return true;
    // p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1))
        return true;
    // p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1))
        return true;
    // p2, q2 and p1 are colinear and p1 lies on segment p2q2
    if (o3 == 0 && onSegment(p2, p1, q2))
        return true;
    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2))
        return true;
    return false;
}
EQUATION convert_to_equation(POINT p1, POINT p2){
    EQUATION temp;
    temp.aa = p2.yy - p1.yy;   temp.bb = p1.xx - p2.xx;
    temp.cc = p1.yy * (p2.xx - p1.xx) + p1.xx * (p1.yy - p2.yy);
    return temp;
}
POINT sol_2_equation(EQUATION eq1, EQUATION eq2) {
    POINT temp, pp;
    INT value = eq1.aa * eq2.bb - eq2.aa * eq1.bb;
    pp.xx = 0;      pp.yy = 0;=
```

```
    if (value == 0)       return pp;
    temp.xx = (eq1.bb * eq2.cc - eq2.bb * eq1.cc) / value;
    temp.yy = (eq1.cc * eq2.aa - eq2.cc * eq1.aa) / value;
    return temp;
}
POINT sol_2_eq_from_point(POINT p1, POINT p2, POINT p3, POINT p4) {
    EQUATION eq1, eq2;
    eq1 = convert_to_equation(p1, p2);       eq2 = convert_to_equation(p2, p3);
    POINT temp = sol_2_equation(eq1, eq2);
    return temp;
}
```

geo_2

```
double CirclishArea(PT a, PT b, PT cen, double r) {
    double ang = fabs(atan2((a - cen).y, (a - cen).x) - atan2((b - cen).y, (b - cen).x));
    if (ang > PI)        ang = 2 * PI - ang;
    return (ang * r * r) / 2.0;
}
//intersection of circle and triangle
double CicleTriangleIntersectionArea(PT a, PT b, PT c, double radius){
    vector<PT> g = CircleLineIntersection(a, b, c, radius);
    if (b < a)           swap(a, b);
    if (g.size() < 2)    return CirclishArea(a, b, c, radius);
    else{
        PT l = g[0], r = g[1];
        if (r < l)              swap(l, r);
        if (b < l || r < a)     return CirclishArea(a, b, c, radius);
        else if (a < l && b < r)
             return fabs(SignedArea(c, b, l)) + CirclishArea(a, l, c, radius);
        else if (r < b && l < a)
            return fabs(SignedArea(a, c, r)) + CirclishArea(r, b, c, radius);
        else if (a < l && r < b)
            return fabs(SignedArea(c, l, r)) + CirclishArea(a, l, c, radius) +
                   CirclishArea(b, r, c, radius);
        Else                    return fabs(SignedArea(a, b, c));
    }
    return 0;
}
//intersection of circle and simple polygon (vertexes in counterclockwise order)
double CirclePolygonIntersectionArea(vector<PT> &p, PT c, double r) {
    int i, j, k, n = p.size();       double sum = 0;
    for (i = 0; i < p.size(); i++){
        double temp = CicleTriangleIntersectionArea(p[i], p[(i + 1) % n], c, r);
        double sign = SignedArea(c, p[i], p[(i + 1) % n]);
        if (dcmp(sign) == 1)         sum += temp;
        else if (dcmp(sign) == -1)   sum -= temp;
    }
    return sum;
}
//returns the left portion of convex polygon u cut by line a---b
vector<PT> CutPolygon(vector<PT> &u, PT a, PT b){
    vector<PT> ret;
    int n = u.size();
    for (int i = 0; i < n; i++){
        PT c = u[i], d = u[(i + 1) % n];
```

```
        if (dcmp((b - a) * (c - a)) >= 0)   ret.push_back(c);
        if (ProjectPointLine(a, b, c) == c || ProjectPointLine(a, b, d) == d)    continue;
        if (dcmp((b - a) * (d - c)) != 0)   {
            PT t;
            getIntersection(a, b - a, c, d - c, t);
            if (PointOnSegment(c, d, t))    ret.push_back(t);
        }
    }
    return ret;
}
typedef pair<PT, PT> seg_t;
vector<PT> tanCP(PT c, double r, PT p){
    double x = dot(p - c, p - c), d = x - r * r;
    vector<PT> res;
    if (d < -EPS)  return res;
    if (d < 0)     d = 0;
    PT q1 = (p - c) * (r * r / x), q2 = RotateCCW90((p - c) * (-r * sqrt(d) / x));
    res.push_back(c + q1 - q2);       res.push_back(c + q1 + q2);
    return res;
}
//Always check if the circles are same
vector<seg_t> tanCC(PT c1, double r1, PT c2, double r2) {
    vector<seg_t> res;
    if (fabs(r1 - r2) < EPS) {
        PT dir = c2 - c1;
        dir = RotateCCW90(dir * (r1 / dir.Magnitude()));
        res.push_back(seg_t(c1 + dir, c2 + dir)); res.push_back(seg_t(c1 - dir, c2 - dir));
    }
    else{
        PT p = ((c1 * -r2) + (c2 * r1)) / (r1 - r2);
        vector<PT> ps = tanCP(c1, r1, p), qs = tanCP(c2, r2, p);
        for (int i = 0; i < ps.size() && i < qs.size(); ++i)
            res.push_back(seg_t(ps[i], qs[i]));
    }
    PT p = ((c1 * r2) + (c2 * r1)) / (r1 + r2);
    vector<PT> ps = tanCP(c1, r1, p), qs = tanCP(c2, r2, p);
    for (int i = 0; i < ps.size() && i < qs.size(); ++i)
        res.push_back(seg_t(ps[i], qs[i]));
    return res;
}
//move segment a---b perpendicularly by distance d
pair<PT, PT> MoveSegmentLeft(PT a, PT b, double d) {
    double l = dist(a, b);
    PT p = ((RotateCCW90(b - a) * d) / l) + a;
    return mp(p, p + b - a);
}
void GetLineABC(Point A, Point B, double &a, double &b, double &c) {
    a = A.y - B.y;       b = B.x - A.x;     c = A.x * B.y - A.y * B.x;
}
double Sector(double r, double alpha) {     return r * r * 0.5 * (alpha - sin(alpha)); }
double CircleCircleIntersectionArea(double r1, double r2, double d) {
    if (dcmp(d - r1 - r2) != -1)      return 0;
    if (dcmp(d + r1 - r2) != 1)       return PI * r1 * r1;
    if (dcmp(d + r2 - r1) != 1)       return PI * r2 * r2;
    // using law of cosines
    double ans = Sector(r1, 2 * acos((r1 * r1 + d * d - r2 * r2) / (2.0 * r1 * d)));
```

```
    ans += Sector(r2, 2 * acos((r2 * r2 + d * d - r1 * r1) / (2.0 * r2 * d)));
    return ans;
}
//length of common part of polygon p and line s-t, O(nlogn)
double PolygonStubbing(vector<PT> &p, PT s, PT t) {
    int n = p.size();
    double sm = 0;
    for (int i = 0; i < n; i++)       sm += p[i] * p[(i + 1) % n];
    if (dcmp(sm) == -1)        reverse(all(p));
    vector<pair<double, int>> event;
    for (int i = 0; i < n; i++){
        int lef = dcmp(cross(p[i] - s, t - s)), rig = dcmp(cross(p[NEX(i)] - s, t - s));
        if (lef == rig)        continue;
        double r = cross(p[NEX(i)] - s, p[NEX(i)] - p[i]) / cross(t - s, p[NEX(i)] - p[i]);
        if (lef > rig)   event.push_back(make_pair(r, (!lef || !rig ? -1 : -2)));
        else             event.push_back(make_pair(r, (!lef || !rig ? 1 : 2)));
    }
    sort(event.begin(), event.end());
    int cnt = 0;         double sum = 0, la = 0;
    for (int i = 0; i < (int)event.size(); i++){
        if (cnt > 0)     sum += event[i].first - la;
        la = event[i].first;    cnt += event[i].second;
    }
    return sum * (t - s).Magnitude();
}
// Minimum Enclosing Circle Randomized O(n)
// Removing Duplicates takes O(nlogn)
typedef pair<PT, double> circle;
bool IsInCircle(circle C, PT p) {     return dcmp(C.second - dist(C.first, p)) >= 0; }
circle MinimumEnclosingCircle2(vector<PT> &p, int m, int n) {
    circle D = mp((p[m] + p[n]) / 2.0, dist(p[m], p[n]) / 2.0);
    for (int i = 0; i < m; i++)
        if (!IsInCircle(D, p[i])) {
            D.first = ComputeCircleCenter(p[i], p[m], p[n]);
            D.second = dist(D.first, p[i]);
        }
    return D;
}
circle MinimumEnclosingCircle1(vector<PT> &p, int n) {
    circle D = mp((p[0] + p[n]) / 2.0, dist(p[0], p[n]) / 2.0);
    for (int i = 1; i < n; i++)
        if (!IsInCircle(D, p[i]))     D = MinimumEnclosingCircle2(p, i, n);
    return D;
}
//changes vector; sorts and removes duplicate points(complexity bottleneck, unneccessary)
circle MinimumEnclosingCircle(vector<PT> p) {
    srand(time(NULL));
    sort(all(p));                                    //comment if tle
    p.resize(distance(p.begin(), unique(all(p)))); //comment if tle
    random_shuffle(all(p));
    if (p.size() == 1)    return mp(p[0], 0);
    circle D = mp((p[0] + p[1]) / 2.0, dist(p[0], p[1]) / 2.0);
    for (int i = 2; i < p.size(); i++)
        if (!IsInCircle(D, p[i]))      D = MinimumEnclosingCircle1(p, i);
    return D;
}
```

```
#define EPS 1e-12

#define NEX(x) ((x + 1) % n)
#define PRV(x) ((x - 1 + n) % n)
#define RAD(x) ((x * M_PI) / 180)
#define DEG(x) ((x * 180) / M_PI)
const double PI = acos(-1.0);
inline int dcmp(double x) { return x < -EPS ? -1 : (x > EPS); }
class PT{
public:
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    double Magnitude() { return sqrt(x * x + y * y); }
    bool operator==(const PT &u) const { return dcmp(x - u.x) == 0 && dcmp(y - u.y) == 0; }
    bool operator!=(const PT &u) const { return !(*this == u); }
    bool operator<(const PT &u) const { return dcmp(x - u.x) < 0 || (dcmp(x - u.x) == 0 && dc
mp(y - u.y) < 0); }
    bool operator>(const PT &u) const { return u < *this; }
    bool operator<=(const PT &u) const { return *this < u || *this == u; }
    bool operator>=(const PT &u) const { return *this > u || *this == u; }
    PT operator+(const PT &u) const { return PT(x + u.x, y + u.y); }
    PT operator-(const PT &u) const { return PT(x - u.x, y - u.y); }
    PT operator*(const double u) const { return PT(x * u, y * u); }
    PT operator/(const double u) const { return PT(x / u, y / u); }
    double operator*(const PT &u) const { return x * u.y - y * u.x; }
};
double dot(PT p, PT q) { return p.x * q.x + p.y * q.y; }
double dist2(PT p, PT q) { return dot(p - q, p - q); }
double dist(PT p, PT q) { return sqrt(dist2(p, q)); }
double cross(PT p, PT q) { return p.x * q.y - p.y * q.x; }
double myAsin(double val) {
    if (val > 1)          return PI * 0.5;
    if (val < -1)         return -PI * 0.5;
    return asin(val);
}
double myAcos(double val) {
    if (val > 1)    return 0;
    if (val < -1)   eturn PI;
    return acos(val);
}
ostream &operator<<(ostream &os, const PT &p) {   os << "(" << p.x << "," << p.y << ")"; }
istream &operator>>(istream &is, PT &p) {   is >> p.x >> p.y;        return is;   }
// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y, p.x); }
PT RotateCW90(PT p) { return PT(p.y, -p.x); }
PT RotateCCW(PT p, double t) {
    return PT(p.x * cos(t) - p.y * sin(t), p.x * sin(t) + p.y * cos(t));
}
PT RotateAroundPointCCW(PT p, PT pivot, double t) {
    PT trans = p - pivot;      PT ret = RotateCCW(trans, t);
    ret = ret + pivot;
    return ret;
```

```
}
// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
    return a + (b - a) * dot(c - a, b - a) / dot(b - a, b - a);
}
double DistancePointLine(PT a, PT b, PT c) {
    return dist(c, ProjectPointLine(a, b, c));
}
// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
    double r = dot(b - a, b - a);
    if (fabs(r) < EPS)          return a;
    r = dot(c - a, b - a) / r;
    if (r < 0)     return a;
    if (r > 1)     return b;
    return a + (b - a) * r;
}
// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c){
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}
// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x,double y,double z,double a,double b,double c,double d){
    return fabs(a * x + b * y + c * z - d) / sqrt(a * a + b * b + c * c);
}
// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {      return dcmp(cross(b - a, c - d)) == 0; }
bool LinesCollinear(PT a, PT b, PT c, PT d) {
    return LinesParallel(a,b,c,d) && dcmp(cross(a-b,a-c)) == 0 && dcmp(cross(c-d,c-a)) == 0;}
//UNTESTED CODE SEGMENT
// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
    if (LinesCollinear(a, b, c, d)) {
        if (dcmp(dist2(a, c)) == 0 || dcmp(dist2(a, d)) == 0 ||
            dcmp(dist2(b, c)) == 0 || dcmp(dist2(b, d)) == 0)
            return true;
        if (dcmp(dot(c - a, c - b)) > 0 && dcmp(dot(d - a, d - b)) > 0 && dcmp(dot(c - b, d
             - b)) > 0)
            return false;
        return true;
    }
    if (dcmp(cross(d - a, b - a)) * dcmp(cross(c - a, b - a)) > 0)   return false;
    if (dcmp(cross(a - c, d - c)) * dcmp(cross(b - c, d - c)) > 0)   return false;
    return true;
}
// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d){
    b = b - a;     d = c - d;          c = c - a;
    assert(dot(b, b) > EPS && dot(d, d) > EPS);
    return a + b * cross(c, d) / cross(b, d);
}
```

```
// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
    b = (a + b) / 2;     c = (a + c) / 2;
    return ComputeLineIntersection(b, b + RotateCW90(a - b), c, c + RotateCW90(a - c));
}
bool PointOnSegment(PT s, PT e, PT p) {
    if (p == s || p == e)      return 1;
    return dcmp(cross(s - p, s - e)) == 0 && dcmp(dot(PT(s.x - p.x, s.y - p.y), PT(e.x - p.x,
 e.y - p.y))) < 0;
}
int PointInPolygon(vector<PT> p, PT q){
    int i, j, cnt = 0, n = p.size();
    for (i = 0, j = n - 1; i < n; j = i++){
        if (PointOnSegment(p[i], p[j], q))        return 1;
        if (p[i].y > q.y != p[j].y > q.y &&
            q.x < (double)(p[j].x - p[i].x) * (q.y - p[i].y) / (double)(p[j].y - p[i].y) + p[
i].x)
            cnt++;
    }
    return cnt & 1;
}
// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q){
    for (int i = 0; i < p.size(); i++)
        if (dist2(ProjectPointSegment(p[i], p[(i + 1) % p.size()], q), q) < EPS) return true;
    return false;
}
// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
//THIS DOESN'T WORK FOR a == b
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r){
    vector<PT> ret;
    b = b - a;     a = a - c;
    double A = dot(b, b), B = dot(a, b), C = dot(a, a) - r * r;
    double D = B * B - A * C;
    if (D < -EPS)        return ret;
    ret.push_back(c + a + b * (-B + sqrt(D + EPS)) / A);
    if (D > EPS)         ret.push_back(c + a + b * (-B - sqrt(D)) / A);
    return ret;
}
// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R){
    vector<PT> ret;
    double d = sqrt(dist2(a, b));
    if (d > r + R || d + min(r, R) < max(r, R))         return ret;
    double x = (d * d - R * R + r * r) / (2 * d);
    double y = sqrt(r * r - x * x);
    PT v = (b - a) / d;
    ret.push_back(a + v * x + RotateCCW90(v) * y);
    if (y > 0)     ret.push_back(a + v * x - RotateCCW90(v) * y);
    return ret;
}
// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  Note that the centroid is often known as
```

```
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p){
    double area = 0;
    for (int i = 0; i < p.size(); i++){
        int j = (i + 1) % p.size();
        area += p[i].x * p[j].y - p[j].x * p[i].y;
    }
    return area / 2.0;
}
double ComputeArea(const vector<PT> &p){    return fabs(ComputeSignedArea(p));}
double ShoeLace(vector<PT> &vec){
    int i, n = vec.size();      double ans = 0;
    for (i = 0; i < n; i++)     ans += vec[i].x * vec[NEX(i)].y - vec[i].y * vec[NEX(i)].x;
    return fabs(ans) * 0.5;
}
PT ComputeCentroid(const vector<PT> &p){
    PT c(0, 0);
    double scale = 6.0 * ComputeSignedArea(p);
    for (int i = 0; i < p.size(); i++){
        int j = (i + 1) % p.size();
        c = c + (p[i] + p[j]) * (p[i].x * p[j].y - p[j].x * p[i].y);
    }
    return c / scale;
}

double PAngle(PT a, PT b, PT c){ //Returns positive angle abc
    PT temp1(a.x - b.x, a.y - b.y), temp2(c.x - b.x, c.y - b.y);
    double ans = myAsin((temp1.x * temp2.y - temp1.y * temp2.x) / (temp1.Magnitude() * temp2.
Magnitude()));
    if ((ans < 0 && (temp1.x * temp2.x + temp1.y * temp2.y) < 0) || (ans >= 0 && (temp1.x * t
emp2.x + temp1.y * temp2.y) < 0))
        ans = PI - ans;
    ans = ans < 0 ? 2 * PI + ans : ans;
    return ans;
}
double SignedArea(PT a, PT b, PT c){ //The area is positive if abc is in counter-clockwise
      //    direction
    PT temp1(b.x - a.x, b.y - a.y), temp2(c.x - a.x, c.y - a.y);
    return 0.5 * (temp1.x * temp2.y - temp1.y * temp2.x);
}

bool XYasscending(PT a, PT b){
    if (abs(a.x - b.x) < EPS)        return a.y < b.y;
    return a.x < b.x;
}
//Makes convex hull in counter-clockwise direction without repeating first point
//undefined if all points in given[] are collinear
//to allow 180' angle replace <= with <
void MakeConvexHull(vector<PT> given, vector<PT> &ans){
    int i, n = given.size(), j = 0, k = 0;
    vector<PT> U, L;
    ans.clear();
    sort(given.begin(), given.end(), XYasscending);
    for (i = 0; i < n; i++){
        while (true) {
            if (j < 2)         break;
```

```
            if (SignedArea(L[j - 2], L[j - 1], given[i]) <= EPS)     j--;
            else                                                     break;
        }
        if (j == L.size())     {
            L.push_back(given[i]);          j++;
        }
        else{
            L[j] = given[i];          j++;
        }
    }
    for (i = n - 1; i >= 0; i--){
        while (1) {
            if (k < 2)           break;
            if (SignedArea(U[k - 2], U[k - 1], given[i]) <= EPS)     k--;
            else                                                     break;
        }
        if (k == U.size())     {
            U.push_back(given[i]);    k++;
        }
        else {
            U[k] = given[i];    k++;
        }
    }
    for (i = 0; i < j - 1; i++)       ans.push_back(L[i]);
    for (i = 0; i < k - 1; i++)       ans.push_back(U[i]);
}
typedef PT Vector;
typedef vector<PT> Polygon;
struct DirLine{
    PT p;          Vector v;    double ang;
    DirLine() {}   //    DirLine (PT p, Vector v): p(p), v(v) { ang = atan2(v.y, v.x); }
    //adds the left of line p-q
    DirLine(PT p, PT q) {
        this->p = p;
        this->v.x = q.x - p.x;        this->v.y = q.y - p.y;
        ang = atan2(v.y, v.x);
    }
    bool operator<(const DirLine &u) const { return ang < u.ang; }
};
bool getIntersection(PT p, Vector v, PT q, Vector w, PT &o) {
    if (dcmp(cross(v, w)) == 0)       return false;
    Vector u = p - q;
    double k = cross(w, u) / cross(v, w);
    o = p + v * k;
    return true;
}
bool onLeft(DirLine l, PT p) { return dcmp(l.v * (p - l.p)) >= 0; }
int halfPlaneIntersection(DirLine *li, int n, PT *poly) {
    sort(li, li + n);
    int first, last;
    PT *p = new PT[n];
    DirLine *q = new DirLine[n];
    q[first = last = 0] = li[0];
    for (int i = 1; i < n; i++){
        while (first < last && !onLeft(li[i], p[last - 1]))          last--;
        while (first < last && !onLeft(li[i], p[first]))             first++;
```

```
        q[++last] = li[i];
        if (dcmp(q[last].v * q[last - 1].v) == 0) {
            last--;
            if (onLeft(q[last], li[i].p))          q[last] = li[i];
        }
        if (first < last)
            getIntersection(q[last - 1].p, q[last - 1].v, q[last].p, q[last].v, p[last - 1]);
    }
    while (first < last && !onLeft(q[first], p[last - 1]))      last--;
    if (last - first <= 1){
        delete[] p;       delete[] q;  return 0;
    }
    getIntersection(q[last].p, q[last].v, q[first].p, q[first].v, p[last]);
    int m = 0;
    for (int i = first; i <= last; i++)     poly[m++] = p[i];
    delete[] p;    delete[] q;  return m;
}
```

intersect point of line

```
struct pt {  double x, y; };
struct line {      double a, b, c;    };
const double EPS = 1e-9;
double det(double a, double b, double c, double d) {    return a*d - b*c;  }
bool intersect(line m, line n, pt & res) {
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS)   return false;
    res.x = -det(m.c, m.b, n.c, n.b) / zn;        res.y = -det(m.a, m.c, n.a, n.c) / zn;
    return true;
}
bool parallel(line m, line n) {       return abs(det(m.a, m.b, n.a, n.b)) < EPS;  }
bool equivalent(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS     && abs(det(m.a, m.c, n.a, n.c)) < EPS
        && abs(det(m.b, m.c, n.b, n.c)) < EPS;
}
//  check intersection
struct pt {
    long long x, y;
    pt() {}
    pt(long long _x, long long _y) : x(_x), y(_y) {}
    pt operator-(const pt& p) const { return pt(x - p.x, y - p.y); }
    long long cross(const pt& p) const { return x * p.y - y * p.x; }
    long long cross(const pt& a, const pt& b) const { return (a - *this).cross(b - *this); }
};
int sgn(const long long& x) { return x >= 0 ? x ? 1 : 0 : -1; }
bool inter1(long long a, long long b, long long c, long long d) {
    if (a > b)     swap(a, b);
    if (c > d)     swap(c, d);
    return max(a, c) <= min(b, d);
}
bool check_inter(const pt& a, const pt& b, const pt& c, const pt& d) {
    if (c.cross(a, d) == 0 && c.cross(b, d) == 0)
        return inter1(a.x, b.x, c.x, d.x) && inter1(a.y, b.y, c.y, d.y);
    return sgn(a.cross(b,c))!=sgn(a.cross(b, d)) && sgn(c.cross(d, a))!=sgn(c.cross(d, b));
}
//  intersection of 2 segment
```

```
const double EPS = 1E-9;
struct pt {
    double x, y;
    bool operator<(const pt& p) const{
        return x < p.x - EPS || (abs(x - p.x) < EPS && y < p.y - EPS);
    }
};
struct line {
    double a, b, c;
    line() {}
    line(pt p, pt q) {
        a = p.y - q.y;    b = q.x - p.x;    c = -a * p.x - b * p.y;  norm();
    }
    void norm(){
        double z = sqrt(a * a + b * b);
        if (abs(z) > EPS)       a /= z, b /= z, c /= z;
    }
    double dist(pt p) const { return a * p.x + b * p.y + c; }
};
double det(double a, double b, double c, double d) {    return a * d - b * c; }
inline bool betw(double l, double r, double x) {
    return min(l, r) <= x + EPS && x <= max(l, r) + EPS;
}
inline bool intersect_1d(double a, double b, double c, double d){
    if (a > b)     swap(a, b);
    if (c > d)     swap(c, d);
    return max(a, c) <= min(b, d) + EPS;
}
bool intersect(pt a, pt b, pt c, pt d, pt& left, pt& right) {
    if (!intersect_1d(a.x, b.x, c.x, d.x) || !intersect_1d(a.y, b.y, c.y, d.y))return false;
    line m(a, b);        line n(c, d);
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS) {
        if (abs(m.dist(c)) > EPS || abs(n.dist(a)) > EPS)     return false;
        if (b < a)       swap(a, b);
        if (d < c)       swap(c, d);
        left = max(a, c);      right = min(b, d);
        return true;
    } else {
        left.x = right.x = -det(m.c, m.b, n.c, n.b) / zn;
        left.y = right.y = -det(m.a, m.c, n.a, n.c) / zn;
        return betw(a.x, b.x, left.x) && betw(a.y, b.y, left.y) &&
               betw(c.x, d.x, left.x) && betw(c.y, d.y, left.y);
    }
}
//  circle - line intersection
double r, a, b, c; // given as input
double x0 = -a*c/(a*a+b*b), y0 = -b*c/(a*a+b*b);
if (c*c > r*r*(a*a+b*b)+EPS)   puts ("no points");
else if (abs (c*c - r*r*(a*a+b*b)) < EPS) {
    puts ("1 point");          cout << x0 << ' ' << y0 << '\n';
}
else {
    double d = r*r - c*c/(a*a+b*b);
    double mult = sqrt (d / (a*a+b*b)), ax, ay, bx, by;
    ax = x0 + b * mult;        bx = x0 - b * mult;
```

```
    ay = y0 - a * mult;         by = y0 + a * mult;
    puts ("2 points");
    cout << ax << ' ' << ay << '\n' << bx << ' ' << by << '\n';
}
```

length of union of segment

```
int length_union(const vector<pair<int, int>> &a) {
    int n = a.size();
    vector<pair<int, bool>> x(n*2);
    for (int i = 0; i < n; i++) {
        x[i*2] = {a[i].first, false};       x[i*2+1] = {a[i].second, true};
    }
    sort(x.begin(), x.end());
    int result = 0, c = 0;
    for (int i = 0; i < n * 2; i++) {
        if (i > 0 && x[i].first > x[i-1].first && c > 0)result += x[i].first - x[i-1].first;
        if (x[i].second)        c--;
        else                    c++;
    }
    return result;
}
```

nearest pair of points

```
struct pt {  int x, y, id;      };
struct cmp_x {
    bool operator()(const pt & a, const pt & b) const {
      return a.x < b.x || (a.x == b.x && a.y < b.y);
    }
};
struct cmp_y {
    bool operator()(const pt & a, const pt & b) const {
        return a.y < b.y;
    }
};
int n;      vector<pt> a;      double mindist;
pair<int, int> best_pair;
void upd_ans(const pt & a, const pt & b) {
    double dist = sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
    if (dist < mindist) {
        mindist = dist;         best_pair = {a.id, b.id};
    }
}
vector<pt> t;
void rec(int l, int r) {
    if (r - l <= 3) {
        for (int i = l; i < r; ++i)
            for (int j = i + 1; j < r; ++j)       upd_ans(a[i], a[j]);
        sort(a.begin() + l, a.begin() + r, cmp_y());
        return;
    }

    int m = (l + r) >> 1;             int midx = a[m].x;
    rec(l, m);           rec(m, r);
    merge(a.begin() + l, a.begin() + m, a.begin() + m, a.begin() + r, t.begin(), cmp_y());
    copy(t.begin(), t.begin() + r - l, a.begin() + l);
```

```
    int tsz = 0;
    for (int i = l; i < r; ++i) {
        if (abs(a[i].x - midx) < mindist) {
            for (int j = tsz - 1; j >= 0 && a[i].y - t[j].y < mindist; --j)
                upd_ans(a[i], t[j]);
            t[tsz++] = a[i];
        }
    }
}
int main(){
    t.resize(n);    sort(a.begin(), a.end(), cmp_x());         mindist = 1E20;
    rec(0, n);
}
```

Point

```
struct point2d {
    ftype x, y;
    point2d() {}
    point2d(ftype x, ftype y): x(x), y(y) {}
    point2d& operator+=(const point2d &t) {
        x += t.x;        y += t.y;
        return *this;
    }
    point2d& operator-=(const point2d &t) {
        x -= t.x;        y -= t.y;
        return *this;
    }
    point2d& operator*=(ftype t) {
        x *= t;          y *= t;
        return *this;
    }
    point2d& operator/=(ftype t) {
        x /= t;          y /= t;
        return *this;
    }
    point2d operator+(const point2d &t) const {   return point2d(*this) += t; }
    point2d operator-(const point2d &t) const {   return point2d(*this) -= t;    }
    point2d operator*(ftype t) const {      return point2d(*this) *= t; }
    point2d operator/(ftype t) const {      return point2d(*this) /= t;    }
};
point2d operator*(ftype a, point2d b) {     return b * a;      }
struct point3d {
    ftype x, y, z;
    point3d() {}
    point3d(ftype x, ftype y, ftype z): x(x), y(y), z(z) {}
    point3d& operator+=(const point3d &t) {
        x += t.x;  y += t.y;    z += t.z;
        return *this;
    }
    point3d& operator-=(const point3d &t) {
        x -= t.x;  y -= t.y;    z -= t.z;
        return *this;
    }
    point3d& operator*=(ftype t) {
        x *= t;    y *= t;      z *= t;
```

```
        return *this;
    }
    point3d& operator/=(ftype t) {
        x /= t;    y /= t;      z /= t;
        return *this;
    }
    point3d operator+(const point3d &t) const {   return point3d(*this) += t;    }
    point3d operator-(const point3d &t) const {   return point3d(*this) -= t;    }
    point3d operator*(ftype t) const {      return point3d(*this) *= t;    }
    point3d operator/(ftype t) const {      return point3d(*this) /= t;    }
};
point3d operator*(ftype a, point3d b) {     return b * a;      }
ftype dot(point2d a, point2d b) {     return a.x * b.x + a.y * b.y; }
ftype dot(point3d a, point3d b) {     return a.x * b.x + a.y * b.y + a.z * b.z; }
ftype norm(point2d a) {  return dot(a, a); }
double abs(point2d a) {  return sqrt(norm(a)); }
double proj(point2d a, point2d b) {   return dot(a, b) / abs(b);     }
double angle(point2d a, point2d b) { return acos(dot(a, b) / abs(a) / abs(b));   }
point3d cross(point3d a, point3d b) {
    return point3d(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z, a.x * b.y - a.y * b.x);
}
ftype triple(point3d a, point3d b, point3d c) {   return dot(a, cross(b, c)); }
ftype cross(point2d a, point2d b) {   return a.x * b.y - a.y * b.x;  }
point2d intersect(point2d a1, point2d d1, point2d a2, point2d d2) {
    return a1 + cross(a2 - a1, d2) / cross(d1, d2) * d1;
}
point3d intersect(point3d a1, point3d n1, point3d a2, point3d n2, point3d a3, point3d n3) {
    point3d x(n1.x, n2.x, n3.x), y(n1.y, n2.y, n3.y);
    point3d z(n1.z, n2.z, n3.z), d(dot(a1, n1), dot(a2, n2), dot(a3, n3));
    return point3d(triple(d, y, z), triple(x, d, z), triple(x, y, d)) / triple(n1, n2, n3);
}
```

Polygon

```
//  area of triangle
int signed_area_parallelogram(point2d p1, point2d p2, point2d p3) {
    return cross(p2 - p1, p3 - p2);
}
double triangle_area(point2d p1, point2d p2, point2d p3) {
    return abs(signed_area_parallelogram(p1, p2, p3)) / 2.0;
}
bool clockwise(point2d p1, point2d p2, point2d p3) {
    return signed_area_parallelogram(p1, p2, p3) < 0;
}
bool counter_clockwise(point2d p1, point2d p2, point2d p3) {
    return signed_area_parallelogram(p1, p2, p3) > 0;
}
//  area of polygon
double area(const vector<point>& fig) {
    double res = 0;
    for (unsigned i = 0; i < fig.size(); i++) {
        point p = i ? fig[i - 1] : fig.back();
        point q = fig[i];
        res += (p.x - q.x) * (p.y + q.y);
    }
    return fabs(res) / 2;
```

```
}
//  check if point belongs to simple polygon
//  complexity : O(log n)
struct pt{
    long long x, y;
    pt(){}
    pt(long long _x, long long _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); }
    long long cross(const pt & p) const { return x * p.y - y * p.x; }
    long long dot(const pt & p) const { return x * p.x + y * p.y; }
    long long cross(const pt & a, const pt & b) const { return (a-*this).cross(b-*this); }
    long long dot(const pt & a, const pt & b) const { return (a - *this).dot(b - *this); }
    long long sqrLen() const { return this->dot(*this); }
};
bool lexComp(const pt & l, const pt & r){   return l.x < r.x || (l.x == r.x && l.y < r.y); }
int sgn(long long val){   return val > 0 ? 1 : (val == 0 ? 0 : -1);  }
vector<pt> seq;
int n;
bool pointInTriangle(pt a, pt b, pt c, pt point){
    long long s1 = abs(a.cross(b, c));
    long long s2 = abs(point.cross(a, b)) + abs(point.cross(b, c)) + abs(point.cross(c, a));
    return s1 == s2;
}
void prepare(vector<pt> & points){
    n = points.size();
    int pos = 0;
    for(int i = 1; i < n; i++)
        if(lexComp(points[i], points[pos]))        pos = i;
    rotate(points.begin(), points.begin() + pos, points.end());
    n--;           seq.resize(n);
    for(int i = 0; i < n; i++)       seq[i] = points[i + 1] - points[0];
}
bool pointInConvexPolygon(pt point){
    if(seq[0].cross(point) != 0 && sgn(seq[0].cross(point)) != sgn(seq[0].cross(seq[n - 1])))
        return false;
    if(seq[n - 1].cross(point) != 0 && sgn(seq[n - 1].cross(point)) != sgn(seq[n - 1].cross(s
eq[0])))
        return false;
    if(seq[0].cross(point) == 0)      return seq[0].sqrLen() >= point.sqrLen();
    int l = 0, r = n - 1;
    while(r - l > 1){
        int mid = (l + r)/2;   int pos = mid;
        if(seq[pos].cross(point) >= 0)      l = mid;
        else                                r = mid;
    }
    int pos = l;
    return pointInTriangle(seq[pos], seq[pos + 1], pt(0, 0), point);
}
//  lattice point inside non-lattice polygon
int count_lattices(Fraction k, Fraction b, long long n) {
    auto fk = k.floor(), fb = b.floor(), cnt = 0LL;
    if (k >= 1 || b >= 1) {
        cnt += (fk * (n - 1) + 2 * fb) * n / 2;
        k -= fk;   b -= fb;
    }
```

```
    auto t = k * n + b;         auto ft = t.floor();
    if (ft >= 1)   cnt += count_lattices(1 / k, (t - t.floor()) / k, t.floor());
    return cnt;
}
```

Race Track

```
#define RAD(x) ((x * PI) / 180)
const double PI = acos(-1.0), EPS = 1e-12;
class PT{
public:
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    double Magnitude() { return sqrt(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 x == p.x ? y <= p.y : x <= p.x; }
};
double dot(PT p, PT q) { return p.x * q.x + p.y * q.y; }
double dist2(PT p, PT q) { return dot(p - q, p - q); }
double dist(PT p, PT q) { return sqrt(dist2(p, q)); }
double cross(PT p, PT q) { return p.x * q.y - p.y * q.x; }
// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c){
    return a + (b - a) * dot(c - a, b - a) / dot(b - a, b - a);
}
double DistancePointSegment(PT a, PT b, PT c){
    if (b < a)     swap(a, b);
    PT on = ProjectPointLine(a, b, c);
    if (a < on && on < b)      return dist(c, on);
    else{
        double ht = dist(c, on), bs = min(dist(a, on), dist(b, on));
        return sqrt(ht * ht + bs * bs);
    }
}
double mindist(vector<PT> &a, vector<PT> &b){
    int i, j, k;         double minn = inf;
    for (i = 1; i < a.size(); i++){
        for (j = 1; j < b.size(); j++){
            double atob = min(DistancePointSegment(a[i], a[i - 1], b[j]), DistancePointSegmen
t(a[i], a[i - 1], b[j - 1]));
            double btoa = min(DistancePointSegment(b[j], b[j - 1], a[i]), DistancePointSegmen
t(b[j], b[j - 1], a[i - 1]));
            minn = min(minn, min(atob, btoa));
        }
    }
    return minn;
}
int main(){
    int t, cs = 0, n, m, i, j, k, x, y;
    in(t);
```

```
    while (t--){
        vector<PT> a, b;
        in(n);
        for (i = 0; i < n; i++)      in2(x, y), a.pb(PT(x, y));
        in(m);
        for (i = 0; i < m; i++)      in2(x, y), b.pb(PT(x, y));
        a.pb(a[0]);      b.pb(b[0]);
        printf("Case %d: %.9f\n", ++cs, mindist(a, b) / 2.0);
    }
}
```

Sweep Line

```
//  search for a pair of intersecting line
const double EPS = 1E-9;
struct pt {  double x, y; };
struct seg {
    pt p, q;       int id;
    double get_y(double x) const {
        if (abs(p.x - q.x) < EPS)          return p.y;
        return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
    }
};
bool intersect1d(double l1, double r1, double l2, double r2) {
    if (l1 > r1)         swap(l1, r1);
    if (l2 > r2)         swap(l2, r2);
    return max(l1, l2) <= min(r1, r2) + EPS;
}
int vec(const pt& a, const pt& b, const pt& c) {
    double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    return abs(s) < EPS ? 0 : s > 0 ? +1 : -1;
}
bool intersect(const seg& a, const seg& b){
    return intersect1d(a.p.x, a.q.x, b.p.x, b.q.x)&&intersect1d(a.p.y, a.q.y, b.p.y, b.q.y)
      &&vec(a.p, a.q, b.p)*vec(a.p, a.q, b.q)<=0&&vec(b.p, b.q, a.p)*vec(b.p, b.q, a.q)<=0;
}
bool operator<(const seg& a, const seg& b){
    double x = max(min(a.p.x, a.q.x), min(b.p.x, b.q.x));
    return a.get_y(x) < b.get_y(x) - EPS;
}
struct event {
    double x;
    int tp, id;
    event() {}
    event(double x, int tp, int id) : x(x), tp(tp), id(id) {}
    bool operator<(const event& e) const {
        if (abs(x - e.x) > EPS)      return x < e.x;
        return tp > e.tp;
    }
};
set<seg> s;
vector<set<seg>::iterator> where;
set<seg>::iterator prev(set<seg>::iterator it) {  return it == s.begin() ? s.end() : --it; }
set<seg>::iterator next(set<seg>::iterator it) {  return ++it; }
pair<int, int> solve(const vector<seg>& a) {
    int n = (int)a.size();
```

```
    vector<event> e;
    for (int i = 0; i < n; ++i) {
        e.push_back(event(min(a[i].p.x, a[i].q.x), +1, i));
        e.push_back(event(max(a[i].p.x, a[i].q.x), -1, i));
    }
    sort(e.begin(), e.end());  s.clear();  where.resize(a.size());
    for (size_t i = 0; i < e.size(); ++i) {
        int id = e[i].id;
        if (e[i].tp == +1) {
            set<seg>::iterator nxt = s.lower_bound(a[id]), prv = prev(nxt);
            if (nxt != s.end() && intersect(*nxt, a[id]))     return make_pair(nxt->id, id);
            if (prv != s.end() && intersect(*prv, a[id]))     return make_pair(prv->id, id);
            where[id] = s.insert(nxt, a[id]);
        } else {
            set<seg>::iterator nxt = next(where[id]), prv = prev(where[id]);
            if (nxt != s.end() && prv != s.end() && intersect(*nxt, *prv))
                return make_pair(prv->id, nxt->id);
            s.erase(where[id]);
        }
    }
    return make_pair(-1, -1);
}
//  point location
typedef long long ll;
bool ge(const ll& a, const ll& b) { return a >= b; }
bool le(const ll& a, const ll& b) { return a <= b; }
bool eq(const ll& a, const ll& b) { return a == b; }
bool gt(const ll& a, const ll& b) { return a > b; }
bool lt(const ll& a, const ll& b) { return a < b; }
int sgn(const ll& x) { return le(x, 0) ? eq(x, 0) ? 0 : -1 : 1; }
struct pt {
    ll x, y;
    pt() {}
    pt(ll _x, ll _y) : x(_x), y(_y) {}
    pt operator-(const pt& a) const { return pt(x - a.x, y - a.y); }
    ll dot(const pt& a) const { return x * a.x + y * a.y; }
    ll dot(const pt& a, const pt& b) const { return (a - *this).dot(b - *this); }
    ll cross(const pt& a) const { return x * a.y - y * a.x; }
    ll cross(const pt& a, const pt& b) const { return (a - *this).cross(b - *this); }
    bool operator==(const pt& a) const { return a.x == x && a.y == y; }
};
struct Edge {      pt l, r;     };
bool edge_cmp(Edge* edge1, Edge* edge2){
    const pt a = edge1->l, b = edge1->r;
    const pt c = edge2->l, d = edge2->r;
    int val = sgn(a.cross(b, c)) + sgn(a.cross(b, d));
    if (val != 0)   return val > 0;
    val = sgn(c.cross(d, a)) + sgn(c.cross(d, b));
    return val < 0;
}
enum EventType { DEL = 2, ADD = 3, GET = 1, VERT = 0 };
struct Event {
    EventType type;
    int pos;
    bool operator<(const Event& event) const { return type < event.type; }
};
```

```
vector<Edge*> sweepline(vector<Edge*> planar, vector<pt> queries){
    using pt_type = decltype(pt::x);
    // collect all x-coordinates
    auto s = set<pt_type, std::function<bool(const pt_type&, const pt_type&)>>(lt);
    for (pt p : queries)        s.insert(p.x);
    for (Edge* e : planar) {
        s.insert(e->l.x);       s.insert(e->r.x);
    }
    // map all x-coordinates to ids
    int cid = 0;
    auto id = map<pt_type, int, std::function<bool(const pt_type&, const pt_type&)>>(lt);
    for (auto x : s)       id[x] = cid++;
    // create events
    auto t = set<Edge*, decltype(*edge_cmp)>(edge_cmp);
    auto vert_cmp = [](const pair<pt_type, int>& l, const pair<pt_type, int>& r) {
        if (!eq(l.first, r.first))    return lt(l.first, r.first);
        return l.second < r.second;
    };
    auto vert = set<pair<pt_type, int>, decltype(vert_cmp)>(vert_cmp);
    vector<vector<Event>> events(cid);
    for (int i = 0; i < (int)queries.size(); i++) {
        int x = id[queries[i].x];     events[x].push_back(Event{GET, i});
    }
    for (int i = 0; i < (int)planar.size(); i++) {
        int lx = id[planar[i]->l.x], rx = id[planar[i]->r.x];
        if (lx > rx) {
            swap(lx, rx);       swap(planar[i]->l, planar[i]->r);
        }
        if (lx == rx)      events[lx].push_back(Event{VERT, i});
        else{
            events[lx].push_back(Event{ADD, i});
            events[rx].push_back(Event{DEL, i});
        }
    }
    // perform sweep line algorithm
    vector<Edge*> ans(queries.size(), nullptr);
    for (int x = 0; x < cid; x++) {
        sort(events[x].begin(), events[x].end());       vert.clear();
        for (Event event : events[x]) {
            if (event.type == DEL)    t.erase(planar[event.pos]);
            if (event.type == VERT)
                vert.insert(make_pair(
                    min(planar[event.pos]->l.y, planar[event.pos]->r.y), event.pos));
            if (event.type == ADD)    t.insert(planar[event.pos]);
            if (event.type == GET) {
                auto jt = vert.upper_bound(make_pair(queries[event.pos].y, planar.size()));
                if (jt != vert.begin()) {
                    --jt;
                    int i = jt->second;
                    if (ge(max(planar[i]->l.y, planar[i]->r.y), queries[event.pos].y)) {
                        ans[event.pos] = planar[i];     continue;
                    }
                }
                Edge* e = new Edge;         e->l = e->r = queries[event.pos];
                auto it = t.upper_bound(e);
                if (it != t.begin())        ans[event.pos] = *(--it);
```

```
                delete e;
            }
        }
        for (Event event : events[x]) {
            if (event.type != GET)    continue;
            if (ans[event.pos] != nullptr && eq(ans[event.pos]->l.x, ans[event.pos]->r.x))
                continue;
            Edge* e = new Edge;       e->l = e->r = queries[event.pos];
            auto it = t.upper_bound(e);
            delete e;
            if (it == t.begin())      e = nullptr;
            else                      e = *(--it);
            if (ans[event.pos] == nullptr) {
                ans[event.pos] = e;         continue;
            }
            if (e == nullptr)         continue;
            if (e == ans[event.pos])  continue;
            if (id[ans[event.pos]->r.x] == x)
                if (id[e->l.x] == x)
                    if (gt(e->l.y, ans[event.pos]->r.y))      ans[event.pos] = e;
            else
                ans[event.pos] = e;
        }
    }
    return ans;
}
struct DCEL {
    struct Edge {
        pt origin;
        Edge* nxt = nullptr, *twin = nullptr;
        int face;
    };
    vector<Edge*> body;
};
vector<pair<int, int>> point_location(DCEL planar, vector<pt> queries){
    vector<pair<int, int>> ans(queries.size());
    vector<Edge*> planar2;
    map<intptr_t, int> pos, added_on;
    int n = planar.body.size();
    for (int i = 0; i < n; i++) {
        if (planar.body[i]->face > planar.body[i]->twin->face)      continue;
        Edge* e = new Edge;
        e->l = planar.body[i]->origin;      e->r = planar.body[i]->twin->origin;
        added_on[(intptr_t)e] = i;
        pos[(intptr_t)e] =
            lt(planar.body[i]->origin.x, planar.body[i]->twin->origin.x)
                ? planar.body[i]->face : planar.body[i]->twin->face;
        planar2.push_back(e);
    }
    auto res = sweepline(planar2, queries);
    for (int i = 0; i < (int)queries.size(); i++) {
        if (res[i] == nullptr) {
            ans[i] = make_pair(1, -1);      continue;
        }
        pt p = queries[i], l = res[i]->l, r = res[i]->r;
        if (eq(p.cross(l, r), 0) && le(p.dot(l, r), 0)) {
```

```
            ans[i] = make_pair(0, added_on[(intptr_t)res[i]]);       continue;
        }
        ans[i] = make_pair(1, pos[(intptr_t)res[i]]);
    }
    for (auto e : planar2)     delete e;
    return ans;
}
```

Tangent

```
//  tangent to 2 circle
truct pt {
    double x, y;
    pt operator- (pt p) {
        pt res = { x-p.x, y-p.y };
        return res;
    }
};
struct circle : pt {     double r;    };
struct line {      double a, b, c;};
const double EPS = 1E-9;
double sqr (double a) {  return a * a;}
void tangents (pt c, double r1, double r2, vector<line> & ans) {
    double r = r2 - r1, z = sqr(c.x) + sqr(c.y);
    double d = z - sqr(r);
    if (d < -EPS)        return;
    d = sqrt (abs (d));
    line l;
    l.a = (c.x * r + c.y * d) / z;    l.b = (c.y * r - c.x * d) / z;        l.c = r1;
    ans.push_back (l);
}
vector<line> tangents (circle a, circle b) {
    vector<line> ans;
    for (int i=-1; i<=1; i+=2)
        for (int j=-1; j<=1; j+=2)    tangents (b-a, a.r*i, b.r*j, ans);
    for (size_t i=0; i<ans.size(); ++i)     ans[i].c -= ans[i].a * a.x + ans[i].b * a.y;
    return ans;
}
```

Graph Theory

Asssn

```
#include <bits/stdc++.h>
using namespace std;
#define read freopen("C:\\Users\\Dell\\Desktop\\in.txt", "r", stdin)
#define write freopen("C:\\Users\\Dell\\Desktop\\out.txt", "w", stdout)
#define pii pair<int, int>
#define pll pair<LL, LL>
#define inf 1111111111
#define in(a) scanf("%d", &a)
#define ins(a) scanf("%s", a)
#define in2(a, b) scanf("%d%d", &a, &b)
#define in3(a, b, c) scanf("%d%d%d", &a, &b, &c)
#define pn printf("\n");
```

```
#define pr(a) printf("%d\n", a)
#define prs(a) printf("%d ", a)
#define pr2(a, b) printf("%d %d\n", a, b)
#define pr3(a, b, c) printf("%d %d %d\n", a, b, c)
#define MP make_pair
#define vi vector<int>
#define vll vector<LL>
#define _ceil(n, a) ((n) % (a) == 0 ? ((n) / (a)) : ((n) / (a) + 1))
#define cl clear()
#define sz size()
#define pb push_back
#define MEM(a, b) memset((a), (b), sizeof(a))
#define CASE printf("Case %d: ", ++cs)
#define all(X) (X).begin(), (X).end()
#define iter(it, X) for (__typeof((X).begin()) it = (X).begin(); it != (X).end(); it++)
#define oka(x, y) ((x) >= 0 && (x) < row && (y) >= 0 && (y) < col)
typedef long long LL;
int getnum(){
    char c = getchar();
    int num, sign = 1;
    for (; c < '0' || c > '9'; c = getchar())
        if (c == '-')    sign = -1;
    for (num = 0; c >= '0' && c <= '9';){
        c -= '0';  num = num * 10 + c;      c = getchar();
    }
    return num * sign;
}
const int M = 2002;
vi A[M], G[M], graph[M], trans[M];
int visited[M], matched_with[M], P[M], n;
bitset<M> reachable[M];
stack<int> S;
void pre_dfs(int u){
    visited[u] = true;
    for (int i = 0; i < graph[u].sz; i++)
        if (!visited[graph[u][i]])          pre_dfs(graph[u][i]);
    S.push(u);
}
void pre_dfs2(int u, int p){
    visited[u] = true;          P[u] = p;
    for (int i = 0; i < trans[u].sz; i++)
        if (!visited[trans[u][i]])          pre_dfs2(trans[u][i], p);
}
int convert_to_DAG(){
    int i, j, cnt = 0;
    MEM(visited, 0);
    for (i = 1; i <= n; i++)
        if (!visited[i])        pre_dfs(i);
    MEM(visited, 0);
    while (!S.empty()){
        if (!visited[S.top()])        pre_dfs2(S.top(), S.top());
        S.pop();
    }
    for (i = 1; i <= n; i++){
        if (P[i] == i)          cnt++;
        for (j = 0; j < graph[i].sz; j++){
```

```
            int u = P[i], v = P[graph[i][j]];
            if (u != v)         G[u].pb(v);
        }
    }
    return cnt;
}
void dfs(int u){
    int i, v;           visited[u] = true;
    for (i = 0; i < G[u].sz; i++){
        v = G[u][i];
        if (!visited[v])        dfs(v);
        reachable[u][v] = true;     reachable[u] |= reachable[v];
    }
}
void make_new_graph(){
    int i, j;
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            if (reachable[i][j])    A[i].pb(j + n);
}
bool find_new_match(int u){
    int i, v;
    for (i = 0; i < A[u].sz; i++){
        v = A[u][i];
        if (visited[v] == true)     continue;
        if (matched_with[v] == -1)  {
            matched_with[v] = u;    matched_with[u] = v;
            return true;
        }
        else if (matched_with[v] != u) {
            visited[v] = true;
            if (find_new_match(matched_with[v])) {
                matched_with[v] = u;        matched_with[u] = v;
                return true;
            }
        }
    }
    return false;
}
int _max_match(){
    int i, _match = 0, N = 2 * n;
    for (i = 0; i <= N; i++)   matched_with[i] = -1;
    _match = 0;
    for (i = 1; i <= n; i++){
        if (P[i] != i)   continue;
        if (matched_with[i] == -1) {
            MEM(visited, false);
            if (find_new_match(i))   _match++;
        }
    }
    return _match;
}
int main(){
#ifndef ONLINE_JUDGE
    read;
#endif
```

```
    int i, j, k, t, cs = 0, m, xnodes;
    t = getnum();
    while (t--){
        n = getnum();    m = getnum();
        while (m--){
            i = getnum();      j = getnum();
            graph[i].pb(j);    trans[j].pb(i);
        }
        xnodes = convert_to_DAG();
        MEM(visited, 0);       MEM(reachable, 0);
        for (i = 1; i <= n; i++)
            if (!visited[i])   dfs(i);
        make_new_graph();
        k = _max_match();
        CASE;
        pr(xnodes - k);
        for (i = 0; i <= 2 * n; i++){
            A[i].cl;     G[i].cl;     graph[i].cl;       trans[i].cl;
        }
    }}
```

Dinic

```
const int MAX = 10100;
int que[MAX];
template <class T>
struct Edge{
    int to, next;        T cap, flow;
    Edge(int to, int next, T cap) {
        this->to = to;         this->next = next;
        this->cap = cap;       this->flow = 0;
    }
};
template <class T>
struct Dinic{
    T INF;
    const int nodes;
    int source, sink, lvl[MAX], nodeEnd[MAX], last[MAX];
    vector<Edge<T>> edgeList;
    Dinic(int n):nodes(n),INF(numeric_limits<T>::max()/4){fill(nodeEnd, nodeEnd + n, -1);}
    void addEdge(int u, int v, T cap = 1){
        edgeList.push_back(Edge<T>(v, nodeEnd[u], cap));
        nodeEnd[u] = (int)edgeList.size() - 1;
        edgeList.push_back(Edge<T>(u, nodeEnd[v], 0));
        nodeEnd[v] = (int)edgeList.size() - 1;
    }
    bool createLevel(){
        memset(lvl, -1, nodes * sizeof(int));
        int qs = 0, qt = 0;
        que[qs] = source, lvl[source] = 0;
        while (qs <= qt) {
            int nd = que[qs++], ch;
            for (int i = nodeEnd[nd]; i != -1; i = edgeList[i].next)
                if (lvl[ch = edgeList[i].to] == -1 && edgeList[i].cap > edgeList[i].flow)
                    lvl[ch] = lvl[nd] + 1, que[++qt] = ch;
        }
```

```
        return lvl[sink] != -1;
    }
    T blockingFlow(int nd, T flow) {
        if (nd == sink)  return flow;
        int ch;
        T pflow = flow;
        for (int &i = last[nd]; i != -1; i = edgeList[i].next)
            if (lvl[ch = edgeList[i].to] == lvl[nd] + 1) {
                T pushed = blockingFlow(ch, min(pflow, edgeList[i].cap - edgeList[i].flow));
                pflow -= pushed;
                edgeList[i].flow += pushed;
                edgeList[i ^ 1].flow -= pushed;
                if (!pflow)    break;
            }
        return flow - pflow;
    }
    T maxFlow(int src, int snk){
        source = src, sink = snk;
        T tot = 0;
        while (createLevel()){
            memcpy(last, nodeEnd, nodes * sizeof(int));
            tot += blockingFlow(source, INF);
        }
        return tot;
    }
};
```

Hopcroft karp

```
const int MAXN1 = 50000, MAXN2 = 50000, MAXM = 150000;
int n1,n2,edges, last[MAXN1], prv[MAXM], head[MAXM], matching[MAXN2], dist[MAXN1], Q[MAXN1];
bool used[MAXN1], vis[MAXN1];
void init(int _n1, int _n2){
    n1 = _n1;      n2 = _n2;    edges = 0;
    fill(last, last + n1, -1);
}
void addEdge(int u, int v){
    head[edges] = v;     prv[edges] = last[u];    last[u] = edges++;
}
void bfs(){
    fill(dist, dist + n1, -1);
    int sizeQ = 0;
    for (int u = 0; u < n1; ++u)
        if (!used[u]) {
            Q[sizeQ++] = u;    dist[u] = 0;
        }
    for (int i = 0; i < sizeQ; i++){
        int u1 = Q[i];
        for (int e = last[u1]; e >= 0; e = prv[e]) {
            int u2 = matching[head[e]];
            if (u2 >= 0 && dist[u2] < 0) {
                dist[u2] = dist[u1] + 1;    Q[sizeQ++] = u2;
            }
        }
    }
}
```

```
bool dfs(int u1){
    vis[u1] = true;
    for (int e = last[u1]; e >= 0; e = prv[e]) {
        int v = head[e];        int u2 = matching[v];
        if (u2 < 0 || !vis[u2] && dist[u2] == dist[u1] + 1 && dfs(u2)) {
            matching[v] = u1;   used[u1] = true;
            return true;
        }
    }
    return false;
}
int maxMatching(){
    fill(used, used + n1, false);     fill(matching, matching + n2, -1);
    for (int res = 0;;) {
        bfs();
        fill(vis, vis + n1, false);
        int f = 0;
        for (int u = 0; u < n1; ++u)
            if (!used[u] && dfs(u))   ++f;
        if (!f)    return res;
        res += f;
    }
}
```

Hungarian Max Weight Matching

```
/*
 * Algorithm : Hungarian algorithm Max Weighted Bi-partite Matching
 * Complexity : O( N^3 )
 * Note : 0 base indexing
 */
long cost[MAX][MAX];   // cost matrix
long N, max_match;   // N workers and N jobs
long lx[MAX], ly[MAX]; // Labels of X and Y parts
long xy[MAX];          // xy[x] - vertex that is matched with x,
long yx[MAX];          // yx[y] - vertex that is matched with y
bool S[MAX], T[MAX];   // Sets S and T in algorithm
long slack[MAX];
long slackx[MAX]; // slackx[y] such a vertex, that
                  // l(slackx[y]) + l(y) - w(slackx[y],y) = slack[y]
long Prev[MAX];   // Array for memorizing alternating paths
void Init_Labels(){
    memset(lx, 0, sizeof(lx)); memset(ly, 0, sizeof(ly));
    long x, y;
    for (x = 0; x < N; x++)
        for (y = 0; y < N; y++)      lx[x] = max(lx[x], cost[x][y]);
}
void Update_Labels(){
    long x, y, delta = INF;
    for (y = 0; y < N; y++)
        if (!T[y])       delta = min(delta, slack[y]);
    for (x = 0; x < N; x++)
        if (S[x])        lx[x] -= delta;
    for (y = 0; y < N; y++)
        if (T[y])        ly[y] += delta;
    for (y = 0; y < N; y++)
```

```
        if (!T[y])       slack[y] -= delta;
}
void Add_To_Tree(long x, long prevx){
    S[x] = true;         Prev[x] = prevx;         long y;
    for (y = 0; y < N; y++)
        if (lx[x] + ly[y] - cost[x][y] < slack[y])      {
            slack[y] = lx[x] + ly[y] - cost[x][y];      slackx[y] = x;
        }
}
void Augment(){
    if (max_match == N)        return;
    long x, y, root, q[MAX], wr = 0, rd = 0;
    memset(S, false, sizeof(S));      memset(T, false, sizeof(T));
    memset(Prev, -1, sizeof(Prev));
    for (x = 0; x < N; x++)
        if (xy[x] == -1) {
            q[wr++] = root = x;       Prev[x] = -2;      S[x] = true;       break;
        }
    for (y = 0; y < N; y++){
        slack[y] = lx[root] + ly[y] - cost[root][y];
        slackx[y] = root;
    }
    while (true) {
        while (rd < wr) {
            x = q[rd++];
            for (y = 0; y < N; y++){
                if (cost[x][y] == lx[x] + ly[y] && !T[y]) {
                    if (yx[y] == -1)        break;
                    T[y] = true;      q[wr++] = yx[y];
                    Add_To_Tree(yx[y], x);
                }
            }
            if (y < N)         break;
        }
        if (y < N)       break;
        Update_Labels();
        wr = rd = 0;
        for (y = 0; y < N; y++){
            if (!T[y] && slack[y] == 0) {
                if (yx[y] == -1) {
                    x = slackx[y];          break;
                }
                else{
                    T[y] = true;
                    if (!S[yx[y]]) {
                        q[wr++] = yx[y];    Add_To_Tree(yx[y], slackx[y]);
                    }
                }
            }
        }
        if (y < N)       break;
    }
    if (y < N) {
        max_match++;
        for (long cx = x, cy = y, ty; cx != -2; cx = Prev[cx], cy = ty) {
            ty = xy[cx];       yx[cy] = cx;       xy[cx] = cy;
```

```
        }
        Augment();
    }
}
long Hungarian(){
    long x, ret = 0;
    max_match = 0;
    memset(xy, -1, sizeof(xy));       memset(yx, -1, sizeof(yx));
    Init_Labels();
    Augment();
    for (x = 0; x < N; x++)    ret += cost[x][xy[x]];
    return ret;
}
```

MinCost MaxFlow

```
/*
 •  (Riaz vai) Algoritm : Min Cost Max Flow using Bellmen Ford
 * Note : Vertex are 0 indexing Based
 */
#define MAX_V 3777
#define INF 777777777
struct NODE{
    long v, Cap, Cost, RevInd; // This ind is necesery for multigraph to knw which edge is us
ed to take flow
};
vector<NODE> Edge[MAX_V + 7];
long nV, nE, P, SRC, TNK;
// This PInd is neceserry for multigraph to knw which edge ind of parent is used to take flow
long Par[MAX_V + 7], PInd[MAX_V + 7], SD[MAX_V + 7]; // Shortest path
void SetEdge(long u, long v, long Cap, long Cost){
    NODE U = {v, Cap, Cost, Edge[v].size()};
    NODE V = {u, 0, -Cost, Edge[u].size()};
    Edge[u].push_back(U);       Edge[v].push_back(V);
}
bool BFord(void){
    long i, u, k;
    for (i = 0; i < nV; i++){
        Par[i] = -1;     SD[i] = INF;
    }
    bool IsChange = true;
    SD[SRC] = 0;
    while (IsChange) {
        IsChange = false;
        for (u = SRC; u <= TNK; u++){
            for (i = 0; i < Edge[u].size(); i++){
                if (!Edge[u][i].Cap)        continue;
                long v = Edge[u][i].v;
                TD = SD[u] + Edge[u][i].Cost;
                if (SD[v] > TD){ // relaxation
                    SD[v] = TD;       Par[v] = u;        PInd[v] = i;
                    IsChange = true;
                }
            }
        }
    }
```

```
    return Par[TNK] != -1;
}
long FindVol(long s, long t){
    long Cap = Edge[Par[t]][PInd[t]].Cap;
    if (s == Par[t])     return Cap;
    else                 return min(Cap, FindVol(s, Par[t]));
}
long AugmentPath(long s, long t, long V){
    if (s == t)    return 0;
    long Cost = Edge[Par[t]][PInd[t]].Cost * V;
    Edge[Par[t]][PInd[t]].Cap -= V;
    Edge[t][Edge[Par[t]][PInd[t]].RevInd].Cap += V;
    return Cost + AugmentPath(s, Par[t], V);
}
void MinCost(long &Flow, long &Cost){
    Flow = Cost = 0;
    while (BFord()){
        long V = FindVol(SRC, TNK);
        Flow += V;
        Cost += AugmentPath(SRC, TNK, V);
    }
}
```

2 Sat

```
int n;
vector<vector<int>> g, gt;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;
void dfs1(int v) {
    used[v] = true;
    for (int u : g[v])
        if (!used[u])    dfs1(u);
    order.push_back(v);
}
void dfs2(int v, int cl) {
    comp[v] = cl;
    for (int u : gt[v])
        if (comp[u] == -1)     dfs2(u, cl);
}
bool solve_2SAT() {
    used.assign(n, false);
    for (int i = 0; i < n; ++i)
        if (!used[i])    dfs1(i);
    comp.assign(n, -1);
    for (int i = 0, j = 0; i < n; ++i) {
        int v = order[n - i - 1];
        if (comp[v] == -1)     dfs2(v, j++);
    }
    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
        if (comp[i] == comp[i + 1])   return false;
        assignment[i / 2] = comp[i] > comp[i + 1];
    }
    return true;
```

}

2nd best min span tree

```
struct edge {
    int s, e, w, id;
    bool operator<(const struct edge& other) { return w < other.w; }
};
typedef struct edge Edge;
const int N = 2e5 + 5;
long long res = 0, ans = 1e18;
int n, m, a, b, w, id, l = 21;
vector<Edge> edges;
vector<int> h(N, 0), parent(N, -1), size(N, 0), present(N, 0);
vector<vector<pair<int, int>>> adj(N), dp(N, vector<pair<int, int>>(l));
vector<vector<int>> up(N, vector<int>(l, -1));
pair<int, int> combine(pair<int, int> a, pair<int, int> b) {
    vector<int> v = {a.first, a.second, b.first, b.second};
    int topTwo = -3, topOne = -2;
    for (int c : v) {
        if (c > topOne) {
            topTwo = topOne;    topOne = c;
        else if (c > topTwo && c < topOne)        topTwo = c;
    }
    return {topOne, topTwo};
}
void dfs(int u, int par, int d) {
    h[u] = 1 + h[par];
    up[u][0] = par;      dp[u][0] = {d, -1};
    for (auto v : adj[u])
        if (v.first != par)    dfs(v.first, u, v.second);
}
pair<int, int> lca(int u, int v) {
    pair<int, int> ans = {-2, -3};
    if (h[u] < h[v])     swap(u, v);
    for (int i = l - 1; i >= 0; i--)
        if (h[u] - h[v] >= (1 << i)) {
            ans = combine(ans, dp[u][i]);   u = up[u][i];
        }
    if (u == v)    return ans;
    for (int i = l - 1; i >= 0; i--)
        if (up[u][i] != -1 && up[v][i] != -1 && up[u][i] != up[v][i]) {
            ans = combine(ans, combine(dp[u][i], dp[v][i]));
            u = up[u][i];      v = up[v][i];
        }
    ans = combine(ans, combine(dp[u][0], dp[v][0]));
    return ans;
}
int main(void) {
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        parent[i] = i;   size[i] = 1;
    }
    for (int i = 1; i <= m; i++) {
        cin >> a >> b >> w; // 1-indexed    edges.push_back({a, b, w, i - 1});
```

```
    }
    sort(edges.begin(), edges.end());
    for (int i = 0; i <= m - 1; i++) {
        a = edges[i].s;  b = edges[i].e;    w = edges[i].w;    id = edges[i].id;
        if (unite_set(a, b)) {
            adj[a].emplace_back(b, w);      adj[b].emplace_back(a, w);
            present[id] = 1;
            res += w;
        }
    }
    dfs(1, 0, 0);
    for (int i = 1; i <= l - 1; i++)
        for (int j = 1; j <= n; ++j)
            if (up[j][i - 1] != -1) {
                int v = up[j][i - 1];
                up[j][i] = up[v][i - 1];
                dp[j][i] = combine(dp[j][i - 1], dp[v][i - 1]);
            }
    for (int i = 0; i <= m - 1; i++) {
        id = edges[i].id;       w = edges[i].w;
        if (!present[id]) {
            auto rem = lca(edges[i].s, edges[i].e);
            if (rem.first != w)
                if (ans > res + w - rem.first)    ans = res + w - rem.first;
            else if (rem.second != -1)
                if (ans > res + w - rem.second)   ans = res + w - rem.second;
        }
    }
    cout << ans << "\n";
}
```

articulation point

```
#define MAX_V 107
vector<long> Edge[MAX_V + 7];
long p[MAX_V + 7], Low[MAX_V + 7], Ind[MAX_V + 7], I , nChild[MAX_V + 7], nVertex;
bool IsArt[MAX_V + 7], Visit[MAX_V + 7];
void Dfs(long u){
    Visit[u] = true;     Ind[u] = ++I;      nChild[u] = 0;
    IsArt[u] = false;    Low[u] = I;        long i;
    for (i = 0; i < Edge[u].size(); i++){
        long v = Edge[u][i];
        if (!Visit[v]) {
            p[v] = u;
            nChild[u]++;
            Dfs(v); // for findin bridge Low[v] > Ind[u]
            if (Low[v] >= Ind[u] && p[u] != -1)         IsArt[u] = true;
            Low[u] = min(Low[u], Low[v]);
        }
        else if (p[u] != v)          Low[u] = min(Low[u], Ind[v]);
    }
}
long Calc(void){
    long i;
    memset(&Visit[1], 0, sizeof(bool) * nVertex);
    for (i = 1; i <= nVertex; i++){
```

```
        if (Visit[i])    continue;
        p[i] = -1;       I = 0;
        Dfs(i);
        if (nChild[i] > 1)     IsArt[i] = true; // special check for root
    };
    long Ans = 0;
    for (i = 1; i <= nVertex; i++)
        if (IsArt[i])    Ans++;
    return Ans;
}
```

Articulation Point & Bridge

```
vector<int>g[MX];
int Tm=0, d[MX], low[MX], NoOfChildren=0, parent[MX];
bool ArticulationPoint[MX], visited[MX];
void FindArticulationPoint(int u, int root){
    if(u==root)    memset(parent, -1, sizeof parent);
    low[u]=d[u]=Tm++;    visited[u]=1;
    for(int i=0 ; i<g[u].size() ; i++){
        int v=g[u][i];
        if(u==root && !visited[v])  NoOfChildren++;
        if(v==parent[u])       continue;
        else if(visited[v])    low[u]=min(low[u], d[v]);
        else{
            parent[v]=u;       FindArticulationPoint(v, root);
        }
        low[u]=min(low[u], low[v]);
        if(d[u]<=low[v])       ArticulationPoint[u]=1;
    }
    if(NoOfChildren>1 && u==root)    ArticulationPoint[root]=1;
    else                             ArticulationPoint[root]=0;
}
set<PII>edge;
void FindArticulationBridge(int u, int root){
    if(u==root)    memset(parent, -1, sizeof parent);
    low[u]=d[u]=Tm++;    visited[u]=1;
    for(auto v : g[u]) {
        if(u==root && !visited[v])     NoOfChildren++;
        if(v==parent[u])       continue;
        else if(visited[v])    low[u]=min(low[u], d[v]);
        else{
            parent[v]=u;       FindArticulationBridge(v, root);
        }
        low[u]=min(low[u], low[v]);
        if(d[u]<low[v]) {
            edge.insert( PII(u, v) );       edge.insert( PII(v, u) );
        }
    }
    if(NoOfChildren>1 && u==root)
    for(auto v : g[u]){
        edge.insert( PII(u, v) );    edge.insert( PII(v, u) );
    }
}
```

BFS, DFS, Dijkstra

```
vector<int> g[MX];
int dist[MX], path[MX], visited[MX]; //use disjoint set find() to print path
void BFS(int source, int destination = -1){
    queue<int> Q;
    for (int i = 0; i < MX; i++)      visited[i] = 0;
    Q.push(source);
    dist[source] = 0;    visited[source] = 1;
    while (!Q.empty()){
        int u = Q.front();      Q.pop();
        for (int i = 0; i < g[u].size(); i++){
            int v = g[u][i];
            if (!visited[v]) {
                dist[v] = dist[u] + 1;      visited[v] = 1;    path[v] = u;
                Q.push(v);
            }
        }
    }
}
int Tm = 0, FinishingTime[MX], ArrivalTime[MX];
void DFS(int node = 0, int d = 0){
    ArrivalTime[node] = Tm++;  visited[node] = 1;       dist[node] = d;
    for (int i = 0; i < g[node].size(); i++)
        if (!visited[g[node][i]]) {
            path[g[node][i]] = node;
            DFS(g[node][i], d + 1);
        }
    FinishingTime[node] = Tm++;
}
vector<int> cost[MX];
int cost1[MX][MX];
void Dijkstra(int source){
    map<int, int> m;
    for (int i = 1; i < MX; i++){
        dist[i] = INT_MAX;      path[i] = -1;
    }
    m[0] = source;        dist[source] = 0;
    while (!m.empty()){
        map<int, int>::iterator it = m.begin();
        int u = it->second;          m.erase(it);
        for (int i = 0; i < g[u].size(); i++){
            int v = g[u][i], NewCost = dist[u] + cost[u][i];
            if (NewCost < dist[v]) {
                path[v] = u;   dist[v] = NewCost;       m[NewCost] = v;
            }
        }
    }
}
void PrintPath(int v){
    if (v == -1)         return;
    PrintPath(path[v]);
    cout << v << " ";
}
```

euler path

```
int main() {
```

```
    int n;
    vector<vector<int>> g(n, vector<int>(n));
    vector<int> deg(n);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)  deg[i] += g[i][j];
    int first = 0;
    while (!deg[first])        ++first;
    int v1 = -1, v2 = -1;
    bool bad = false;
    for (int i = 0; i < n; ++i)
        if (deg[i] & 1)
            if (v1 == -1)            v1 = i;
            else if (v2 == -1)       v2 = i;
            else                     bad = true;
    if (v1 != -1)              ++g[v1][v2], ++g[v2][v1];
    stack<int> st;       st.push(first);
    vector<int> res;
    while (!st.empty()) {
        int v = st.top(), i;
        for (i = 0; i < n; ++i)
            if (g[v][i])       break;
        if (i == n) {
            res.push_back(v);        st.pop();
        }else {
            --g[v][i];         --g[i][v];         st.push(i);
        }
    }
    if (v1 != -1) {
        for (size_t i = 0; i + 1 < res.size(); ++i) {
            if ((res[i] == v1 && res[i + 1] == v2) || (res[i] == v2 && res[i + 1] == v1)) {
                vector<int> res2;
                for (size_t j = i + 1; j < res.size(); ++j)    res2.push_back(res[j]);
                for (size_t j = 1; j <= i; ++j)                res2.push_back(res[j]);
                res = res2;
                break;
            }
        }
    }
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (g[i][j])       bad = true;
    if (bad)       cout << -1;
    else           for (int x : res)        cout << x << " ";
}
```

Euler Circuit & Path

```
vector<int> g[30];
int Tm = 0, d[MX], low[MX], NoOfChildren = 0, parent[MX];
bool ArticulationPoint[MX], visited[MX];
set<PII> edge;
int indegree[30], outdegree[30];
void FindArticulationBridge(int u = 0, int root = 0){
    if (u == root)        memset(parent, -1, sizeof parent);
    low[u] = d[u] = Tm++;      visited[u] = 1;
    for (int i = 0; i < g[u].size(); i++){
```

```
        int v = g[u][i];
        if (u == root && !visited[v])       NoOfChildren++;
        if (v == parent[u])                 continue;
        else if (visited[v])                low[u] = min(low[u], d[v]);
        else{
            parent[v] = u;
            FindArticulationBridge(v, root);
        }
        low[u] = min(low[u], low[v]);
        if (d[u] < low[v])      edge.insert(PII(u, v));
    }
    if (NoOfChildren > 1 && u == root)
        for (int i = 0; i < g[u].size(); i++)     edge.insert(PII(u, g[u][i]));
}
PII HasDirectedEulerPath(){
    vector<int> v;
    PII p(-1, -1);
    for (int i = 0; i < 30; i++)
        if (outdegree[i] != indegree[i])    v.push_back(i);
    if (v.size() == 2)
        if (outdegree[v[0]] - indegree[v[0]] == 1 && indegree[v[1]] - outdegree[v[1]] == 1)
            p = PII(v[0], v[1]);
        else if(outdegree[v[1]]-indegree[v[1]] == 1 && indegree[v[0]]-outdegree[v[0]] == 1)
            p = PII(v[1], v[0]);
    v.clear();
    return p;
}
deque<int> path;
void EulerCircuit(int n){
    visited[n] = 1;
    for (auto i : g[n])
        if (!visited[i] && edge.find(PII(n, i)) == edge.end())       EulerCircuit(i);
    for (auto i : g[n])
        if (!visited[i])        EulerCircuit(i);
    path.push_front(n);
}
int main(){
    FindArticulationBridge(1, 1);
    memset(visited, 0, sizeof visited);
    EulerCircuit(1);
}
```

find bridge online

```
vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
int bridges, 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, 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;
            last_visit[a] = lca_iteration;
            a = par[a];
        }
        if (b != -1) {
            path_b.push_back(b);
            b = find_2ecc(b);
            if (last_visit[b] == lca_iteration)         lca = b;
            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), 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);
}
```

Fleury's Euler Circuit & Path

```
vector<int> g[30];
int Tm = 0, d[MX], low[MX], NoOfChildren = 0, parent[MX], indegree[30], outdegree[30];
bool ArticulationPoint[MX], visited[MX];
set<PII> edge;
void FindArticulationBridge(int u = 0, int root = 0){
    if (u == root)        memset(parent, -1, sizeof parent);
    low[u] = d[u] = Tm++;             visited[u] = 1;
    for (int i = 0; i < g[u].size(); i++){
        int v = g[u][i];
        if (u == root && !visited[v])       NoOfChildren++;
        if (v == parent[u])           continue;
        else if (visited[v])          low[u] = min(low[u], d[v]);
        else{
            parent[v] = u;
            FindArticulationBridge(v, root);
        }
        low[u] = min(low[u], low[v]);
        if (d[u] < low[v])      edge.insert(PII(u, v));
    }
    if (NoOfChildren > 1 && u == root)
        for (int i = 0; i < g[u].size(); i++)     edge.insert(PII(u, g[u][i]));
}
PII HasDirectedEulerPath(){ //first is starting node & 2nd is ending node
    vector<int> v;
    PII p(-1, -1);
    for (int i = 0; i < 30; i++)
        if (outdegree[i] != indegree[i])    v.push_back(i);
    if (v.size() == 2)
        if (outdegree[v[0]] - indegree[v[0]] == 1 && indegree[v[1]] - outdegree[v[1]] == 1)
            p = PII(v[0], v[1]);
        else if (outdegree[v[1]]-indegree[v[1]] == 1 && indegree[v[0]]-outdegree[v[0]] == 1)
            p = PII(v[1], v[0]);
    v.clear();
    return p;
}
deque<int> path;
void EulerCircuit(int n){
    visited[n] = 1;
    for (auto i : g[n])
        if (!visited[i] && edge.find(PII(n, i)) == edge.end())       EulerCircuit(i);
    for (auto i : g[n])
        if (!visited[i])        EulerCircuit(i);
    path.push_front(n);
}
int main(){
    FindArticulationBridge(1, 1);
```

```
    EulerCircuit(1);
}
```

hierholzers Euler path

```
vector<int> g[MX];
deque<int> circuit, CurPath;
void HierHolzar(int start){
    circuit.clear();     CurPath.clear();
    int EdgeCount[MX];
    for (int i = 0; i < MX; i++)      EdgeCount[i] = g[i].size();
    CurPath.push_back(start);
    int CurV = start;
    while (CurPath.size())
        if (EdgeCount[CurV]) {
            CurPath.push_back(CurV);
            int NextV = g[CurV].back();
            EdgeCount[CurV]--;
            g[CurV].pop_back();
            CurV = NextV;
        }
        else{
            circuit.push_front(CurV);
            CurV = CurPath.back();
            CurPath.pop_back();
        }
}
```

HLD

```
#define MAX 30007
int N; // number of node in tree
vector<int> Edge[MAX + 7];
int SubT[MAX + 7];  // subtree size
int Par[MAX + 7];   // parent of a node
int Level[MAX + 7]; // level of a node
int nC;                 // number of chain
int ChainLdr[MAX + 7]; // chainleadr of a node
                        // for light edge chainldr is that node
int Chain[MAX + 7]; // node v in is which chain
int nP;                 // number of position , obviously == N
int Pos[MAX + 7];     // Pos of a node in chain/dfs order
int Explore(int u, int p, int l){ // find subtree size and level
    SubT[u] = 1;   Par[u] = p;        Level[u] = l;      int i;
    for (i = 0; i < Edge[u].size(); i++){
        int v = Edge[u][i];
        if (p == v)      continue;
        SubT[u] += Explore(v, u, l + 1);
    }
    return SubT[u];
}
void HeavyLight(int u, int k, bool IsL){ //if IsL make this node a chainledr of new chain
    if (IsL) {
        k = ++nC;         ChainLdr[k] = u;
    }
    Chain[u] = k;  Pos[u] = ++nP;//Update( nP,W[u] ); if query is need can b updated here
    int i, mx = -1; // max subtree size child is mx
```

```
    for (i = 0; i < Edge[u].size(); i++){
        int v = Edge[u][i];
        if (Par[u] == v)       continue;
        if (mx == -1)                       mx = v;
        else if (SubT[v] > SubT[mx])        mx = v;
    }
    if (mx == -1)        return;
    HeavyLight(mx, k, false);
    for (i = 0; i < Edge[u].size(); i++){
        int v = Edge[u][i];
        if (Par[u] == v || mx == v)         continue;
        HeavyLight(v, 0, true);
    }
}
int LCA(int u, int v){
    while (Chain[u] != Chain[v])
        if (Level[ChainLdr[Chain[u]]] < Level[ChainLdr[Chain[v]]])
            v = Par[ChainLdr[Chain[v]]];
        else       u = Par[ChainLdr[Chain[u]]];
    if (Level[u] < Level[v])    return u;
    else                        return v;
}
int main(void){
    Explore(0, 0, 0);
    HeavyLight(0, 0, true);
}
//  cp-algorithm
vector<int> parent, depth, heavy, head, pos;
int cur_pos;
int dfs(int v, vector<vector<int>> const& adj) {
    int size = 1;
    int max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c, adj);
            size += c_size;
            if (c_size > max_c_size)        max_c_size = c_size, heavy[v] = c;
        }
    }
    return size;
}
int decompose(int v, int h, vector<vector<int>> const& adj) {
    head[v] = h, pos[v] = cur_pos++;
    if (heavy[v] != -1)         decompose(heavy[v], h, adj);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy[v])      decompose(c, c, adj);
    }
}
void init(vector<vector<int>> const& adj) {
    int n = adj.size();
    parent = vector<int>(n);    depth = vector<int>(n);  heavy = vector<int>(n, -1);
    head = vector<int>(n);      pos = vector<int>(n);
    cur_pos = 0;
    dfs(0, adj);
    decompose(0, 0, adj);
```

```
}
int query(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = parent[head[b]]) {
        if (depth[head[a]] > depth[head[b]])      swap(a, b);
        int cur_heavy_path_max = segment_tree_query(pos[head[b]], pos[b]);
        res = max(res, cur_heavy_path_max);
    }
    if (depth[a] > depth[b])    swap(a, b);
    int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
    res = max(res, last_heavy_path_max);
    return res;
}
```

min cost flow

```
struct Edge{       int from, to, capacity, cost; };
vector<vector<int>> adj, cost, capacity;
const int INF = 1e9;
void shortest_paths(int n, int v0, vector<int>& d, vector<int>& p) {
    d.assign(n, INF);
    d[v0] = 0;
    vector<bool> inq(n, false);
    queue<int> q;
    q.push(v0);
    p.assign(n, -1);
    while (!q.empty()) {
        int u = q.front();      q.pop();
        inq[u] = false;
        for (int v : adj[u])
            if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v]) {
                d[v] = d[u] + cost[u][v];   p[v] = u;
                if (!inq[v]) {
                    inq[v] = true;    q.push(v);
                }
            }
    }
}
int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t) {
    adj.assign(N, vector<int>());
    cost.assign(N, vector<int>(N, 0));
    capacity.assign(N, vector<int>(N, 0));
    for (Edge e : edges) {
        adj[e.from].push_back(e.to);        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity;
    }
    int flow = 0, cost = 0;
    vector<int> d, p;
    while (flow < K) {
        shortest_paths(N, s, d, p);
        if (d[t] == INF)        break;
        int f = K - flow, cur = t;
        while (cur != s) {
            f = min(f, capacity[p[cur]][cur]);          cur = p[cur];
        }
```

```
        flow += f;       cost += f * d[t];       cur = t;
        while (cur != s) {
            capacity[p[cur]][cur] -= f;     capacity[cur][p[cur]] += f;
            cur = p[cur];
        }
    }
    if (flow < K)        return -1;
    else                 return cost;
}
//  assignment
const int INF = 1000 * 1000 * 1000;
vector<int> assignment(vector<vector<int>> a) {
    int n = a.size(), m = n * 2 + 2;
    vector<vector<int>> f(m, vector<int>(m));
    int s = m - 2, t = m - 1, cost = 0;
    while (true) {
        vector<int> dist(m, INF), p(m);
        vector<bool> inq(m, false);
        queue<int> q;
        dist[s] = 0;     p[s] = -1;
        q.push_back(s);
        while (!q.empty()) {
            int v = q.front();       q.pop();
            inq[v] = false;
            if (v == s) {
                for (int i = 0; i < n; ++i) {
                    if (f[s][i] == 0) {
                        dist[i] = 0;       p[i] = s;    inq[i] = true;
                        q.push(i);
                    }
                }
            } else {
                if (v < n) {
                    for (int j = n; j < n + n; ++j) {
                        if (f[v][j] < 1 && dist[j] > dist[v] + a[v][j - n]) {
                            dist[j] = dist[v] + a[v][j - n];
                            p[j] = v;
                            if (!inq[j]) {
                                q.push(j);  inq[j] = true;
                            }
                        }
                    }
                } else {
                    for (int j = 0; j < n; ++j) {
                        if (f[v][j] < 0 && dist[j] > dist[v] - a[j][v - n]) {
                            dist[j] = dist[v] - a[j][v - n];
                            p[j] = v;
                            if (!inq[j]) {
                                q.push(j);       inq[j] = true;
                            }
                        }
                    }
                }
            }
        }
        int curcost = INF;
```

```
        for (int i = n; i < n + n; ++i)
            if (f[i][t] == 0 && dist[i] < curcost) {
                curcost = dist[i];    p[t] = i;
            }
        if (curcost == INF)    break;
        cost += curcost;
        for (int cur = t; cur != -1; cur = p[cur]) {
            int prev = p[cur];
            if (prev != -1)    f[cur][prev] = -(f[prev][cur] = 1);
        }
    }
    vector<int> answer(n);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (f[i][j + n] == 1)    answer[i] = j;
    return answer;
}
//  isbipartite
void isBipartitie() {
    int n;
    vector<vector<int>> adj;
    vector<int> side(n, -1);
    bool is_bipartite = true;
    queue<int> q;
    for (int st = 0; st < n; ++st)
        if (side[st] == -1) {
            q.push(st);
            side[st] = 0;
            while (!q.empty()) {
                int v = q.front();    q.pop();
                for (int u : adj[v])
                    if (side[u] == -1) {
                        side[u] = side[v] ^ 1           q.push(u);
                    } else           is_bipartite &= side[u] != side[v];
            }
        }
    cout << (is_bipartite ? "YES" : "NO") << endl;
}
```

MST

```
#define MX 10000
struct edge{
    int u, v, w;
    edge(int _u, int _v, int _w) {
        u = _u;    v = _v;      w = _w;
    }
} vector<edge> e;
int pr[MX];
bool comp(edge a, edge b) { return a.w <= b.w; }
int Find(int r){
    int m = n;
    while (pr[m] != m)   m = pr[m];
    while (pr[n] != n)   {
        int k = pr[n];   pr[n] = m;   n = k;
    }
```

```
    return m;
}
int mst(int node){
    sort(e.begin(), e.end(), comp);
    for (int i = 0; i <= node; i++)  pr[i] = i;
    int cnt = 0, sum = 0;
    for (int i = 0; i < e.size(); i++){
        int u = Find(e[i].u), v = Find(e[i].v);
        if (u != v) {
            pr[u] = v;   cnt++;       sum += e[i].w;
            if (cnt == n - 1)  break;
        }
    }
    return sum;
}
```

Paint Edge of Tree

```
typedef vector<vector<int>> graph;
vector<int> dfs_list, edges_list, h;
void dfs(int v, const graph& g, const graph& edge_ids, int cur_h = 1) {
    h[v] = cur_h;
    dfs_list.push_back(v);
    for (size_t i = 0; i < g[v].size(); ++i)
        if (h[g[v][i]] == -1) {
            edges_list.push_back(edge_ids[v][i]);
            dfs(g[v][i], g, edge_ids, cur_h + 1);
            edges_list.push_back(edge_ids[v][i]);
            dfs_list.push_back(v);
        }
}
vector<int> lca_tree, first;
void lca_tree_build(int i, int l, int r) {
    if (l == r)    lca_tree[i] = dfs_list[l];
    else {
        int m = (l + r) >> 1;
        lca_tree_build(i + i, l, m);
        lca_tree_build(i + i + 1, m + 1, r);
        int lt = lca_tree[i + i], rt = lca_tree[i + i + 1];
        lca_tree[i] = h[lt] < h[rt] ? lt : rt;
    }
}
void lca_prepare(int n) {
    lca_tree.assign(dfs_list.size() * 8, -1);
    lca_tree_build(1, 0, (int)dfs_list.size() - 1);
    first.assign(n, -1);
    for (int i = 0; i < (int)dfs_list.size(); ++i) {
        int v = dfs_list[i];
        if (first[v] == -1)    first[v] = i;
    }
}
int lca_tree_query(int i, int tl, int tr, int l, int r) {
    if (tl == l && tr == r)    return lca_tree[i];
    int m = (tl + tr) >> 1;
    if (r <= m)          return lca_tree_query(i + i, tl, m, l, r);
    if (l > m)           return lca_tree_query(i + i + 1, m + 1, tr, l, r);
```

```
    int lt = lca_tree_query(i + i, tl, m, l, m);
    int rt = lca_tree_query(i + i + 1, m + 1, tr, m + 1, r);
    return h[lt] < h[rt] ? lt : rt;
}
int lca(int a, int b) {
    if (first[a] > first[b])   swap(a, b);
    return lca_tree_query(1, 0, (int)dfs_list.size() - 1, first[a], first[b]);
}
vector<int> first1, first2, tree1, tree2;
vector<char> edge_used;
void query_prepare(int n) {
    first1.resize(n - 1, -1);        first2.resize(n - 1, -1);
    for (int i = 0; i < (int)edges_list.size(); ++i) {
        int j = edges_list[i];
        if (first1[j] == -1)   first1[j] = i;
        else                   first2[j] = i;
    }
    edge_used.resize(n - 1);
    tree1.resize(edges_list.size() * 8);    tree2.resize(edges_list.size() * 8);
}
void sum_tree_update(vector<int>& tree, int i, int l, int r, int j, int delta) {
    tree[i] += delta;
    if (l < r) {
        int m = (l + r) >> 1;
        if (j <= m)      sum_tree_update(tree, i + i, l, m, j, delta);
        else             sum_tree_update(tree, i + i + 1, m + 1, r, j, delta);
    }
}
int sum_tree_query(const vector<int>& tree, int i, int tl, int tr, int l, int r) {
    if (l > r || tl > tr)            return 0;
    if (tl == l && tr == r)          return tree[i];
    int m = (tl + tr) >> 1;
    if (r <= m)          return sum_tree_query(tree, i + i, tl, m, l, r);
    if (l > m)           return sum_tree_query(tree, i + i + 1, m + 1, tr, l, r);
    return sum_tree_query(tree, i + i, tl, m, l, m) +
           sum_tree_query(tree, i + i + 1, m + 1, tr, m + 1, r);
}
int query(int v1, int v2) {
    return sum_tree_query(tree1, 1, 0, (int)edges_list.size() - 1, first[v1], first[v2] - 1)-
           sum_tree_query(tree2, 1, 0, (int)edges_list.size() - 1, first[v1], first[v2] - 1);
}
int main() {
    h.assign(n, -1);
    dfs(0, g, edge_ids);
    lca_prepare(n);
    query_prepare(n);
    for (;;) {
        if () {
            // request for painting edge x;
            // if start = true, then the edge is painted, otherwise the painting
            // is removed
            edge_used[x] = start;
            sum_tree_update(tree1, 1, 0, (int)edges_list.size() - 1, first1[x],start?1:-1);
            sum_tree_update(tree2, 1, 0, (int)edges_list.size() - 1, first2[x],start?1:-1);
        } else {
            // query the number of colored edges on the path between v1 and v2
```

```
            int l = lca(v1, v2);
            int result = query(l, v1) + query(l, v2);
            // result - the answer to the request
        }
    }
}
```

strong orientation

```
//  A strong orientation of an undirected graph is an assignment of a direction to each edge
// that makes it a strongly connected graph. That is, after the orientation we should be able
// to isit any vertex from any vertex by following the directed edges.
vector<vector<pair<int, int>>> adj; // adjacency list - vertex and edge pairs
vector<pair<int, int>> edges;
vector<int> tin, low;
int bridge_cnt;
string orient;
vector<bool> edge_used;
void find_bridges(int v) {
    static int time = 0;
    low[v] = tin[v] = time++;
    for (auto p : adj[v]) {
        if (edge_used[p.second]) continue;
        edge_used[p.second] = true;
        orient[p.second] = v == edges[p.second].first ? '>' : '<';
        int nv = p.first;
        if (tin[nv] == -1) { // if nv is not visited yet
            find_bridges(nv);
            low[v] = min(low[v], low[nv]);
            if (low[nv] > tin[v])    bridge_cnt++;// a bridge between v and nv
        } else     low[v] = min(low[v], low[nv]);
    }
}
int main() {
    adj.resize(n);  tin.resize(n, -1);      low.resize(n, -1); orient.resize(m);
    edges.resize(m);     edge_used.resize(m);
    int comp_cnt = 0;
    for (int v = 0; v < n; v++) {
        if (tin[v] == -1) {
            comp_cnt++;        find_bridges(v);
        }
    }
    printf("%d\n%s\n", comp_cnt + bridge_cnt, orient.c_str());
}
```

strongly connected component

```
vector < vector<int> > g, gr;
vector<bool> used;
vector<int> order, component;
void dfs1 (int v) {
    used[v] = true;
    for (size_t i=0; i<g[v].size(); ++i)
        if (!used[ g[v][i] ])  dfs1 (g[v][i]);
    order.push_back (v);
}
void dfs2 (int v) {
```

```
    used[v] = true;       component.push_back (v);
    for (size_t i=0; i<gr[v].size(); ++i)
        if (!used[ gr[v][i] ])       dfs2 (gr[v][i]);
}
int main() {
    used.assign (n, false);
    for (int i=0; i<n; ++i)
        if (!used[i])    dfs1 (i);
    used.assign (n, false);
    for (int i=0; i<n; ++i) {
        int v = order[n-1-i];
        if (!used[v]) {
            dfs2 (v);
            component.clear();
        }
    }
}
```

topological sort

```
#define MX 100005
vector<int> g[MX];
bool visited[MX];
deque<int> d;
int indegree[MX];
void topological_sort(int n){
    if (visited[n])      return;
    visited[n] = 1;
    for (int i = 0; i < g[n].size(); i++)   topological_sort(g[n][i]);
    d.push_front(n);
}
void TopologicalSort(int sz){
    for (int i = 1; i <= sz; i++)
        if (indegree[i] == 0)  topological_sort(i);
}
int main(){
    for (int i = 0; i < n; i++){
        g[u].push_back(v);
        indegree[v]++;
    }
    TopologicalSort(n);
}
```

Linear Algebra

determinant of a matrix by Gauss

```
int main() {
    const double EPS = 1E-9;
    int n;
    vector < vector<double> > a (n, vector<double> (n));
    double det = 1;
    for (int i=0; i<n; ++i) {
        int k = i;
        for (int j=i+1; j<n; ++j)
```

```
            if (abs (a[j][i]) > abs (a[k][i]))    k = j;
        if (abs (a[k][i]) < EPS) {
            det = 0;     break;
        }
        swap (a[i], a[k]);
        if (i != k)      det = -det;
        det *= a[i][i];
        for (int j=i+1; j<n; ++j)     a[i][j] /= a[i][i];
        for (int j=0; j<n; ++j)
            if (j != i && abs (a[j][i]) > EPS)
                for (int k=i+1; k<n; ++k)   a[j][k] -= a[i][k] * a[j][i];
    }
    cout << det;
}
```

determinant using Kraut method

```
//  complexity : O(n^3)
static BigInteger det (BigDecimal a [][], int n) {
    try {
    for (int i=0; i<n; i++) {
        boolean nonzero = false;
        for (int j=0; j<n; j++)
            if (a[i][j].compareTo (new BigDecimal (BigInteger.ZERO)) > 0)  nonzero = true;
        if (!nonzero)    return BigInteger.ZERO;
    }
    BigDecimal scaling [] = new BigDecimal [n];
    for (int i=0; i<n; i++) {
        BigDecimal big = new BigDecimal (BigInteger.ZERO);
        for (int j=0; j<n; j++)
            if (a[i][j].abs().compareTo (big) > 0)      big = a[i][j].abs();
        scaling[i] = (new BigDecimal (BigInteger.ONE)) .divide
            (big, 100, BigDecimal.ROUND_HALF_EVEN);
    }
    int sign = 1;
    for (int j=0; j<n; j++) {
        for (int i=0; i<j; i++) {
            BigDecimal sum = a[i][j];
            for (int k=0; k<i; k++)   sum = sum.subtract (a[i][k].multiply (a[k][j]));
            a[i][j] = sum;
        }
        BigDecimal big = new BigDecimal (BigInteger.ZERO);
        int imax = -1;
        for (int i=j; i<n; i++) {
            BigDecimal sum = a[i][j];
            for (int k=0; k<j; k++)   sum = sum.subtract (a[i][k].multiply (a[k][j]));
            a[i][j] = sum;
            BigDecimal cur = sum.abs();
            cur = cur.multiply (scaling[i]);
            if (cur.compareTo (big) >= 0) {
                big = cur;     imax = i;
            }
        }
        if (j != imax) {
            for (int k=0; k<n; k++) {
                BigDecimal t = a[j][k];
```

```
                a[j][k] = a[imax][k];       a[imax][k] = t;
            }
            BigDecimal t = scaling[imax];
            scaling[imax] = scaling[j];     scaling[j] = t;
            sign = -sign;
        }
        if (j != n-1)
            for (int i=j+1; i<n; i++)
                a[i][j] = a[i][j].divide(a[j][j], 100, BigDecimal.ROUND_HALF_EVEN);
    }
    BigDecimal result = new BigDecimal (1);
    if (sign == -1)      result = result.negate();
    for (int i=0; i<n; i++)    result = result.multiply (a[i][i]);
    return result.divide(BigDecimal.valueOf(1),0,BigDecimal.ROUND_HALF_EVEN).toBigInteger();
    }
    catch (Exception e) {      return BigInteger.ZERO; }
}
```

Gauss & System of Linear Equations

```
//  complexity : O(n + m)
const double EPS = 1e-9;
const int INF = 2; // it doesn't actually have to be infinity or a big number
int gauss (vector < vector<double> > a, vector<double> & ans) {
    int n = (int) a.size(), m = (int) a[0].size() - 1;
    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs (a[sel][col]))    sel = i;
        if (abs (a[sel][col]) < EPS)        continue;
        for (int i=col; i<=m; ++i)    swap (a[sel][i], a[row][i]);
        where[col] = row;
        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] / a[row][col];
                for (int j=col; j<=m; ++j)        a[i][j] -= a[row][j] * c;
            }
        ++row;
    }
    ans.assign (m, 0);
    for (int i=0; i<m; ++i)
        if (where[i] != -1)    ans[i] = a[where[i]][m] / a[where[i]][i];
    for (int i=0; i<n; ++i) {
        double sum = 0;
        for (int j=0; j<m; ++j)       sum += ans[j] * a[i][j];
        if (abs (sum - a[i][m]) > EPS)      return 0;
    }
    for (int i=0; i<m; ++i)
        if (where[i] == -1)    return INF;
    return 1;
}
int gauss (vector < bitset<N> > a, int n, int m, bitset<N> & ans) {
    vector<int> where (m, -1);
    for (int col=0, row=0; col<m && row<n; ++col) {
        for (int i=row; i<n; ++i)
```

```
            if (a[i][col]) {
                swap (a[i], a[row]);        break;
            }
        if (! a[row][col])     continue;
        where[col] = row;
        for (int i=0; i<n; ++i)
            if (i != row && a[i][col])      a[i] ^= a[row];
        ++row;
    }// The rest of implementation is the same as above
}
```

rank of a matrix

```
const double EPS = 1E-9;
int compute_rank(vector<vector<double>> A) {
    int n = A.size(), m = A[0].size(), rank = 0;
    vector<bool> row_selected(n, false);
    for (int i = 0; i < m; ++i) {
        int j;
        for (j = 0; j < n; ++j) {
            if (!row_selected[j] && abs(A[j][i]) > EPS)       break;
        }
        if (j != n) {
            ++rank;      row_selected[j] = true;
            for (int p = i + 1; p < m; ++p)       A[j][p] /= A[j][i];
            for (int k = 0; k < n; ++k)
                if (k != j && abs(A[k][i]) > EPS)
                    for (int p = i + 1; p < m; ++p)     A[k][p] -= A[j][p] * A[k][i];
        }
    }
    return rank;
}
```

Number Theory

Big Number Calculation

```
const int base = 1000 * 1000 * 1000;
void output(VI &a) {
    printf ("%d", a.empty() ? 0 : a.back());
    for (int i=(int)a.size()-2; i>=0; --i)        printf ("%09d", a[i]);
}
void input(string s, VI &a) {
    for (int i=(int)s.length(); i>0; i-=9)
    if (i < 9)     a.push_back (atoi (s.substr (0, i).c_str()));
    else           a.push_back (atoi (s.substr (i-9, 9).c_str()));
    while (a.size() > 1 && a.back() == 0)         a.pop_back();
}
VI& add(VI &a, VI &b) {
    int carry = 0;
    for (size_t i=0; i<max(a.size(),b.size()) || carry; ++i) {
        if (i == a.size())     a.push_back (0);
        a[i] += carry + (i < b.size() ? b[i] : 0);
        carry = a[i] >= base;
        if (carry)       a[i] -= base;
```

```
    }
    return a;
}
VI& substract(VI &a, VI &b) {
    int carry = 0;
    for (size_t i=0; i<b.size() || carry; ++i) {
        a[i] -= carry + (i < b.size() ? b[i] : 0);
        carry = a[i] < 0;
        if (carry)          a[i] += base;
    }
    while (a.size() > 1 && a.back() == 0)    a.pop_back();
    return a;
}
VI& multiplicationLS(VI &a, VI &b) {
    int carry = 0;
    for (size_t i=0; i<a.size() || carry; ++i) {
        if (i == a.size())      a.push_back (0);
        long long cur = carry + a[i] * 1ll * b;
        a[i] = int (cur % base);
        carry = int (cur / base);
    }
    while (a.size() > 1 && a.back() == 0)         a.pop_back();
    return a;
}
VI multiplicationLL(VI &a, VI &b) {
    VI c (a.size()+b.size());
    for (size_t i=0; i<a.size(); ++i)
        for (int j=0, carry=0; j<(int)b.size() || carry; ++j) {
            long long cur = c[i+j] + a[i] * 1ll * (j < (int)b.size() ? b[j] : 0) + carry;
            c[i+j] = int (cur % base);
            carry = int (cur / base);
        }
    while (c.size() > 1 && c.back() == 0)         c.pop_back();
    return c;
}
VI& divionLS(VI &a, VI &b) {
    int carry = 0;
    for (int i=(int)a.size()-1; i>=0; --i) {
        long long cur = a[i] + carry * 1ll * base;
        a[i] = int (cur / b);
        carry = int (cur % b);
    }
    while (a.size() > 1 && a.back() == 0)   a.pop_back();
    return a;
}
```

Bit Operation

```
int Set(int MASK, int pos) { return MASK = MASK | (1 << pos); }
int reset(int MASK, int pos) { return MASK = MASK & ~(1 << pos); }
bool check(int MASK, int pos) { return (bool)(MASK & (1 << pos)); }
int count(int MASK){
    int count = 0;
    for (int pos = 0; (1 << pos) < MASK; pos++)
        if (check(MASK, pos))         count++;
    return count;
```

```
}
bool IsPowerOfTwo(int x) {      return (x && !(x & (x - 1)));   }
long long int LargestPowerOfTwo(long long int N) //lower_bound{
    N = N | (N >> 1);    N = N | (N >> 2);
    N = N | (N >> 4);    N = N | (N >> 8);
    return (N + 1) >> 1;
}
int next_popcount(int n){
    int c = (n & -n);          int r = n + c;
    return (((r ^ n) >> 2) / c) | r;
}
1. x ^ (x & (x - 1))       // Returns the rightmost 1 in binary representation of x
    1010 = 010 2. x & (-x) // Returns the rightmost 1 in binary representation of x
           1010 = 0010,
    1000 = 1000, 10101000 = 1000 3. x | (1 << n) // Returns the number x with the nth bit set
                            1010 = 1110
// Odd - Even checking ==>>
if (x & 1)
    -- > Odd else -- > Even
// 2^n dara gun or vag ==>>
gun-- > x << n
vag-- >
x >> n
// 2^n or 2 er power kina ==>>
if (x & (x - 1))
-- > 2 er power na else -- > 2 er power
// 2^n dara divisible naki ==>>
let, d = 2 ^ n
d = 8; // 8=2^3
if (x & (d - 1))
-- > x, d dara divisible else -- > x, d dara divisible na
//SWAP ==>>
int x, y;
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Digit

```
vector <int> prime;
bool islow(char ch){if(ch>='a' && ch<='z') return true; return false;}
bool isupp(char ch){if(ch>='A' && ch<='Z') return true; return false;}
bool isdig(char ch){if(ch>='0' && ch<='9') return true; return false;}
//any base to decimal conversion
int todec(string s, int base) {
    int i, j, temp, len, sum = 0;          len = s.length() - 1;
    for (i = 0, j = len; i <= len; i++, j--) {
        char ch = s.at(i);
        if (isdig(ch))          temp = ch - '0';
        else if (islow(ch))     temp = 10 + ch - 'a';
        else if (isupp(ch))     temp = 10 + ch - 'A';
        sum += (temp * (power(base, j)));
    }
    return sum;
}
//decimal to any base conversion
```

```
string tobase(int num, int base) {
    int temp;      string s;    char ch;
    if (!num)      return "0"; //special '0' case handling
    while (num > 0) {
        temp = num % base;      num /= base;
        if (temp <= 9)    s += (temp + '0');
        else              s += ((temp - 10) + 'A');
    }
    reverse(all(s));
    return s;
}
int numberOfDigit(int n, int base = 10) {
    int res = floor(log(n) / log(base));
    return (res + 1 + EPS);
}
int numberofDigitFactorial(int n, int base) {
    double x = 0.0;
    for (int i = 1; i <= n; i++)      x += log(i) / log(base);
    return (x + 1 + EPS);
}
// how many time p occurs in n!(p is prime)
ll factorialPrimePower(ll n, ll p) {
    ll freq = 0, x = n;
    while (x) {
        freq += x / p;          x /= p;
    }
    return freq;
}
//  (n!)^k
int fact_pow (int n, int k) {
    int res = 0;
    while (n) {
        n /= k;    res += n;
    }
    return res;
}
//  pascale triangle
const int maxn = ...;
int C[maxn + 1][maxn + 1];      C[0][0] = 1;
for (int n = 1; n <= maxn; ++n) {
    C[n][0] = C[n][n] = 1;
    for (int k = 1; k < n; ++k)       C[n][k] = C[n - 1][k - 1] + C[n - 1][k];
}
//  first k leading digit of n!
int leadingDigitofFactorial(int n, int k) {
    double fact = 0;
    for (int i = 1; i <= n; i++)      fact += log10(i);    //  log(n!)
    double q = fact - floor(fact + EPS);    double b = pow(10, q);
    for (int i = 0; i < k - 1; i++)   b *= 10;
    return floor(b + EPS);
}
void factorialFactorize(int n) {
    for (int i = 0; i < prime.size() && prime[i] <= n; i++) {
        int x = n, freq = 0;
        while (x / prime[i]) {
            freq += x / prime[i];     x /= prime[i];
```

```
        }
        cout << prime[i] << " " <<  freq << endl;
    }
}
```

Discrete Root

```
//  iven a prime n and two integers a and k, find all x for which: x^k≡a(mod n)
int gcd(int a, int b) {  return a ? gcd(b % a, a) : b; }
int powmod(int a, int b, int p){
    int res = 1;
    while (b > 0) {
        if (b & 1)       res = res * a % p;
        a = a * a % p;
        b >>= 1;
    }
    return res;
}
// Finds the primitive root modulo p
int generator(int p){
    vector<int> fact;
    int phi = p - 1, n = phi;
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) {
            fact.push_back(i);
            while (n % i == 0)       n /= i;
        }
    }
    if (n > 1)     fact.push_back(n);
    for (int res = 2; res <= p; ++res){
        bool ok = true;
        for (int factor : fact) {
            if (powmod(res, phi / factor, p) == 1) {
                ok = false;    break;
            }
        }
        if (ok)    return res;
    }
    return -1;
}
int main(){
    int n, k, a;
    scanf("%d %d %d", &n, &k, &a);
    if (a == 0) {
        puts("1\n0");    return 0;
    }
    int g = generator(n);
    // Baby-step giant-step discrete logarithm algorithm
    int sq = (int)sqrt(n + .0) + 1;
    vector<pair<int, int>> dec(sq);
    for (int i = 1; i <= sq; ++i)     dec[i - 1] = {powmod(g, i * sq * k % (n - 1), n), i};
    sort(dec.begin(), dec.end());
    int any_ans = -1;
    for (int i = 0; i < sq; ++i) {
        int my = powmod(g, i * k % (n - 1), n) * a % n;
        auto it = lower_bound(dec.begin(), dec.end(), make_pair(my, 0));
```

```
        if (it != dec.end() && it->first == my) {
            any_ans = it->second * sq - i;        break;
        }
    }
    if (any_ans == -1) {
        puts("0");       return 0;
    }
    // Print all possible answers
    int delta = (n - 1) / gcd(k, n - 1);
    vector<int> ans;
    for(int cur=any_ans%delta; cur<n-1; cur+=delta)     ans.push_back(powmod(g, cur, n));
    sort(ans.begin(), ans.end());
    printf("%d\n", ans.size());
    for (int answer : ans)     printf("%d ", answer);
}
```

Fast fourier Transform

```
typedef complex <long double> Complex;
typedef valarray <Complex> ValComplex;
const long double PI = 2 * acos(0.0);
void fft(ValComplex &p, bool inverse = 0) {
  int n = p.size();
  if(n <= 1)              return;
  ValComplex f = p[slice(0, n/2, 2)], g = p[slice(1, n/2, 2)];
  // splice(a, b, c) will return number in indexes a, a + c, a + 2c, .... a + (b-1)c
  fft(f, inverse); fft(g, inverse); // FFT for F and G
  Complex omega_n = exp(Complex(0, 2 * PI / n)), w = 1;
  if(inverse) omega_n = Complex(1, 0) / omega_n;
  for(int k = 0; k < n / 2; k++) {
    Complex add = w * g[k];   // Here w = omega_n^k
    p[k]       = f[k] + add;  // this is p(x)
    p[k + n/2] = f[k] - add;  // Note that p(-x) should be in (x+n/2)th position
    w *= omega_n;
  }
}
void ifft(ValComplex &p) {
  fft(p, 1);       p /= p.size(); // Divide each element by p.size()
}
vector<int> multiply(vector<int> a, vector<int> b) {
  int n = a.size(), m = b.size();
  int t = n + m - 1, sz = 1; // t is degree of R
  while(sz < t)    sz <<= 1;  // rounding to nearest 2^x
  ValComplex x(sz), y(sz), z(sz);
  // Resize first polynomial by inserting 0.
  for(int i = 0; i < n; i++)          x[i] = Complex(a[i], 0);
  for(int i = n; i < sz; i++)         x[i] = Complex(0, 0);
  // Resize second polynomial by inserting 0.
  for(int i = 0; i < m; i++)          y[i] = Complex(b[i], 0);
  for(int i = m; i < sz; i++)         y[i] = Complex(0, 0);=
  fft(x);          fft(y);   // Do fft on both polynomial
  // Multiply in Point-Value Form
  for(int i = 0; i < sz; i++)         z[i] = x[i] * y[i];
  ifft(z);  // Inverse FFT
  vector<int> res(sz);
  // Precision problem may occur, round to nearest integer
```

```
  for(int i = 0; i < sz; i++) res[i] = z[i].real() + 0.5;
  // remove trailing 0's
  while(res.size() > 1 && res.back() == 0)        res.pop_back();
  return res;
}
// cp-algorthms
using cd = complex<double>;
const double PI = acos(-1);
void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    if (n == 1)    return;
    vector<cd> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++) {
        a0[i] = a[2*i];  a1[i] = a[2*i+1];
    }
    fft(a0, invert);     fft(a1, invert);
    double ang = 2 * PI / n * (invert ? -1 : 1);
    cd w(1), wn(cos(ang), sin(ang));
    for (int i = 0; 2 * i < n; i++) {
        a[i] = a0[i] + w * a1[i];
        a[i + n/2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;   a[i + n/2] /= 2;
        }
        w *= wn;
    }
}
vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())   n <<= 1;
    fa.resize(n);        fb.resize(n);
    fft(fa, false);      fft(fb, false);
    for (int i = 0; i < n; i++)      fa[i] *= fb[i];
    fft(fa, true);
    vector<int> result(n);
    for (int i = 0; i < n; i++)      result[i] = round(fa[i].real());
    return result;
}
int carry = 0;
for (int i = 0; i < n; i++) {
    result[i] += carry;
    carry = result[i] / 10;
    result[i] %= 10;
}
using cd = complex<double>;
const double PI = acos(-1);
int reverse(int num, int lg_n) {
    int res = 0;
    for (int i = 0; i < lg_n; i++)
        if (num & (1 << i))    res |= 1 << (lg_n - 1 - i);
    return res;
}
void fft(vector<cd> & a, bool invert) {
    int n = a.size(), lg_n = 0;
    while ((1 << lg_n) < n)    lg_n++;
```

```
    for (int i = 0; i < n; i++)
        if (i < reverse(i, lg_n))     swap(a[i], a[reverse(i, lg_n)]);
    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }
    if (invert)
        for (cd & x : a)        x /= n;
}
using cd = complex<double>;
const double PI = acos(-1);
void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)    j ^= bit;
        j ^= bit;
        if (i < j)        swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }
    if (invert)
        for (cd & x : a)        x /= n;
}
const int mod = 7340033, root = 5, root_1 = 4404020, root_pw = 1 << 20;
void fft(vector<int> & a, bool invert) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)     j ^= bit;
        j ^= bit;
        if (i < j)        swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        int wlen = invert ? root_1 : root;
        for (int i = len; i < root_pw; i <<= 1)    wlen = (int)(1LL * wlen * wlen % mod);
```

```
        for (int i = 0; i < n; i += len) {
            int w = 1;
            for (int j = 0; j < len / 2; j++) {
                int u = a[i+j], v = (int)(1LL * a[i+j+len/2] * w % mod);
                a[i+j] = u + v < mod ? u + v : u + v - mod;
                a[i+j+len/2] = u - v >= 0 ? u - v : u - v + mod;
                w = (int)(1LL * w * wlen % mod);
            }
        }
    }
    if (invert) {
        int n_1 = inverse(n, mod);
        for (int & x : a)       x = (int)(1LL * x * n_1 % mod);
    }
}
```

GCD

```
vector <int> prime;
int gcd(int a, int b) {
    while (b) {
        a = a % b;        swap(a, b);
    }
    return a;
}
int lcm(int a, int b) {  return ((a / gcd(a, b)) * b);}
//  aX + bY = gcd(a, b)
//  (x, y) = (x + (kb) / gcd(a, b)   ,   y - (ka) / gcd(a, b))
int ext_GCD(int a, int b, int &X, int &Y) {
    int x, y, x1, y1, x2, y2, r, r1, r2, q;
    x1 = 0;      y1 = 1;
    x2 = 1;      y2 = 0;
    r1 = b;      r2 = a;
    for ( ; r1 != 0; ) {
        q = r2 / r1;      r = r2 % r1;
        x = x2 - (q * x1);      y = y2 - (q * x1);
        r2 = r1;   r1 = r;
        x2 = x1;   y2 = y1;
        x1 = x;    y1 = y;
    }
    X = x2;      Y = y2;
    return r2;
}
//  solve (x, y) for Ax + By = C
bool linearDiophantineEquation(int A, int B, int C, int &x, int&y) {
    int g = gcd(A, B);
    if (C % g != 0)       return false;
    int a = A / g, b = B / g, c = C / g;
    ext_GCD(a, b, x, y);    //  solve ax + by = 1
    if (g < 0) {
        a *= -1;    b *= -1;    c *= -1;
    }
    x *= c;      y *= c; //  ax + by = c
    return true;
}
//  simple Hyperbolic Diophantine Equation solve (x,y) for Axy+Bx+Cy=D= >(Ax+C) (Ay+B)=AD+BC
```

```
//  (x, y) = ((d - C) / A   ,   (P - Bd) / Ad) where P = AD + BC d is counted in res;
bool isValidSolution(int a, int b, int c, int p, int div) {
    if ( (div - c) % a != 0)    return false;    //  x = (div - c) / a
    if ( (p - b*div) % (a*div) != 0)        return false;    //  y = (p - b*div) / (a*div)
    return true;
}
int simpleHyperbolicDiophantineEquation(int a, int b, int c, int d) {
    int p = a*d + b*c;
    if (!p) {   //  ad + bc = 0
        if ((-c % a == 0) || (-b % a == 0))       return - 1;
        return 0;
    }
    int res = 0, sqrtp = sqrt(p), div;
    for (int i = 1; i <= sqrtp; i++) {
        if (p%i == 0) {
            res += isValidSolution(a, b, c, p, i) + isValidSolution(a, b, c, p, -i);
            res += ((p/i != i) * isValidSolution(a, b, c, p, p/i));
            res += ((p/i != i) * isValidSolution(a, b, c, p, -p/i));
        }
    }
    return res;
}
//  Euler Phi Function : count of numbers <= N that are coPrime with N
//  Number of elements e, such that gcd(e,n)=d is equal to φ(nd). ∑of (d/n)  [ ] = n.
int eulerPhi(int n) {
    int res = n, sqrtn = sqrt(n);
    for (int i = 0; i < prime.size() && prime[i] <= sqrtn; i++) {
        if (n % prime[i] == 0) {
            while (n % prime[i] == 0)       n /= prime[i];
            sqrtn = sqrt(n);
            res /= prime[i];    res *= prime[i] - 1;
        }
    }
    if (n != 1) {
        res /= n;        res *= n - 1;
    }
    return res;
}
//  returns (n^p) % mod
int bigMod(int n, int p, int mod ) {
    int res = 1%mod, x = n%mod;
    while (p) {
        if (p&1)   res = (res * x) % mod;
        x = (x * x) % mod;
        p >>= 1;
    }
    return res;
}
//  x = (1/a) % mod
int modInv(int a, int mod) {    return bigMod(a, mod - 2, mod); }//  mod is prime
int modInv2(int a, int mod) {   //  mod is not prime
    int x, y;
    ext_GCD(a, mod, x, y);
    x %= mod;
    if (x < 0)     x += mod;
    return x;
```

```
}
//  modular inverse of n
//  complexity = O(n)
int modInvArray[MX];
void allModInv(int n, int mod) {
    modInvArray[1] = 1;
    for (int i = 1; i <= n; i++) {
        modInvArray[i] = (-(mod / i) * modInvArray[mod % i]) % mod;
        modInvArray[i] += mod;
    }
}
//  return {-1. -1} if invalid input        return {x, l}   where x is unique
//  when mod by l [answer => x(MOD L)]      answer = x + L * k; k = 0,1,2,3...
//  complexity: O( nlog(L) )
PII ChineseRemainderTheorem(vector <int> A, vector <int> M) {
    if (A.size() != M.size())  return {-1, -1};
    int n = A.size(), a1 = A[0], m1 = M[0];
    for (int i = 1; i < n; i++) {
        int a2 = A[i], m2 = M[i];
        int g = gcd(m1, m2);
        if (a1%g != a2%g)       return {-1, -1};
        int p, q;
        ext_GCD(m1/g, m2/g, p, q);
        int mod = m1 / g * m2;
        int x = (a1 * (m2 / g) * q + a2 * (m1 / g) * p) % mod; //modify inCase Overflow
        a1 = x;
        if (a1 < 0)      a1 += mod;
        m1 = mod;
    }
    return PII(a1, m1);
}
int main() {
    cout << gcd(6, 8);
    //Linear Diophantine Equation
    int a = 2, b = 3, c = 5, x, y;    int g = gcd(a, b);
    if (linearDiophantineEquation(a, b, c, x, y))
        for (int k = 1; k <= 100; k++)
            cout << x + k * (b / g) << "  " << y - k * (a / g) << endl;
}
```

LCM Sum

```
//  lcm(1, n) + lcm(2, n) + .....lcm(n, n)
ll res[MX], phi[MX];
void preCalc(int n) {
    for (int i = 1; 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] /= i;   phi[h] *= i - 1;
            }
    for (int i = 1; i <= n; i++)
        for (int j = i; j <= n; j += i)     res[j] += (i * phi[i]);
}
int main() {
    preCalc(1000000);
```

```
    int n;  scanf("%d", &n);
    ll ans = res[n] + 1;        ans *= n;    ans /= 2;
    cout << ans << endl;
}
```

Mod Inverse

```
int modInverse(int a, int m){
    a %= m;
    for (int x = 1; x < m; x++)
        if ((a * x) % m == 1)    return x;
}
```

Montgomery Multiplication

```
//      Fast inverse trick
long long result = (__int128)x * y % n;
using u64 = uint64_t;
using u128 = __uint128_t;
using i128 = __int128_t;
struct u256 {
    u128 high, low;
    static u256 mult(u128 x, u128 y) {
        u64 a = x >> 64, b = x;
        u64 c = y >> 64, d = y;
        // (a*2^64 + b) * (c*2^64 + d) =
        // (a*c) * 2^128 + (a*d + b*c)*2^64 + (b*d)
        u128 ac = (u128)a * c;
        u128 ad = (u128)a * d;
        u128 bc = (u128)b * c;
        u128 bd = (u128)b * d;
        u128 carry = (u128)(u64)ad + (u128)(u64)bc + (bd >> 64u);
        u128 high = ac + (ad >> 64u) + (bc >> 64u) + (carry >> 64u);
        u128 low = (ad << 64u) + (bc << 64u) + bd;
        return {high, low};
    }
};
struct Montgomery {
    Montgomery(u128 n) : mod(n), inv(1) {
        for (int i = 0; i < 7; i++)  inv *= 2 - n * inv;
    }
    u128 init(u128 x) {
        x %= mod;
        for (int i = 0; i < 128; i++) {
            x <<= 1;
            if (x >= mod)       x -= mod;
        }
        return x;
    }
    u128 reduce(u256 x) {
        u128 q = x.low * inv;
        i128 a = x.high - u256::mult(q, mod).high;
        if (a < 0)       a += mod;
        return a;
    }
    u128 mult(u128 a, u128 b) {       return reduce(u256::mult(a, b)); }
    u128 mod, inv;
```

```
};
//      Fast transformation
struct Montgomery {
    Montgomery(u128 n) : mod(n), inv(1), r2(-n % n) {
        for (int i = 0; i < 7; i++)   inv *= 2 - n * inv;
        for (int i = 0; i < 4; i++) {
            r2 <<= 1;
            if (r2 >= mod)      r2 -= mod;
        }
        for (int i = 0; i < 5; i++)         r2 = mul(r2, r2);
    }
    u128 init(u128 x) {  return mult(x, r2); }
    u128 mod, inv, r2;
};
```

Prime

```
bool flag[MX];
vector<int> prime;
void SieveOfEratosthenes(int limit = MX){
    prime.clear();        flag[0] = flag[1] = 1;
    prime.push_back(2);
    for (int i = 4; i <= limit; i += 2)     flag[i] = 1;
    for (int i = 3; i * i < limit; i += 2)
        if (flag[i] == 0)
            for (int j = i * i; j <= limit; j += 2 * i)       flag[j] = 1;
    for (int i = 3; i <= limit; i += 2)
        if (flag[i] == 0)       prime.push_back(i);
}
bool SegmentedSieve_flag[MX];
vector<LLI> SegPrime;
void SegmentedSieve(LLI a, LLI b){
    SegPrime.clear();    prime.clear();
    SieveOfEratosthenes((int)sqrt(b));
    if (a == 1)    a++;
    for (LLI i = a + a & 1; i <= b; i += 2)       SegmentedSieve_flag[i] = 1;
    for (int i = 0; i < prime.size() && prime[i] * prime[i] <= b; i++){
        LLI p = prime[i];       LLI j = p * p;
        if (j < a)        j = ((a + p - 1) / p) * p;
        for (; j <= b; j += 2 * p)    SegmentedSieve_flag[j - a] = 1;
    }
    for (LLI i = a; i <= b; i++)
        if (!SegmentedSieve_flag[i - a])    SegPrime.push_back(i);
}
#define PII pair<int, int>
vector<PII> factors; //base, power i.e. (2^2) * (5^1) * (7^2)
void factorize(int n){
    int sqrtn = sqrt(n);
    for (int i = 0; i < prime.size() && prime[i] <= sqrtn; i++){
        if (n % prime[i] == 0) {
            int cnt = 0;
            while (n % prime[i] == 0) {
                n /= prime[i];        cnt++;
            }
            factors.push_back(PII(prime[i], cnt));
            sqrtn = sqrt(n);
```

```
        }
    }
    if (n != 1)    factors.push_back(PII(n, 1));
}
int NumberOfDivisors(int n){
    int res = 1;    prime.clear();
    SieveOfEratosthenes(n); //if use sqrt(n), remove flag[n] from loop cndition
    for (int i = 0; i < prime.size() && prime[i] * prime[i] <= n && flag[n]; i++)
        if (n % prime[i] == 0) {
            int cnt = 0;
            for (; n && n % prime[i] == 0; cnt++)    n /= prime[i];
            res *= cnt + 1;
        }
    if (n != 1)    res = res << 1;
    return res;
}
int resNum, resDiv, n;
//  A Highly Composite Number (HCN) is a positive integer
//  which has more divisors than any smaller positive integer
void HighleCompositeNumber(int pos, int limit, ll num, int div) {
    if (div > resDIv) {
        resNum = num;    resDiv = div;
    }
    else if (div == resDiv && num < resNum)       resNum = num;
    if (pos == 9)        return;
    ll p = prime[pos];
    for (int i = 1 i <= limit; i++) {
        if (num * p > n)       break;
        HighleCompositeNumber(pos + 1, i, num * p, div * (i+1));
        p *= prime[pos];
    }
}
//  complexity O(sqrt(n))
int SumofNumberOfDivisor(int n) {
    int res = 0, u = sqrt(n);
    for (int i = 1; i <= u; i++)     res += (n/i) - i;
    res *= 2;      res += u;
    return res;
}
int SumOfDivisor(int n) {
    int res = 1, sqrtn = sqrt(n);
    for (int i = 0; i < prime.size() && prime[i] <= sqrtn; i++) {
        if (n % prime[i] == 0) {
            int tempSum = 1, p = 1;=
            while (n % prime[i] == 0) {
                n /= prime[i];  p *= prime[i];
                tempSum += p;
            }
            sqrtn = sqrt(n);
            res *= tempSum;
        }
    }
    if (n != 1)    res *= n + 1;
    return res;
}
int main(){
```

```
    SieveOfEratosthenes();
    SegmentedSieve(100000, 200000);
    PrimeFactorization(980);
    int nod_252 = NumberOfDivisors(252);
    n = 1000000000;
    resNum = resDiv = 00
    HighleCompositeNumber(0, 30, 1, 1);
    printf("%d %D\n", resNum, resDiv);
}
```

primitive root

```
//  definiton : In modular arithmetic, a number g is called a primitive root modulo n
//  if every number coprime to n is congruent to a power of g modulo n.
//  Mathematically, g is a primitive root modulo n if and only if  for any integer a such
//  that gcd(a,n)=1, there exists an integer k such that: g^k≡a(mod n).
//  k is then called the index or discrete logarithm of a to the base g modulo n.
//  g is also called the generator of the multiplicative group of integers modulo n.
//  The following code assumes that the modulo p is a prime number.
//  To make it works for any value of p, we must add calculation of φ(p).
int powmod(int a, int b, int p){
    int res = 1;
    while (b)
        if (b & 1)       res = int(res * 1ll * a % p), --b;
        else             a = int(a * 1ll * a % p), b >>= 1;
    return res;
}
int generator(int p){
    vector<int> fact;
    int phi = p - 1, n = phi;
    for (int i = 2; i * i <= n; ++i)
        if (n % i == 0){
            fact.push_back(i);
            while (n % i == 0)       n /= i;
        }
    if (n > 1)     fact.push_back(n);
    for (int res = 2; res <= p; ++res) {
        bool ok = true;
        for (size_t i = 0; i < fact.size() && ok; ++i)
            ok &= powmod(res, phi / fact[i], p) != 1;
        if (ok)    return res;
    }
    return -1;
}
```

Simple Division

```
//Given an array of numbers, find the largest number d such that, when elements of the array
// are divided by d, they leave the same remainder gcd((a-b), (b-c), (c-d).....)
ll gcd ( ll a, ll b ) {
    while ( b ) {
        a = a % b;       swap ( a, b );
    }
    return a;
}
ll arr[1010];
```

```
int main () {
    while ( scanf ( "%d", &arr[0] ) != EOF ) {
        if ( arr[0] == 0 )      break;
        int cur = 1;
        while ( 1 ) {
            scanf ( "%lld", &arr[cur] );
            if ( arr[cur] == 0 )      break;
            else                      cur++;
        }
        ll g = 0; // Start with 0 since gcd(0,x) = x.
        for ( int i = 1; i < cur; i++ ) {
            int dif = arr[i] - arr[i-1]; // Calculate difference
            g = gcd ( g, dif ); // Find gcd() of differences
        }
        if ( g < 0 )      g *= -1; // In case gcd() comes out negative
        printf ( "%lld\n", g );
    }
}
```

Other

15 puzzle game

```
// This game is played on a 4×4 board. On this board there are 15 playing tiles numbered from
// 1 to 15. One cell is left empty (denoted by 0). You need to get the board to the position
// presented below by repeatedly moving one of the tiles to the free space:
int main() {
    int a[16];
    for (int i=0; i<16; ++i)   cin >> a[i];
    int inv = 0;
    for (int i=0; i<16; ++i)
        if (a[i])
            for (int j=0; j<i; ++j)
                if (a[j] > a[i])      ++inv;
    for (int i=0; i<16; ++i)
        if (a[i] == 0)   inv += 1 + i / 4;
    puts ((inv & 1) ? "No Solution" : "Solution Exists");
}
```

Game

```
//  Policeman and thief
vector<vector<int>> adj_rev;
vector<bool> winning, losing, visited;
vector<int> degree;
void dfs(int v) {
    visited[v] = true;
    for (int u : adj_rev[v]) {
        if (!visited[u]) {
            if (losing[v])                  winning[u] = true;
            else if (--degree[u] == 0)      losing[u] = true;
            else                            continue;
            dfs(u);
        }
    }
```

```
}
struct State {     int P, T;    bool Pstep;  };
vector<State> adj_rev[100][100][2]; // [P][T][Pstep]
bool winning[100][100][2], losing[100][100][2], visited[100][100][2];
int degree[100][100][2];
void dfs(State v) {
    visited[v.P][v.T][v.Pstep] = true;
    for (State u : adj_rev[v.P][v.T][v.Pstep]) {
        if (!visited[u.P][u.T][u.Pstep]) {
            if (losing[v.P][v.T][v.Pstep])              winning[u.P][u.T][u.Pstep] = true;
            else if (--degree[u.P][u.T][u.Pstep] == 0)  losing[u.P][u.T][u.Pstep] = true;
            else                                        continue;
            dfs(u);
        }
    }
}
int main() {
    int n, m;      cin >> n >> m;
    vector<string> a(n);
    for (int i = 0; i < n; i++)       cin >> a[i];
    for (int P = 0; P < n*m; P++) {
        for (int T = 0; T < n*m; T++) {
            for (int Pstep = 0; Pstep <= 1; Pstep++) {
                int Px = P/m, Py = P%m, Tx = T/m, Ty = T%m;
                if (a[Px][Py]=='*' || a[Tx][Ty]=='*')   continue;
                bool& win = winning[P][T][Pstep];
                bool& lose = losing[P][T][Pstep];
                if (Pstep) {
                    win = Px==Tx && Py==Ty;
                    lose = !win && a[Tx][Ty] == 'E';
                } else {
                    lose = Px==Tx && Py==Ty;
                    win = !lose && a[Tx][Ty] == 'E';
                }
                if (win || lose)     continue;
                State st = {P,T,!Pstep};
                adj_rev[P][T][Pstep].push_back(st);
                st.Pstep = Pstep;
                degree[P][T][Pstep]++;
                const int dx[] = {-1, 0, 1, 0, -1, -1, 1, 1};
                const int dy[] = {0, 1, 0, -1, -1, 1, -1, 1};
                for (int d = 0; d < (Pstep ? 8 : 4); d++) {
                    int PPx = Px, PPy = Py, TTx = Tx, TTy = Ty;
                    if (Pstep) {
                        PPx += dx[d];       PPy += dy[d];
                    } else {
                        TTx += dx[d];       TTy += dy[d];
                    }
                    if (PPx >= 0 && PPx < n && PPy >= 0 && PPy < m && a[PPx][PPy] != '*' &&
                        TTx >= 0 && TTx < n && TTy >= 0 && TTy < m && a[TTx][TTy] != '*') {
                        adj_rev[PPx*m+PPy][TTx*m+TTy][!Pstep].push_back(st);
                        ++degree[P][T][Pstep];
                    }
                }
            }
        }
```

```
    }
    for (int P = 0; P < n*m; P++)
        for (int T = 0; T < n*m; T++)
            for (int Pstep = 0; Pstep <= 1; Pstep++)
                if ((winning[P][T][Pstep] || losing[P][T][Pstep]) && !visited[P][T][Pstep])
                    dfs({P, T, (bool)Pstep});
    int P_st, T_st;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            if (a[i][j] == 'P')             P_st = i*m+j;
            else if (a[i][j] == 'T')        T_st = i*m+j;
    if (winning[P_st][T_st][true])          cout << "Police catches the thief"  << endl;
    else if (losing[P_st][T_st][true])      cout << "The thief escapes" << endl;
    else                                    cout << "Draw" << endl;
}
```

Integration

```
const int N = 1000 * 1000; // number of steps (already multiplied by 2)
double simpson_integration(double a, double b){
    double h = (b - a) / N;
    double s = f(a) + f(b); // a = x_0 and b = x_2n
    for (int i = 1; i <= N - 1; ++i) { // Refer to final Simpson's formula
        double x = a + h * i;
        s += f(x) * ((i & 1) ? 4 : 2);
    }
    s *= h / 3;
    return s;
}
```

Josephus

```
int josephus(int n, int k) {    return n > 1 ? (joseph(n-1, k) + k - 1) % n + 1 : 1; }
int josephus(int n, int k) {
    int res = 0;
    for (int i = 1; i <= n; ++i)      res = (res + k) % i;
    return res + 1;
}
int josephus(int n, int k) {
    if (n == 1)    return 0;
    if (k == 1)    return n-1;
    if (k > n)     return (joseph(n-1, k) + k) % n;
    int cnt = n / k;
    int res = joseph(n - cnt, k);
    res -= n % k;
    if (res < 0)   res += n;
    else           res += res / (k - 1);
    return res;
}
```

kth order statistic

```
//  Given an array A of size N and a number K. The challenge is to find K-
th largest number in the array, i.e., K-th order statistic.

template <class T>
T order_statistics (std::vector<T> a, unsigned n, unsigned k){
```

```
    using std::swap;
    for (unsigned l=1, r=n; ; ){
        if (r <= l+1) {
            // the current part size is either 1 or 2, so it is easy to find the answer
            if (r == l+1 && a[r] < a[l])    swap (a[l], a[r]);
            return a[k];
        }
        // ordering a[l], a[l+1], a[r]
        unsigned mid = (l + r) >> 1;
        swap (a[mid], a[l+1]);
        if (a[l] > a[r])        swap (a[l], a[r]);
        if (a[l+1] > a[r])      swap (a[l+1], a[r]);
        if (a[l] > a[l+1])      swap (a[l], a[l+1]);
        // performing division barrier is a[l + 1], i.e. median among a[l], a[l + 1], a[r]
        Unsigned i = l+1, j = r;
        const T cur = a[l+1];
        for (;;){
            while (a[++i] < cur) ;
            while (a[--j] > cur) ;
            if (i > j)   break;
            swap (a[i], a[j]);
        }
        // inserting the barrier
        a[l+1] = a[j];   a[j] = cur;
        // we continue to work in that part, which must contain the required element
        if (j >= k)      r = j-1;
        if (j <= k)      l = i;
    }
}
```

root by newton

```
double sqrt_newton(double n) {
    const double eps = 1E-15;
    double x = 1;
    for (;;) {
        double nx = (x + n / x) / 2;
        if (abs(x - nx) < eps)       break;
        x = nx;
    }
    return x;
}
int isqrt_newton(int n) {
    int x = 1;
    bool decreased = false;
    for (;;) {
        int nx = (x + n / x) >> 1;
        if (x == nx || nx > x && decreased)       break;
        decreased = nx < x;
        x = nx;
    }
    return x;
}
public static BigInteger isqrtNewton(BigInteger n) {
    BigInteger a = BigInteger.ONE.shiftLeft(n.bitLength() / 2);
    boolean p_dec = false;
```

```
    for (;;) {
        BigInteger b = n.divide(a).add(a).shiftRight(1);
        if (a.compareTo(b) == 0 || a.compareTo(b) < 0 && p_dec)      break;
        p_dec = a.compareTo(b) > 0;
        a = b;
    }
    return a;
}
```

sheduling job on 2 machine

```
struct Job {
    int a, b, idx;
    bool operator<(Job o) const {     return min(a, b) < min(o.a, o.b); }
};
vector<Job> johnsons_rule(vector<Job> jobs) {
    sort(jobs.begin(), jobs.end());
    vector<Job> a, b;
    for (Job j : jobs)
        if (j.a < j.b)   a.push_back(j);
        else             b.push_back(j);
    a.insert(a.end(), b.rbegin(), b.rend());
    return a;
}
pair<int, int> finish_times(vector<Job> const& jobs) {
    int t1 = 0, t2 = 0;
    for (Job j : jobs) {
        t1 += j.a;       t2 = max(t2, t1) + j.b;
    }
    return make_pair(t1, t2);
}
//  Optimal schedule of jobs given their deadlines and durations
struct Job {
    int deadline, duration, idx;
    bool operator<(Job o) const {     return deadline < o.deadline; }
};
vector<int> compute_schedule(vector<Job> jobs) {
    sort(jobs.begin(), jobs.end());
    set<pair<int,int>> s;
    vector<int> schedule;
    for (int i = jobs.size()-1; i >= 0; i--) {
        int t = jobs[i].deadline - (i ? jobs[i-1].deadline : 0);
        s.insert(make_pair(jobs[i].duration, jobs[i].idx));
        while (t && !s.empty()) {
            auto it = s.begin();
            if (it->first <= t) {
                t -= it->first;       schedule.push_back(it->second);
            } else {
                s.insert(make_pair(it->first - t, it->second));      t = 0;
            }
            s.erase(it);
        }
    }
    return schedule;
}
```

Stern-Brocot tree and Farey sequences

```
void build(int a = 0, int b = 1, int c = 1, int d = 0, int level = 1) {
    int x = a + c, y = b + d;
    // ... output the current fraction x/y at the current level in the tree
    build(a, b, x, y, level + 1);    build(x, y, c, d, level + 1);
}
//  Fraction Search Algorithm
string find(int x, int y, int a = 0, int b = 1, int c = 1, int d = 0) {
    int m = a + c, n = b + d;
    if (x == m && y == n)      return "";
    if (x*n < y*m)             return 'L' + find(x, y, a, b, m, n);
    else                       return 'R' + find(x, y, m, n, c, d);
}
```

terneray search

```
double ternary_search(double l, double r) {
    double eps = 1e-9;              //set the error limit here
    while (r - l > eps) {
        double m1 = l + (r - l) / 3, m2 = r - (r - l) / 3;
        double f1 = f(m1);      //evaluates the function at m1
        double f2 = f(m2);      //evaluates the function at m2
        if (f1 < f2)     l = m1;
        else             r = m2;
    }
    return f(l);                    //return the maximum of f(x) in [l, r]
}
```

String

Aho-Corasick

```
const int K = 26;
struct Vertex {
    int link = -1, go[K], p = -1, next[K];
    bool leaf = false;
    char pch;
    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);        fill(begin(go), end(go), -1);
    }
};
vector<Vertex> t(1);
void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();       t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].leaf = true;
}
```

```
int go(int v, char ch);
int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)    t[v].link = 0;
        else                          t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}
int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1)
        if (t[v].next[c] != -1)       t[v].go[c] = t[v].next[c];
        else                          t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    return t[v].go[c];
}
```

Expression parsing

```
bool delim(char c) {     return c == ' '; }
bool is_op(char c) {    return c == '+' || c == '-' || c == '*' || c == '/'; }
int priority (char op) {
    if (op == '+' || op == '-')       return 1;
    if (op == '*' || op == '/')       return 2;
    return -1;
}
void process_op(stack<int>& st, char op) {
    int r = st.top(); st.pop();
    int l = st.top(); st.pop();
    switch (op) {
        case '+': st.push(l + r); break;
        case '-': st.push(l - r); break;
        case '*': st.push(l * r); break;
        case '/': st.push(l / r); break;
    }
}
int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))       continue;
        if (s[i] == '(')    op.push('(');
        else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());         op.pop();
            }
            op.pop();
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            while (!op.empty() && priority(op.top()) >= priority(cur_op)) {
            // replace with
            // while (!op.empty() && ( (left_assoc(cur_op) && priority(op.top()) >=
            // priority(cur_op)) ||  (!left_assoc(cur_op) && priority(op.top()) >
            // priority(cur_op)))) for right associative
                process_op(st, op.top());
                op.pop();
            }
```

```
            op.push(cur_op);
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnum(s[i]))  number = number * 10 + s[i++] - '0';
            --i;
            st.push(number);
        }
    }
    while (!op.empty()) {
        process_op(st, op.top());     op.pop();
    }
    return st.top();
}
bool delim(char c) {     return c == ' '; }//   + - / *
bool is_op(char c) {    return c == '+' || c == '-' || c == '*' || c == '/'; }
bool is_unary(char c) {    return c == '+' || c=='-'; }
int priority (char op) {
    if (op < 0)          return 3;  // unary operator
    if (op == '+' || op == '-')      return 1;
    if (op == '*' || op == '/')      return 2;
    return -1;
}
void process_op(stack<int>& st, char op) {
    if (op < 0) {
        int l = st.top(); st.pop();
        switch (-op) {
            case '+': st.push(l); break;
            case '-': st.push(-l); break;
        }
    } else {
        int r = st.top(); st.pop();
        int l = st.top(); st.pop();
        switch (op) {
            case '+': st.push(l + r); break;
            case '-': st.push(l - r); break;
            case '*': st.push(l * r); break;
            case '/': st.push(l / r); break;
        }
    }
}
int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    bool may_be_unary = true;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))       continue;
        if (s[i] == '(') {
            op.push('(');
            may_be_unary = true;
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());   op.pop();
            }
            op.pop();
            may_be_unary = false;
        } else if (is_op(s[i])) {
```

```
            char cur_op = s[i];
            if (may_be_unary && is_unary(cur_op))       cur_op = -cur_op;
            while (!op.empty() && ( (cur_op >= 0 && priority(op.top()) >= priority(cur_op))
                   ||(cur_op < 0 && priority(op.top()) > priority(cur_op)))) {
                process_op(st, op.top());
                op.pop();
            }
            op.push(cur_op);
            may_be_unary = true;
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnum(s[i]))  number = number * 10 + s[i++] - '0';
            --i;
            st.push(number);
            may_be_unary = false;
        }
    }
    while (!op.empty()) {
        process_op(st, op.top());    op.pop();
    }
    return st.top();
}
```

Find Repetation

```
vector<int> z_function(string const& s) {
    int n = s.size();
    vector<int> z(n);
    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;
        }
    }
    return z;
}
int get_z(vector<int> const& z, int i) {
    if (0 <= i && i < (int)z.size())        return z[i];
    else                                    return 0;
}
vector<pair<int, int>> repetitions;
void convert_to_repetitions(int shift, bool left, int cntr, int l, int k1, int k2) {
    for (int l1 = max(1, l - k2); l1 <= min(l, k1); l1++) {
        if (left && l1 == l) break;
        int l2 = l - l1, pos = shift + (left ? cntr - l1 : cntr - l - l1 + 1);
        repetitions.emplace_back(pos, pos + 2*l - 1);
    }
}
void find_repetitions(string s, int shift = 0) {
    int n = s.size();
    if (n == 1)    return;
    int nu = n / 2, nv = n - nu;
    string u = s.substr(0, nu), v = s.substr(nu);
    string ru(u.rbegin(), u.rend()), rv(v.rbegin(), v.rend());
    find_repetitions(u, shift);    find_repetitions(v, shift + nu);
```

```
    vector<int> z1 = z_function(ru), z2 = z_function(v + '#' + u);
    vector<int> z3 = z_function(ru + '#' + rv), z4 = z_function(v);
    for (int cntr = 0; cntr < n; cntr++) {
        int l, k1, k2;
        if (cntr < nu) {
            l = nu - cntr;
            k1 = get_z(z1, nu - cntr);      k2 = get_z(z2, nv + 1 + cntr);
        } else {
            l = cntr - nu + 1;
            k1 = get_z(z3, nu + 1 + nv - 1 - (cntr - nu));
            k2 = get_z(z4, (cntr - nu) + 1);
        }
        if (k1 + k2 >= l)      convert_to_repetitions(shift, cntr < nu, cntr, l, k1, k2);
    }
}
```

Hash

```
//  hash(s) = [s[0]+s[1]·p+s[2]·p2+...+s[n-1]·p^(n-1)](mod m)
long long compute_hash(string const& s) {
    const int p = 31, m = 1e9 + 9;
    long long hash_value = 0, p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
vector<vector<int>> group_identical_strings(vector<string> const& s) {
    int n = s.size();
    vector<pair<long long, int>> hashes(n);
    for (int i = 0; i < n; i++)      hashes[i] = {compute_hash(s[i]), i};
    sort(hashes.begin(), hashes.end());
    vector<vector<int>> groups;
    for (int i = 0; i < n; i++) {
        if (i == 0 || hashes[i].first != hashes[i-1].first)    groups.emplace_back();
        groups.back().push_back(hashes[i].second);
    }
    return groups;
}
int count_unique_substrings(string const& s) {
    int n = s.size();
    const int p = 31, m = 1e9 + 9;
    vector<long long> p_pow(n);
    p_pow[0] = 1;
    for (int i = 1; i < n; i++)      p_pow[i] = (p_pow[i-1] * p) % m;
    vector<long long> h(n + 1, 0);
    for (int i = 0; i < n; i++)      h[i+1] = (h[i] + (s[i] - 'a' + 1) * p_pow[i]) % m;
    int cnt = 0;
    for (int l = 1; l <= n; l++) {
        set<long long> hs;
        for (int i = 0; i <= n - l; i++) {
            long long cur_h = (h[i + l] + m - h[i]) % m;
            cur_h = (cur_h * p_pow[n-i-1]) % m;
            hs.insert(cur_h);
        }
```

```
        cnt += hs.size();
    }
    return cnt;
}
```

KMP

```
int lps[1000005]; //for prefix of suffix of pattern
vector<int> cnt;
void lps_array(string pattern) {//time : o(n)
    int i = 1, j = 0;
    lps[0] = 0;
    while (i < pattern.length()){
        if (pattern[j] == pattern[i])       lps[i++] = ++j;
        else if (j)                         j = lps[j - 1];
        else                                lps[i++] = 0;
    }
}
int Search(string text, string pattern) //time : o(n){
    cnt.clear();
    lps_array(pattern);

    int i = 0, j = 0; //i for text & j for pattern
    while (i < text.length()){
        if (pattern[j] == text[i]) {
            i++;         j++;
        }
        if (j == pattern.length()){
            cnt.push_back(i - j);     j = lps[j - 1];
        }
        if (pattern[j] != text[i])
            if (j)       j = lps[j - 1];
            else         i++;
    }
    if (cnt.size())      return cnt[0];
    else                 return -1;
}
int main(){
    string text = "ABABDABACDABABCABAB";
    cout << Search(text, "ABABCABAB") << endl;
    text = "bcabdabdababdababd";
    cout << Search(text, "abdababd") << endl;
    for (int i = 0; i < cnt.size(); i++)    cout << cnt[i] << "  ";
}
```

Knuth–Morris–Pratt

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])       j = pi[j-1];
        if (s[i] == s[j])      j++;
        pi[i] = j;
    }
```

```
    return pi;
}
void compute_automaton(string s, vector<vector<int>>& aut) {
    s += '#';
    int n = s.size();
    vector<int> pi = prefix_function(s);
    aut.assign(n, vector<int>(26));
    for (int i = 0; i < n; i++)
        for (int c = 0; c < 26; c++)
            if (i > 0 && 'a' + c != s[i])   aut[i][c] = aut[pi[i-1]][c];
            else                            aut[i][c] = i + ('a' + c == s[i]);
}
```

Lundon

```
vector<string> duval(string const& s) {
    int n = s.size(), i = 0;
    vector<string> factorization;
    while (i < n) {
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])   k = i;
            else               k++;
            j++;
        }
        while (i <= k) {
            factorization.push_back(s.substr(i, j - k));      i += j - k;
        }
    }
    return factorization;
}
string min_cyclic_string(string s) {
    s += s;
    int n = s.size(), i = 0, ans = 0;
    while (i < n / 2) {
        ans = i;
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])   k = i;
            else               k++;
            j++;
        }
        while (i <= k)   i += j - k;
    }
    return s.substr(ans, n / 2);
}
```

Manacher's Algorithm

```
//  find all sub plaindrome in O(n)
int main() {
    vector<int> d1(n);
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
        while (0 <= i - k && i + k < n && s[i - k] == s[i + k])      k++;
        d1[i] = k--;
        if (i + k > r) {
```

```
            l = i - k;          r = i + k;
        }
    }
    vector<int> d2(n);
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
        while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k])    k++;
        d2[i] = k--;
        if (i + k > r) {
            l = i - k - 1;      r = i + k ;
        }
    }
}
```

Rabin Karp

```
#define PI 2.0 * acos(0.0)
vector<int> rabin_karp(string const& s, string const& t) {
    const int p = 31, m = 1e9 + 9;
    int S = s.size(), T = t.size();
    vector<long long> p_pow(max(S, T));
    p_pow[0] = 1;
    for (int i = 1; i < (int)p_pow.size(); i++)    p_pow[i] = (p_pow[i-1] * p) % m;
    vector<long long> h(T + 1, 0);
    for (int i = 0; i < T; i++)    h[i+1] = (h[i] + (t[i] - 'a' + 1) * p_pow[i]) % m;
    long long h_s = 0;
    for (int i = 0; i < S; i++)    h_s = (h_s + (s[i] - 'a' + 1) * p_pow[i]) % m;
    vector<int> occurences;
    for (int i = 0; i + S - 1 < T; i++) {
        long long cur_h = (h[i+S] + m - h[i]) % m;
        if (cur_h == h_s * p_pow[i] % m)    occurences.push_back(i);
    }
    return occurences;
}
```

Suffix Array

```
int m, SA[MX], LCP[MX], suffix[MX], index[MX], cnt[MX], rank[MX];
inline bool cmp(const int a, const int b, const int l){
    return (index[a] == index[b] && index[a + l] == index[b + l]);
}
void Sort(int len){
    for (int i = 0; i < 256; i++)           cnt[i] = 0;
    for (int i = 0; i < len; i++)           cnt[suffix[index[i]]]++;
    for (int i = 0; i < 255; i++)           cnt[i + 1] += cnt[i];
    for (int i = len - 1; i >= 0; i--)      SA[--cnt[suffix[index[i]]]] = index[i];
}
void kasaiLCP(string text){
    int len = text.length();
    for (int i = 0; i < len; i++)    rank[SA[i]] = i;
    LCP[len - 1] = 0;
    for (int i = 0, h = 0; i < len; i++)
        if (rank[i] > 0) {
            int j = SA[rank[i] - 1];
            while (i + h < len && j + h < len && text[i + h] == text[j + h])     h++;
            LCP[rank[i] - 1] = h;
            if (h > 0)   h--;
```

```
        }
}
void SuffixArray(string text){
    int len = text.length() + 1;
    for (int i = 0; i < len; i++){
        suffix[i] = text[i];    index[i] = i;
    }
    Sort(len);
    for (int i, j = 1, p = 1; p < len; j <<= 1, m = p){
        for (p = 0, i = len - j; i < len; i++)    index[p++] = i;
        for (int k = 0; k < len; k++)
            if (SA[k] >= j)    index[p++] = SA[k] - j;
        Sort(len);
        swap(suffix, index);
        suffix[SA[0]] = 0;
        for (p = 1, i = 1; i < len; i++)
            suffix[SA[i]] = cmp(SA[i - 1], SA[i], j) ? p - 1 : p++;
    }
    for (int i = 1; i < len; i++)    SA[i - 1] = SA[i];
    kasaiLCP(text);
}
int main(){
    string text="banana";      SuffixArray(text);
}
```

Suffix Automation

```
struct state {    int len, link;    map<char, int> next; };
const int MAXLEN = 100000;
state st[MAXLEN * 2];
int sz, last;
void sa_init() {
    st[0].len = 0;       st[0].link = -1;
    sz++;    last = 0;
}
void sa_extend(char c) {
    int cur = sz++;
    st[cur].len = st[last].len + 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 {
            int clone = sz++;
            st[clone].len = st[p].len + 1;
            st[clone].next = st[q].next;
            st[clone].link = st[q].link;
            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;
}
// output all positions of occurrences
void output_all_occurrences(int v, int P_length) {
    if (!st[v].is_clone)       cout << st[v].first_pos - P_length + 1 << endl;
    for (int u : st[v].inv_link)     output_all_occurrences(u, P_length);
}
string lcs (string S, string T) {
    sa_init();
    for (int i = 0; i < S.size(); i++)      sa_extend(S[i]);
    int v = 0, l = 0, best = 0, bestpos = 0;
    for (int i = 0; i < T.size(); i++) {
        while (v && !st[v].next.count(T[i])) {
            v = st[v].link ;    l = st[v].length ;
        }
        if (st[v].next.count(T[i])) {
            v = st [v].next[T[i]];    l++;
        }
        if (l > best) {
            best = l;    bestpos = i;
        }
    }
    return t.substr(bestpos - best + 1, best);
}
```