

## Contents

Preface .....	2	Fast Fourier Transform .....	13
Template .....	2	Karatsuba .....	14
C++ .....	2	Simplex .....	14
Graph Theory .....	2	Gauss Jordan Elimination .....	15
Articulation Point .....	2	Reduced Row Echelon Form .....	16
Articulation Bridge .....	3	Data Structures .....	17
Tarjan's Directed SCC .....	3	K-d Tree .....	17
Max Flow .....	4	Fenwick Tree .....	19
Max Flow Min Cost .....	4	Splay Tree .....	19
Lowest Common Ancestor .....	5	DP Convex Hull Optimization .....	20
Blossom .....	6	Geometry .....	21
Minimum Cut .....	7	Point, Segment, Line, Circle .....	21
String Processing .....	7	Polygons (Area, Orientation) .....	22
Knuth-Morris-Pratt .....	7	Convex Hull .....	23
Z-Algorithm .....	8	Dealunay Triangulation .....	23
Suffix Array .....	8	Miscellaneous .....	23
Suffix Tree .....	8	Graph Theorems .....	23
Aho-Corasick .....	10	Combinatorics .....	24
Mathematics .....	11	Notes .....	24
Extended Euclid .....	11		
Diophantine .....	11		
Chinese Remainder Theorem .....	11		
Rabin Miller Primality Test .....	11		
Lagrange Interpolation .....	12		

## Preface

This is our Team Notebook for ACM ICPC and other Competitive Programming contests. Notable sources are:

- Introduction to Algorithm 3<sup>rd</sup> edition
- Competitive Programming 2 by Felix and Steven Halim
- Topcoder Algorithm Tutorials
- <https://sites.google.com/site/indy256/>
- <http://stanford.edu/~lisz90/acm/notebook.html>
- Dongskar Pedongi and DELAPAN.3gp Team Notebook
- Google, Wikipedia

Regards,

hehehe

Aufar Gilbran, Ahmad Zaky, Willy

Institut Teknologi Bandung, Indonesia

## Template

### C++

```
#include <bits/stdc++.h> <vector> <map> <set> <queue> <deque> <stack>
<algorithm> <sstream> <iostream> <iomanip> <fstream> <cstring> <cmath>
<cstdlib> <ctime> <cassert> <limits> <numeric> <utility>
using namespace std;

#ifdef DEBUG
    #define debug(...) printf( VA ARGS )
    #define GetTime() fprintf(stderr,"Running time: %.3lf
second\n", ((double)clock())/CLOCKS_PER_SEC)
#else
    #define debug(...)
    #define GetTime()
#endif

//type definitions
typedef long long ll;
typedef double db;
typedef pair<int,int> pii;
typedef vector<int> vint;

//abbreviations
#define A first
#define B second
#define F first
#define S second
#define MP make_pair
```

```
#define PB push_back

//macros
#define REP(i,n) for (int i = 0; i < (n); ++i)
#define REPD(i,n) for (int i = (n)-1; 0 <= i; --i)
#define FOR(i,a,b) for (int i = (a); i <= (b); ++i)
#define FORD(i,a,b) for (int i = (a); (b) <= i; --i)
#define FORIT(it,c) for (__typeof ((c).begin()) it = (c).begin(); it !=
(c).end(); it++)
#define ALL(a) (a).begin(), (a).end()
#define SZ(a) ((int)(a).size())
#define RESET(a,x) memset(a,x,sizeof(a))
#define EXIST(a,s) ((s).find(a) != (s).end())
#define MX(a,b) a = max((a), (b));
#define MN(a,b) a = min((a), (b));

inline void OPEN(const string &s) {
    freopen((s + ".in").c_str(), "r", stdin);
    freopen((s + ".out").c_str(), "w", stdout);
}

/* ----- end of template ----- */
```

## Graph Theory

### Articulation Point

```
/** Articulation Point */
/* complexity : O(|V| + |E|) */

#define MAXN 100100

int n, m, low[MAXN], num[MAXN], parent[MAXN], art[MAXN], root,
rootChildren, counter;
vector<int> adj[MAXN];

void dfs(int u) {
    low[u] = num[u] = counter++;
    FORIT(it, adj[u]) {
        int v = *it;
        if (num[v] == -1) {
            parent[v] = u;
            if (u == root) rootChildren++;
            dfs(v);
            if (low[v] >= num[u]) art[u] = 1;
            MN(low[u], low[v]);
        }
        else if (v != parent[u]) {
            MN(low[u], num[v]);
        }
    }
}
```

```

int main() {
    // read the graph here. It should be 0-indexed
    // initialization
    counter = 0;
    REP(i, n) {
        num[i] = -1;
        low[i] = parent[i] = art[i] = 0;
    }
    // perform the dfs
    REP(i, n) {
        if (num[i] == -1) {
            root = i, rootChildren = 0;
            dfs(i);
            art[root] = (rootChildren > 1);
        }
    }
    // now the articulation points are stored in art[]
    return 0;
}

```

### Articulation Bridge

```

/** Bridge **/
/* complexity : O(|V| + |E| + |E| log |E|) */

#define MAXN 100100

int n, low[MAXN], num[MAXN], parent[MAXN], bridge[MAXN], counter;
vector<pii> adj[MAXN]; // adj[u].PB(MP(v, idx_of_edge));

void dfs(int u) {
    low[u] = num[u] = counter++;
    FORIT(it, adj[u]) {
        int v = it->A;
        if (num[v] == -1) {
            parent[v] = u;
            dfs(v);
            if (low[v] > num[u]) bridge[it->B] = 1;
            MN(low[u], low[v]);
        }
        else if (v != parent[u]) {
            MN(low[u], num[v]);
        }
    }
}

int main() {
    // read the graph here. it should be 0-indexed
    // should not work if multiple edges exist
    // initialization
    counter = 0;
    REP(i, n) {

```

```

        num[i] = -1;
        low[i] = parent[i] = 0;
    }
    REP(i, m) {
        bridge[i] = 0;
    }
    // perform the dfs
    REP(i, n) {
        if (num[i] == -1) {
            dfs(i);
        }
    }
    // the bridges are stored in bridge[]

    return 0;
}

```

### Tarjan's Directed SCC

```

/** Tarjan's Directed Strongly Connected Component **/
/* complexity : O(|V| + |E|) */

#define MAXN 100100

int n, low[MAXN], num[MAXN], visited[MAXN], counter;
vector<int> adj[MAXN], s;
vector<vector<int>> scc;

void dfs(int u) {
    low[u] = num[u] = counter++;
    s.PB(u);
    visited[u] = 1;
    FORIT(it, adj[u]) {
        int v = *it;
        if (num[v] == -1) dfs(v);
        if (visited[v]) {
            MN(low[u], low[v]);
        }
    }
    if (low[u] == num[u]) {
        vector<int> temp;
        int v = -1;
        while (u != v) {
            v = s.back(); s.pop_back(); visited[v] = 0;
            temp.PB(v);
        }
        scc.PB(temp);
    }
}

int main() {
    // read the graph here. it should be 0-indexed
    // initialization

```

```

counter = 0;
scc.clear();
REP(i, n) {
    num[i] = -1;
    low[i] = visited[i] = 0;
}
// perform the dfs
REP(i, n) {
    if (num[i] == -1) {
        dfs(i);
    }
}
// the components are stored in scc
return 0;
}

```

## Max Flow

```

#define MAXN 1100
#define INF 0x3FFFFFFF

int res[MAXN][MAXN], vis[MAXN];

/** Maximum Flow **/
/* Edmond Karp | complexity : O(|V|*(|V|+|E|)) */
void augment(int v, int minEdge, int &s, int &f, vector<int> &p){
    if (v == s) { f = minEdge; return; }
    else if (p[v] != -1) {
        augment(p[v], min(minEdge, res[p[v]][v]), s, f, p); res[p[v]][v] -= f;
        res[v][p[v]] += f;
    }
}

int maxFlowEdmondKarp(int n, int source, int target) {
    int mf = 0;
    while (1) {
        int f = 0;
        vector<int> dist(n+5, INF);
        dist[source] = 0;
        queue<int> q; q.push(source);
        vector<int> p; p.assign(n+5, -1);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            if (u == target) break;
            for (int v = 0; v < n; v++)
                if (res[u][v] > 0 && dist[v] == INF)
                    dist[v] = dist[u] + 1, q.push(v), p[v] = u;
            augment(target, INF, source, f, p);
            if (f == 0) break;
            mf += f;
        }
        return mf;
    }
}

```

```

/* Ford Fulkerson | complexity : O(|V|^2 F) */
int findPath(int n, int u, int t, int f){
    if (u == t) return f;
    vis[u] = 1;
    for (int v = 0; v < n; ++v){
        if (!vis[v] && res[u][v] > 0){
            int df = findPath(n, v, t, min(f, res[u][v]));
            if (df > 0){
                res[u][v] -= df;
                res[v][u] += df;
                return df;
            }
        }
    }
    return 0;
}

int maxFlowFordFulkerson(int n, int source, int target) {
    for (int flow = 0;;){
        for (int i = 0; i < n; ++i) vis[i] = 0;
        int df = findPath(n, source, target, INF);
        if (df == 0) return flow;
        flow += df;
    }
}

/* WARNING: res will be modified during the process */

```

## Max Flow Min Cost

```

/** Max Flow Min Cost **/
/* complexity: O(min(E^2 V log V, E log V F)) */
const int maxnodes = 200000;

int nodes = maxnodes;
int prio[maxnodes], curflow[maxnodes], prevedge[maxnodes],
prevnode[maxnodes], q[maxnodes], pot[maxnodes];
bool inqueue[maxnodes];

struct Edge {
    int to, f, cap, cost, rev;
};

vector<Edge> graph[maxnodes];

void addEdge(int s, int t, int cap, int cost) {
    Edge a = {t, 0, cap, cost, graph[t].size()};
    Edge b = {s, 0, 0, -cost, graph[s].size()};
    graph[s].push_back(a);
    graph[t].push_back(b);
}

void bellmanFord(int s, int dist[]) {
    fill(dist, dist + nodes, 1000000000);
}

```

```

dist[s] = 0;
int qt = 0;
q[qt++] = s;
for (int qh = 0; (qh - qt) % nodes != 0; qh++) {
    int u = q[qh % nodes];
    inqueue[u] = false;
    for (int i = 0; i < (int) graph[u].size(); i++) {
        Edge &e = graph[u][i];
        if (e.cap <= e.f) continue;
        int v = e.to;
        int ndist = dist[u] + e.cost;
        if (dist[v] > ndist) {
            dist[v] = ndist;
            if (!inqueue[v]) {
                inqueue[v] = true;
                q[qt++] % nodes = v;
            }
        }
    }
}

pii minCostFlow(int s, int t, int maxf) {
    // bellmanFord can be safely commented if edges costs are non-negative
    bellmanFord(s, pot);
    int flow = 0;
    int flowCost = 0;
    while (flow < maxf) {
        priority_queue<ll, vector<ll>, greater<ll>> > q;
        q.push(s);
        fill(prio, prio + nodes, 1000000000);
        prio[s] = 0;
        curflow[s] = 1000000000;
        while (!q.empty()) {
            ll cur = q.top();
            int d = cur >> 32;
            int u = cur;
            q.pop();
            if (d != prio[u]) continue;
            for (int i = 0; i < (int) graph[u].size(); i++) {
                Edge &e = graph[u][i];
                int v = e.to;
                if (e.cap <= e.f) continue;
                int nprio = prio[u] + e.cost + pot[u] - pot[v];
                if (prio[v] > nprio) {
                    prio[v] = nprio;
                    q.push(((ll) nprio << 32) + v);
                    prevnode[v] = u;
                    prevedge[v] = i;
                    curflow[v] = min(curflow[u], e.cap - e.f);
                }
            }
        }
    }
}

```

```

if (prio[t] == 1000000000) break;
for (int i = 0; i < nodes; i++) pot[i] += prio[i];
int df = min(curflow[t], maxf - flow);
flow += df;
for (int v = t; v != s; v = prevnode[v]) {
    Edge &e = graph[prevnode[v]][prevedge[v]];
    e.f += df;
    graph[v][e.rev].f -= df;
    flowCost += df * e.cost;
}
}
return make_pair(flow, flowCost);
}

/* usage example:
* addEdge(source, target, capacity, cost)
* minCostFlow(source, target, INF) -> <flow, flowCost>
*/

```

### Lowest Common Ancestor

```

/** Lowest Common Ancestor */
/* complexity : LCApre : O(N log N), LCAquery : O(log N) */
/* legend:
* N : number of vertices. WARNING: zero based
* T : direct parent. T[v] is parent of v
* L : L[v] is the level of v. zero/one based is okay
* P : dp table of size [MAXN][LOGMAXN]. P[v][i] is the 2^i-th parent of v
*/

#define MAXN 100100
#define LOGMAXN 18

int L[MAXN], P[MAXN][LOGMAXN], T[MAXN], N;

void pre() {
    int i, j;

    //we initialize every element in P with -1
    for (i = 0; i < N; i++) {
        for (j = 0; 1 << j < N; j++) {
            P[i][j] = -1;
        }
    }

    //the first ancestor of every node i is T[i]
    for (i = 0; i < N; i++) {
        P[i][0] = T[i];
    }

    //bottom up dynamic programming
    for (j = 1; 1 << j < N; j++) {
        for (i = 0; i < N; i++) {

```

```

        if (P[i][j - 1] != -1) {
            P[i][j] = P[P[i][j - 1]][j - 1];
        }
    }
}

int query(int p, int q){
    int log, i;

    //if p is situated on a higher level than q then we swap them
    if (L[p] < L[q]) {
        swap(p,q);
    }

    //we compute the value of [log(L[p])
    for (log = 1; 1 << log <= L[p]; log++);
    log--;

    //we find the ancestor of node p situated on the same level
    //with q using the values in P
    for (i = log; i >= 0; i--) {
        if (L[p] - (1 << i) >= L[q]) {
            p = P[p][i];
        }
    }

    if (p == q) return p;

    //we compute LCA(p, q) using the values in P
    for (i = log; i >= 0; i--) {
        if (P[p][i] != -1 && P[q][i] != P[p][i]) {
            p = P[p][i];
            q = P[q][i];
        }
    }

    return T[p];
}

```

## Blossom

```

/** Maximum Matching on General Graph */
/* Blossom | O(V^3) */

int lca(vector<int> &match, vector<int> &base, vector<int> &p, int a, int
b) {
    vector<bool> used(SZ(match));
    while (true) {
        a = base[a];
        used[a] = true;
        if (match[a] == -1) break;
        a = p[match[a]];
    }
}

```

```

    }
    while (true) {
        b = base[b];
        if (used[b]) return b;
        b = p[match[b]];
    }
    return -1;
}

void markPath(vector<int> &match, vector<int> &base, vector<bool> &blossom,
vector<int> &p, int v, int b, int children) {
    for (; base[v] != b; v = p[match[v]]) {
        blossom[base[v]] = blossom[base[match[v]]] = true;
        p[v] = children;
        children = match[v];
    }
}

int findPath(vector<vector<int> > &graph, vector<int> &match, vector<int>
&p, int root) {
    int n = SZ(graph);
    vector<bool> used(n);
    FORIT(it, p) *it = -1;
    vector<int> base(n);
    for (int i = 0; i < n; ++i) base[i] = i;

    used[root] = true;
    int qh = 0;
    int qt = 0;
    vector<int> q(n);
    q[qt++] = root;
    while (qh < qt) {
        int v = q[qh++];
        FORIT(it, graph[v]) {
            int to = *it;
            if (base[v] == base[to] || match[v] == to) continue;
            if (to == root || match[to] != -1 && p[match[to]] != -1) {
                int curbase = lca(match, base, p, v, to);
                vector<bool> blossom(n);
                markPath(match, base, blossom, p, v, curbase, to);
                markPath(match, base, blossom, p, to, curbase, v);
                for (int i = 0; i < n; ++i) {
                    if (blossom[base[i]]) {
                        base[i] = curbase;
                        if (!used[i]) {
                            used[i] = true;
                            q[qt++] = i;
                        }
                    }
                }
            }
        }
    }
    } else if (p[to] == -1) {
        p[to] = v;
        if (match[to] == -1) return to;
    }
}

```

```

        to = match[to];
        used[to] = true;
        q[qt++] = to;
    }
}
return -1;
}

int maxMatching(vector<vector<int> > graph) {
    int n = SZ(graph);
    vector<int> match(n, -1);
    vector<int> p(n);
    for (int i = 0; i < n; ++i) {
        if (match[i] == -1) {
            int v = findPath(graph, match, p, i);
            while (v != -1) {
                int pv = p[v];
                int ppv = match[pv];
                match[v] = pv;
                match[pv] = v;
                v = ppv;
            }
        }
    }

    int matches = 0;
    for (int i = 0; i < n; ++i) {
        if (match[i] != -1) {
            ++matches;
        }
    }
    return matches / 2;
}

```

## Minimum Cut

```

// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
//
// Running time:
//  $O(|V|^3)$ 
//
// INPUT:
// - graph, constructed using AddEdge()
//
// OUTPUT:
// - (min cut value, nodes in half of min cut)

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

```

```

typedef vector<int> VI;
typedef vector<VI> VVI;

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights) {
    int N = weights.size();
    VI used(N), cut, best_cut;
    int best_weight = -1;

    for (int phase = N-1; phase >= 0; phase--) {
        VI w = weights[0];
        VI added = used;
        int prev, last = 0;
        for (int i = 0; i < phase; i++) {
            prev = last;
            last = -1;
            for (int j = 1; j < N; j++)
                if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
            if (i == phase-1) {
                for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
                for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
                used[last] = true;
                cut.push_back(last);
                if (best_weight == -1 || w[last] < best_weight) {
                    best_cut = cut;
                    best_weight = w[last];
                }
            } else {
                for (int j = 0; j < N; j++)
                    w[j] += weights[last][j];
                added[last] = true;
            }
        }
    }
    return make_pair(best_weight, best_cut);
}

```

## String Processing

### Knuth-Morris-Pratt

```

/** Knuth-Morris-Pratt */
/* Complexity:  $O(N)$  */
void buildFailTable(char *pattern, int *t){
    int i = 0, j = -1, m = strlen(pattern);
    t[0] = -1;
    while (i < m){
        while (j >= 0 && pattern[i] != pattern[j]) j = t[j];
        i++; j++;
        t[i] = j;
    }
}

```

```

    }
}

vector<int> kmpSearch(char *pattern, char *text){
    vector<int> res;
    int i = 0, j = 0, n = strlen(text), m = strlen(pattern);
    int t[m+5];
    buildFailTable(pattern,t);
    while (i < n){
        while (j >= 0 && text[i] != pattern[j]) j = t[j];
        i++; j++;
        if (j == m){
            res.push_back(i-j);
            j = t[j];
        }
    }
    return res;
}

```

## Z-Algorithm

```

/* Z-Algorithm */
// Z[i] is the longest substring starting from i which is also a prefix of
// s
// Z[0] is not set
int L = 0, R = 0;
for (int i = 1; i < n; ++i) {
    if (i > R) {
        L = R = i;
        while (R < n && s[R] == s[R-L]) ++R;
        Z[i] = R-L; --R;
    }
    else {
        int k = i-L;
        if (Z[k] < R-i+1) Z[i] = Z[k];
        else {
            L = i;
            while (R < n && s[R] == s[R-L]) ++R;
            Z[i] = R-L; --R;
        }
    }
}
}

```

## Suffix Array

```

/** Suffix Array */
/* complexity: O(N log N) */

#define MAXN 200000

char T[MAXN+5]; // input
int n; // length
int RA[MAXN+5], tempRA[MAXN+5]; // rank array
int SA[MAXN+5], tempSA[MAXN+5]; // suffix array

```

```

int c[MAXN+5]; //for counting/radix sort

void countingSort(int k) {
    int sum, maxi = max(300,n);
    memset(c,0,sizeof(c));
    for (int i = 0; i < n; i++)
        c[i+k < n ? RA[i+k] : 0]++;
    for (int i = sum = 0; i < maxi; i++) {
        int t = c[i]; c[i] = sum;
        sum += t;
    }
    for (int i = 0; i < n; i++)
        tempSA[c[SA[i]+k<n?RA[SA[i]+k]:0]++] = SA[i];
    for (int i = 0; i < n; i++) SA[i] = tempSA[i];
}

void SuffixArray_Construct() {
    int r;
    for (int i = 0; i < n; i++) RA[i] = T[i]-'.';
    for (int i = 0; i < n; i++) SA[i] = i;
    for (int k = 1; k < n; k <= 1) {
        countingSort(k);
        countingSort(0);
        tempRA[SA[0]] = r = 0;
        for (int i = 1; i < n; i++)
            tempRA[SA[i]] =
                (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
        for (int i = 0; i < n; i++) RA[i] = tempRA[i];
    }
}

```

## Suffix Tree

```

/** Suffix Tree Ukkonen's algorithm */
/* Complexity: O(N) (Warning: large multiplier) */

const string ALPHABET = "abcdefghijklmnopqrstuvwxyz0123456789\1\2";
const int NALPHABET = 38;

struct Node {
    int begin, end, depth;
    Node* parent;
    Node** children;
    Node* suffixLink;

    Node(int begin, int end, int depth, Node* parent) {
        this->begin = begin;
        this->end = end;
        this->depth = depth;
        this->parent = parent;
        this->children = new Node*[NALPHABET];
        for (int i = 0; i < NALPHABET; ++i) {
            this->children[i] = NULL;
        }
    }
}

```



```

    }
}

~Node() {
    delete[] children;
}

};

Node* buildSuffixTree(string s) {
    int n = s.length();
    char* a = new char[n];
    for (int i = 0; i < n; ++i) {
        a[i] = (char) ALPHABET.find(s[i]);
    }

    Node* root = new Node(0, 0, 0, NULL);
    Node* node = root;
    for (int i = 0, tail = 0; i < n; ++i, ++tail) {
        Node* last = NULL;
        while (tail >= 0) {
            Node* ch = node->children[a[i - tail]];
            while (ch != NULL && tail >= ch->end - ch->begin) {
                tail -= (ch->end - ch->begin);
                node = ch;
                ch = ch->children[a[i - tail]];
            }

            if (ch == NULL) {
                node->children[a[i]] = new Node(i, n, node->depth + node->end -
node->begin, node);
                if (last != NULL) {
                    last->suffixLink = node;
                }
                last = NULL;
            } else {
                char t = a[ch->begin + tail];
                if (t == a[i]) {
                    if (last != NULL) {
                        last->suffixLink = node;
                    }
                    break;
                } else {
                    Node* splitNode = new Node(ch->begin, ch->begin + tail, node-
>depth + node->end - node->begin, node);
                    splitNode->children[a[i]] = new Node(i, n, ch->depth + tail,
splitNode);
                    splitNode->children[t] = ch;
                    ch->begin += tail;
                    ch->depth += tail;
                    ch->parent = splitNode;
                    node->children[a[i - tail]] = splitNode;
                    if (last != NULL) {
                        last->suffixLink = splitNode;
                    }
                }
            }
        }
    }
}

```

```

    }
    last = splitNode;
}

}

if (node == root) {
    --tail;
} else {
    node = node->suffixLink;
}

}

delete[] a;

return root;
}

/* Example: longest common substring */
int lcsLength;
Node* lcsNode;

int traverseLCS(Node* node, const vector<int>& stops, const int target) {
    for (int i = 0; i < stops.size(); ++i) {
        if (node->begin <= stops[i] && stops[i] < node->end) {
            return 1 << i;
        }
    }

    int mask = 0;
    for (int f = 0; f < ALPHABET.length(); ++f) {
        if (node->children[f] != NULL) {
            mask |= traverseLCS(node->children[f], stops, target);
        }
    }

    if (mask == target) {
        int curLength = node->depth + node->end - node->begin;
        if (lcsLength < curLength) {
            lcsLength = curLength;
            lcsNode = node;
        }
    }

    return mask;
}

int longestCommonSubstring(const vector<string> &ss) {
    int totalN = 0;
    int n = ss.size();
    for (int i = 0; i < n; ++i) {
        totalN += ss[i].length() + 1;
    }
}

```

```

string s;
s.resize(totalN);
int offset = 0;
vector<int> stops;
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < ss[i].length(); ++j) {
        s[offset + j] = ss[i][j];
    }
    offset += ss[i].length() + 1;
    s[offset - 1] = '0' + i;
    stops.push_back(offset - 1);
}

Node* tree = buildSuffixTree(s);
lcsLength = 0;
lcsNode = NULL;
traverseLCS(tree, stops, (1 << n) - 1);
delete tree;
return lcsLength;
}

```

### Aho-Corasick

```

/** Aho-Corasick Dictionary Matching */
const int NALPHABET = 26;

struct Node {
    Node** children, go;
    bool leaf;
    char charToParent;
    Node* parent, suffLink, dictSuffLink;
    int count, value;

    Node() {
        children = new Node*[NALPHABET];
        go = new Node*[NALPHABET];
        for (int i = 0; i < NALPHABET; ++i) {
            children[i] = go[i] = NULL;
        }
        parent = suffLink = dictSuffLink = NULL;
        leaf = false;
        count = 0;
    }
};

Node* createRoot() {
    Node* node = new Node();
    node->suffLink = node;
    return node;
}

void addString(Node* node, const string& s, int value = -1) {
    for (int i = 0; i < s.length(); ++i) {

```

```

        int c = s[i] - 'a';
        if (node->children[c] == NULL) {
            Node* n = new Node();
            n->parent = node;
            n->charToParent = s[i];
            node->children[c] = n;
        }
        node = node->children[c];
    }
    node->leaf = true;
    node->count++;
    node->value = value;
}

Node* suffLink(Node* node);
Node* dictSuffLink(Node* node);
Node* go(Node* node, char ch);
int calc(Node* node);

Node* suffLink(Node* node) {
    if (node->suffLink == NULL) {
        if (node->parent->parent == NULL) {
            node->suffLink = node->parent;
        } else {
            node->suffLink = go(suffLink(node->parent), node->charToParent);
        }
    }
    return node->suffLink;
}

Node* dictSuffLink(Node* node) {
    if (node->dictSuffLink == NULL) {
        Node* n = suffLink(node);
        if (node == n) {
            node->dictSuffLink = node;
        } else {
            while (!n->leaf && n->parent != NULL) {
                n = dictSuffLink(n);
            }
            node->dictSuffLink = n;
        }
    }
    return node->dictSuffLink;
}

Node* go(Node* node, char ch) {
    int c = ch - 'a';
    if (node->go[c] == NULL) {
        if (node->children[c] != NULL) {
            node->go[c] = node->children[c];
        } else {
            node->go[c] = node->parent == NULL ? node : go(suffLink(node), ch);
        }
    }
}

```

```

    }
    return node->go[c];
}

int calc(Node* node) {
    if (node->parent == NULL) {
        return 0;
    } else {
        return node->count + calc(dictSuffLink(node));
    }
}

int main() {
    Node* root = createRoot();
    addString(root, "a", 0);
    addString(root, "aa", 1);
    addString(root, "abc", 2);

    string s("abcaadc");
    Node* node = root;
    for (int i = 0; i < s.length(); ++i) {
        node = go(node, s[i]);
        Node* temp = node;
        while (temp != root) {
            if (temp->leaf) {
                printf("string (%d) occurs at position %d\n", temp->value, i);
            }
            temp = dictSuffLink(temp);
        }
    }

    return 0;
}

```

## Mathematics

### Extended Euclid

```

/** Extended Euclid | returns <x,y> where ax + by = gcd(a,b) */
/* complexity: O(min(log(a),log(b))) */
pair<ll,ll> extendedEuclid(ll a, ll b){
    ll x = 0, y = 1, lastx = 1, lasty = 0;
    while (b != 0){
        ll quotient = a / b;
        /* (a, b) = (b, a mod b) */
        ll temp = a;
        a = b;
        b = temp % b;
        /* (x, lastx) = (lastx - quotient*x, x) */
        temp = x;
        x = lastx - quotient * x;
        lastx = temp;
    }
}

```

```

/* (y, lasty) = (lasty - quotient*y, y) */
temp = y;
y = lasty - quotient * y;
lasty = temp;
}
return make_pair(lastx, lasty);
}

```

### Diophantine

```

// computes x and y such that ax + by = c; on failure, x = y = -1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
    int d = gcd(a,b);
    if (c%d) {
        x = y = -1;
    } else {
        x = c/d * mod_inverse(a/d, b/d);
        y = (c-a*x)/b;
    }
}

```

### Chinese Remainder Theorem

```

// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).
// Return (z,M). On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
    int s, t;
    int d = extended_euclid(x, y, s, t);
    if (a%d != b%d) return make_pair(0, -1);
    return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i. Note that the solution is
// unique modulo M = lcm i (x[i]). Return (z,M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &x, const VI &a) {
    PII ret = make_pair(a[0], x[0]);
    for (int i = 1; i < x.size(); i++) {
        ret = chinese_remainder_theorem(ret.second, ret.first, x[i], a[i]);
        if (ret.second == -1) break;
    }
    return ret;
}

```

### Rabin Miller Primality Test

```

/** Works for all 64-bit integers */
bool rabinMillerPrimalityTest(long long n) {
    if ((n & 1) == 0) return n == 2;
    if (n == 1) return false;

    long long a[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
}

```

```

int s = 0;
long long d = n - 1;
while ((d & 1) == 0) {
    d /= 2LL;
    ++s;
}
for (int i = 0; i < 12; ++i) {
    if (a[i] >= n) break;
    long long ad = powerMod(a[i], d, n);
    if (ad != 1) {
        bool composite = true;
        for (int j = 0; j < s; ++j) {
            if (ad == n - 1) {
                composite = false;
                break;
            }
            ad = mulmod(ad, ad, n);
        }
        if (composite) return false;
    }
}
return true;
}

```

### Pollard-Rho Factorization

```

ll mulmod(ll a, ll b, ll m) {
    ll r = 0;
    a %= m;
    b %= m;
    while (b > 0) {
        if (b & 1) {
            r = ((m-r) > a) ? r+a : r+a-m;
        }
        b >>= 1;
        if (b) {
            a = ((m-a) > a) ? a+a : a+a-m;
        }
    }
    return r;
}

ll powmod(ll a, ll b, ll mod) {
    if (b == 0) {
        return 1;
    } else if (b & 1) {
        return mulmod(a, powmod(a, b - 1, mod), mod);
    } else {
        return powmod(mulmod(a, a, mod), b >> 1LL, mod);
    }
}

ll gcd(ll a, ll b) {

```

```

if (a == 0) {
    return b;
} else {
    return gcd(b % a, a);
}
}

ll g(ll x, ll n, ll addum = 1) {
    return (mulmod(x, x, n) + addum + n) % n;
}

/* On primality testing, rely on rabin miller first */
ll pollardRho(ll n, ll start_x = 2, ll addum = 1) {
    ll x = start_x, y = start_x, d = 1;
    while (d == 1) {
        x = g(x, n, addum);
        y = g(g(y, n, addum), n, addum);
        d = gcd(abs(x - y), n);
    }
    if (d != 1) {
        return d;
    } else {
        return pollardRho(n, rand(), rand() % 2 ? 1 : -1);
    }
}
}

```

### Lagrange Interpolation

```

/** Lagrange Polynomial Interpolation */
/* complexity: O(n^2) */
class lagrangeInterpolation {
public:
    lagrangeInterpolation () : x_(0), y_(0) {}

    void addCoef (db x, db y){
        x_.push_back(x);
        y_.push_back(y);
    }

    db interpolate (db x){
        db value = 0;
        for (int i = 0; i < (int)x_.size(); ++i){
            db addum = y_[i];
            for (int j = 0; j < (int)x_.size(); ++j) if (i != j){
                addum *= (x - x_[j]);
                addum /= (x_[i] - x_[j]);
            }
            value += addum;
        }
        return value;
    }
}

vector<db> x_, y_;

```

```

};

class modularInterpolation {
public:
    modularInterpolation (const ll &modu) : modu_(modu), x_(0), y_(0) {}

    void addCoef (ll x, ll y){
        x %= modu_;
        if (x < 0LL) x += modu_;
        x_.push_back(x);

        y %= modu_;
        if (y < 0LL) y += modu_;
        y_.push_back(y);
    }

    ll interpolate (ll x){
        x %= modu_;
        if (x < 0LL) x += modu_;

        for (int i = 0; i < (int)x_.size(); ++i) if (x_[i] == x) return y_[i];

        ll value = 0LL;
        for (int i = 0; i < (int)x_.size(); ++i){
            ll addum = y_[i];
            for (int j = 0; j < (int)x_.size(); ++j) if (j != i){
                ll delta1 = (x - x_[j] + modu_) % modu_;
                ll delta2 = (x_[i] - x_[j] + modu_) % modu_;
                addum = (addum * delta1) % modu_;
                addum = (addum * multInverse(delta2, modu_)) % modu_;
            }
            value += addum;
            value %= modu_;
        }

        return value;
    }
    const ll modu_;
    vector<ll> x_, y_;
};

/* WARNING: no two x_[i] should be the same */

```

### Fast Fourier Transform

```

/** Fast Fourier Transform */
/* complexity: O(N log N) */
vector< complex<db> > iterativeDFT (const vector< complex<db> > &seq, int
direction) {
    int n = SZ(seq);
    int bits = 0;
    int tmp_n = n;
    complex<db> *placeholder = new complex<db>[n];

```

```

    complex<db> *tmp = new complex<db>[n];

    while (tmp_n > 1){
        ++bits;
        tmp_n /= 2;
    }

    REP(i,n){
        int res = 0;
        int tmp_i = i;
        REP(j,bits){
            if (tmp_i % 2) res += (1 << (bits-j-1));
            tmp_i /= 2;
        }
        placeholder[i] = seq[res];
    }

    for (int comp_size = 2; comp_size <= n; comp_size *= 2){
        for (int j = 0; j < n; j += comp_size){
            int n_mem = comp_size / 2;
            db w_mult_exp_i = 2. * acos(-1.) / (db)comp_size;
            if (!direction) w_mult_exp_i *= -1.;
            complex<db> w_mult (cos(w_mult_exp_i), sin(w_mult_exp_i));
            complex<db> w (1., 0.);
            for (int k = 0; k < comp_size; ++k){
                int idx = k % n_mem;
                tmp[k] = placeholder[j+idx] + w * placeholder[j+n_mem+idx];
                w = w * w_mult;
            }
            for (int k = 0; k < comp_size; ++k){
                placeholder[j+k] = tmp[k];
            }
        }
    }

    vector< complex<db> > result;
    for (int i = 0; i < n; ++i) result.PB(placeholder[i]);

    delete[] placeholder;
    delete[] tmp;
    return result;
}

vector<db> FFT(vector<db> a, vector<db> b) {
    if (SZ(a) == 0) a.PB(0.);
    if (SZ(b) == 0) b.PB(0.);
    int n_final_elements = SZ(a) + SZ(b) - 1;

    int actual_size = 1;
    while (actual_size < max(SZ(a), SZ(b))){
        actual_size *= 2;
    }
    actual_size *= 2;

```

```

while (SZ(a) < actual_size) a.PB(0.);
while (SZ(b) < actual_size) b.PB(0.);

vector< complex<db> > dft_input_a, dft_input_b;
REP(i,actual_size) {
    dft_input_a.PB(complex<db> (a[i], 0.));
    dft_input_b.PB(complex<db> (b[i], 0.));
}

dft_input_a = iterativeDFT (dft_input_a, 1);
dft_input_b = iterativeDFT (dft_input_b, 1);
REP(i,actual_size) {
    dft_input_a[i] = dft_input_a[i] * dft_input_b[i];
}
dft_input_a = iterativeDFT (dft_input_a, 0);

vector<db> res;
REP(i,n_final_elements) {
    res.PB(dft_input_a[i].real() / (db) actual_size);
}
return res;
}

```

## Karatsuba

```

typedef vector<long long> vll;

vll karatsubaMultiply(const vll &a, const vll &b) {
    int n = a.size();
    vll res(n + n);
    if (n <= 32) {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                res[i + j] += a[i] * b[j];
        return res;
    }

    int k = n >> 1;
    vll a1(a.begin(), a.begin() + k);
    vll a2(a.begin() + k, a.end());
    vll b1(b.begin(), b.begin() + k);
    vll b2(b.begin() + k, b.end());

    vll a1b1 = karatsubaMultiply(a1, b1);
    vll a2b2 = karatsubaMultiply(a2, b2);

    for (int i = 0; i < k; i++)
        a2[i] += a1[i];
    for (int i = 0; i < k; i++)
        b2[i] += b1[i];

    vll r = karatsubaMultiply(a2, b2);
    for (int i = 0; i < (int) a1b1.size(); i++)

```

```

        r[i] -= a1b1[i];
    for (int i = 0; i < (int) a2b2.size(); i++)
        r[i] -= a2b2[i];

    for (int i = 0; i < (int) r.size(); i++)
        res[i + k] += r[i];
    for (int i = 0; i < (int) a1b1.size(); i++)
        res[i] += a1b1[i];
    for (int i = 0; i < (int) a2b2.size(); i++)
        res[i + n] += a2b2[i];
    return res;
}

```

## Simplex

```

// Two-phase simplex algorithm for solving linear programs of the form
//
// maximize c^T x
// subject to Ax <= b
//             x >= 0
//
// INPUT: A -- an m x n matrix
//         b -- an m-dimensional vector
//         c -- an n-dimensional vector
//         x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments. Then, call Solve(x).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
    int m, n;
    VI B, N;
    VVD D;

    LPSolver(const VVD &A, const VD &b, const VD &c) :

```

```

    m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++) D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) {
        B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];
    }
    for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
    N[n] = -1; D[m+1][n] = 1;
}

void Pivot(int r, int s) {
    for (int i = 0; i < m+2; i++) if (i != r)
        for (int j = 0; j < n+2; j++) if (j != s)
            D[i][j] -= D[r][j] * D[i][s] / D[r][s];
    for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];
    for (int i = 0; i < m+2; i++) if (i != r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
}

bool Simplex(int phase) {
    int x = phase == 1 ? m+1 : m;
    while (true) {
        int s = -1;
        for (int j = 0; j <= n; j++) {
            if (phase == 2 && N[j] == -1) continue;
            if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] <
N[s]) s = j;
        }
        if (D[x][s] >= -EPS) return true;
        int r = -1;
        for (int i = 0; i < m; i++) {
            if (D[i][s] <= 0) continue;
            if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
                D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r = i;
        }
        if (r == -1) return false;
        Pivot(r, s);
    }
}

DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] <= -EPS) {
        Pivot(r, n);
        if (!Simplex(1) || D[m+1][n+1] < -EPS) return -
numeric_limits<DOUBLE>::infinity();
        for (int i = 0; i < m; i++) if (B[i] == -1) {
            int s = -1;
            for (int j = 0; j <= n; j++)
                if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] <
N[s]) s = j;

```

```

        Pivot(i, s);
    }
}

if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
x = VD(n);
for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
return D[m][n+1];
}
};

int main() {
    const int m = 4;
    const int n = 3;
    DOUBLE _A[m][n] = {
        { 6, -1, 0 },
        { -1, -5, 0 },
        { 1, 5, 1 },
        { -1, -5, -1 }
    };
    DOUBLE _b[m] = { 10, -4, 5, -5 };
    DOUBLE _c[n] = { 1, -1, 0 };

    VVD A(m);
    VD b(_b, _b + m);
    VD c(_c, _c + n);
    for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);

    LPSolver solver(A, b, c);
    VD x;
    DOUBLE value = solver.Solve(x);

    cerr << "VALUE: " << value << endl;
    cerr << "SOLUTION:";
    for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
    cerr << endl;
    return 0;
}

```

## Gauss Jordan Elimination

```

// Gauss-Jordan elimination with full pivoting.
//
// Uses:
// (1) solving systems of linear equations (AX=B)
// (2) inverting matrices (AX=I)
// (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT: a[][] = an nxn matrix
//        b[][] = an nxm matrix
//
// OUTPUT: X = an nxm matrix (stored in b[][])
```

```
//      A^{-1} = an nxn matrix (stored in a[][])
//      returns determinant of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
    const int n = a.size();
    const int m = b[0].size();
    VI irow(n), icol(n), ipiv(n);
    T det = 1;

    for (int i = 0; i < n; i++) {
        int pj = -1, pk = -1;
        for (int j = 0; j < n; j++) if (!ipiv[j])
            for (int k = 0; k < n; k++) if (!ipiv[k])
                if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    }
    if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl;
    exit(0); }
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
        c = a[p][pk];
        a[p][pk] = 0;
        for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
        for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
}

for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
}
```

```
return det;
}

int main() {
    const int n = 4;
    const int m = 2;
    double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
    double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
    VVT a(n), b(n);
    for (int i = 0; i < n; i++) {
        a[i] = VT(A[i], A[i] + n);
        b[i] = VT(B[i], B[i] + m);
    }

    double det = GaussJordan(a, b);

    // expected: 60
    cout << "Determinant: " << det << endl;

    // expected: -0.233333 0.166667 0.133333 0.0666667
    //          0.166667 0.166667 0.333333 -0.333333
    //          0.233333 0.833333 -0.133333 -0.0666667
    //          0.05 -0.75 -0.1 0.2
    cout << "Inverse: " << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }

    // expected: 1.63333 1.3
    //          -0.166667 0.5
    //          2.36667 1.7
    //          -1.85 -1.35
    cout << "Solution: " << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            cout << b[i][j] << ' ';
        cout << endl;
    }
}
```

### Reduced Row Echelon Form

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting. This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//
// INPUT: a[][] = an nxm matrix
//
```



```
// OUTPUT:  rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPSILON = 1e-10;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

int rref(VVT &a) {
    int n = a.size();
    int m = a[0].size();
    int r = 0;
    for (int c = 0; c < m && r < n; c++) {
        int j = r;
        for (int i = r+1; i < n; i++)
            if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
        if (fabs(a[j][c]) < EPSILON) continue;
        swap(a[j], a[r]);

        T s = 1.0 / a[r][c];
        for (int j = 0; j < m; j++) a[r][j] *= s;
        for (int i = 0; i < n; i++) if (i != r) {
            T t = a[i][c];
            for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
        }
        r++;
    }
    return r;
}

int main(){
    const int n = 5;
    const int m = 4;
    double A[n][m] = {
        {16,2,3,13},{5,11,10,8},{9,7,6,12},{4,14,15,1},{13,21,21,13} };
    VVT a(n);
    for (int i = 0; i < n; i++)
        a[i] = VT(A[i], A[i] + n);
    int rank = rref (a);
    // expected: 4
    cout << "Rank: " << rank << endl;
    // expected: 1 0 0 1
    //      0 1 0 3
    //      0 0 1 -3
    //      0 0 0 2.78206e-15
    //      0 0 0 3.22398e-15
}
```

```
cout << "rref: " << endl;
for (int i = 0; i < 5; i++){
    for (int j = 0; j < 4; j++){
        cout << a[i][j] << ' ';
    }
    cout << endl;
}
```

## Data Structures

### K-d Tree

```
// -----
// A straightforward, but probably sub-optimal KD-tree implementation that's
// probably good enough for most things (current it's a 2D-tree)
//
// - constructs from n points in O(n lg^2 n) time
// - handles nearest-neighbor query in O(lg n) if points are well
//   distributed
// - worst case for nearest-neighbor may be linear in pathological case
//
// Sonny Chan, Stanford University, April 2009
// -----
--

#include <iostream>
#include <vector>
#include <limits>
#include <cstdlib>

using namespace std;

// number type for coordinates, and its maximum value
typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D
struct point {
    ntype x, y;
    point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
};

bool operator==(const point &a, const point &b){return a.x == b.x && a.y == b.y;}

// sorts points on x-coordinate
bool on_x(const point &a, const point &b){return a.x < b.x;}

// sorts points on y-coordinate
bool on_y(const point &a, const point &b){return a.y < b.y;}

// squared distance between points
```

```

ntype pdist2(const point &a, const point &b) {
    ntype dx = a.x-b.x, dy = a.y-b.y;
    return dx*dx + dy*dy;
}

// bounding box for a set of points
struct bbox {
    ntype x0, x1, y0, y1;
    bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}
    // computes bounding box from a bunch of points
    void compute(const vector<point> &v) {
        for (int i = 0; i < v.size(); ++i) {
            x0 = min(x0, v[i].x);    x1 = max(x1, v[i].x);
            y0 = min(y0, v[i].y);    y1 = max(y1, v[i].y);
        }
    }
}

// squared distance between a point and this bbox, 0 if inside
ntype distance(const point &p) {
    if (p.x < x0) {
        if (p.y < y0) return pdist2(point(x0, y0), p);
        else if (p.y > y1) return pdist2(point(x0, y1), p);
        else return pdist2(point(x0, p.y), p);
    }
    else if (p.x > x1) {
        if (p.y < y0) return pdist2(point(x1, y0), p);
        else if (p.y > y1) return pdist2(point(x1, y1), p);
        else return pdist2(point(x1, p.y), p);
    }
    else {
        if (p.y < y0) return pdist2(point(p.x, y0), p);
        else if (p.y > y1) return pdist2(point(p.x, y1), p);
        else return 0;
    }
}

// stores a single node of the kd-tree, either internal or leaf
struct kdnnode {
    bool leaf; // true if this is a leaf node (has one point)
    point pt; // the single point of this is a leaf
    bbox bound; // bounding box for set of points in children

    kdnnode *first, *second; // two children of this kd-node

    kdnnode() : leaf(false), first(0), second(0) {}
    ~kdnnode() { if (first) delete first; if (second) delete second; }

    // intersect a point with this node (returns squared distance)
    ntype intersect(const point &p) {
        return bound.distance(p);
    }

    // recursively builds a kd-tree from a given cloud of points

```

```

void construct(vector<point> &vp) {
    // compute bounding box for points at this node
    bound.compute(vp);
    // if we're down to one point, then we're a leaf node
    if (vp.size() == 1) {
        leaf = true;
        pt = vp[0];
    }
    else {
        // split on x if the bbox is wider than high (not best heuristic...)
        if (bound.x1-bound.x0 >= bound.y1-bound.y0)
            sort(vp.begin(), vp.end(), on_x);
        // otherwise split on y-coordinate
        else
            sort(vp.begin(), vp.end(), on_y);

        // divide by taking half the array for each child
        // (not best performance if many duplicates in the middle)
        int half = vp.size()/2;
        vector<point> vl(vp.begin(), vp.begin()+half);
        vector<point> vr(vp.begin()+half, vp.end());
        first = new kdnode(); first->construct(vl);
        second = new kdnode(); second->construct(vr);
    }
}

// simple kd-tree class to hold the tree and handle queries
struct kdtree {
    kdnnode *root;

    // constructs a kd-tree from a points (copied here, as it sorts them)
    kdtree(const vector<point> &vp) {
        vector<point> v(vp.begin(), vp.end());
        root = new kdnode();
        root->construct(v);
    }
    ~kdtree() { delete root; }

    // recursive search method returns squared distance to nearest point
    ntype search(kdnnode *node, const point &p)
    {
        if (node->leaf) {
            // commented special case tells a point not to find itself
            // if (p == node->pt) return sentry;
            // else
            return pdist2(p, node->pt);
        }
        ntype bfirst = node->first->intersect(p);
        ntype bsecond = node->second->intersect(p);
        // choose the side with the closest bounding box to search first
        // (note that the other side is also searched if needed)
        if (bfirst < bsecond) {

```

```

        ntype best = search(node->first, p);
        if (bsecond < best)
            best = min(best, search(node->second, p));
        return best;
    }
    else {
        ntype best = search(node->second, p);
        if (bfirst < best)
            best = min(best, search(node->first, p));
        return best;
    }
}
// squared distance to the nearest
ntype nearest(const point &p) {
    return search(root, p);
}
};

int main() {
    // generate some random points for a kd-tree
    vector<point> vp;
    for (int i = 0; i < 100000; ++i) {
        vp.push_back(point(rand()%100000, rand()%100000));
    }
    kdtree tree(vp);

    // query some points
    for (int i = 0; i < 10; ++i) {
        point q(rand()%100000, rand()%100000);
        cout << "Closest squared distance to (" << q.x << ", " << q.y << ")"
              << " is " << tree.nearest(q) << endl;
    }

    return 0;
}

```

## Fenwick Tree

```

/** Fenwick Tree with Range Update */
#define MAXN 100005

int n, bitMul[MAXN], bitAdd[MAXN];

void internalUpdate(int k, int mul, int add) {
    for (int x = k; x <= n; x += (x & -x)) {
        bitMul[x] += mul;
        bitAdd[x] += add;
    }
}

void update(int l, int r, int value) {
    internalUpdate(l, value, -value * (l - 1));
    internalUpdate(r, -value, value * r);
}

```

```

}

int query(int k) {
    int mul = 0, add = 0;
    for (int x = k; x > 0; x -= (x & -x)) {
        mul += bitMul[x];
        add += bitAdd[x];
    }
    return mul * k + add;
}

```

## Splay Tree

```

#include <cstdio>
#include <algorithm>
using namespace std;

const int N_MAX = 130010;
const int oo = 0x3f3f3f3f;
struct Node {
    Node *ch[2], *pre;
    int val, size;
    bool isTurned;
} nodePool[N_MAX], *null, *root;

Node *allocNode(int val) {
    static int freePos = 0;
    Node *x = &nodePool[freePos++];
    x->val = val, x->isTurned = false;
    x->ch[0] = x->ch[1] = x->pre = null;
    x->size = 1;
    return x;
}

inline void update(Node *x) {
    x->size = x->ch[0]->size + x->ch[1]->size + 1;
}

inline void makeTurned(Node *x) {
    if(x == null)
        return;
    swap(x->ch[0], x->ch[1]);
    x->isTurned ^= 1;
}

inline void pushDown(Node *x) {
    if(x->isTurned) {
        makeTurned(x->ch[0]);
        makeTurned(x->ch[1]);
        x->isTurned ^= 1;
    }
}

```

```

inline void rotate(Node *x, int c) {
    Node *y = x->pre;
    x->pre = y->pre;
    if(y->pre != null)
        y->pre->ch[y == y->pre->ch[1]] = x;
    y->ch[!c] = x->ch[c];
    if(x->ch[c] != null)
        x->ch[c]->pre = y;
    x->ch[c] = y, y->pre = x;
    update(y);
    if(y == root)
        root = x;
}

void splay(Node *x, Node *p) {
    while(x->pre != p) {
        if(x->pre->pre == p)
            rotate(x, x == x->pre->ch[0]);
        else {
            Node *y = x->pre, *z = y->pre;
            if(y == z->ch[0]) {
                if(x == y->ch[0])
                    rotate(y, 1), rotate(x, 1);
                else
                    rotate(x, 0), rotate(x, 1);
            } else {
                if(x == y->ch[1])
                    rotate(y, 0), rotate(x, 0);
                else
                    rotate(x, 1), rotate(x, 0);
            }
        }
    }
    update(x);
}

void select(int k, Node *fa) {
    Node *now = root;
    while(1) {
        pushDown(now);
        int tmp = now->ch[0]->size + 1;
        if(tmp == k)
            break;
        else if(tmp < k)
            now = now->ch[1], k -= tmp;
        else
            now = now->ch[0];
    }
    splay(now, fa);
}

Node *makeTree(Node *p, int l, int r) {
    if(l > r)

```

```

        return null;
    int mid = (l + r) / 2;
    Node *x = allocNode(mid);
    x->pre = p;
    x->ch[0] = makeTree(x, l, mid - 1);
    x->ch[1] = makeTree(x, mid + 1, r);
    update(x);
    return x;
}

int main() {
    int n, m;
    null = allocNode(0);
    null->size = 0;
    root = allocNode(0);
    root->ch[1] = allocNode(oo);
    root->ch[1]->pre = root;
    update(root);

    scanf("%d%d", &n, &m);
    root->ch[1]->ch[0] = makeTree(root->ch[1], 1, n);
    splay(root->ch[1]->ch[0], null);

    while(m --) {
        int a, b;
        scanf("%d%d", &a, &b);
        a ++, b ++;
        select(a - 1, null);
        select(b + 1, root);
        makeTurned(root->ch[1]->ch[0]);
    }

    for(int i = 1; i <= n; i ++) {
        select(i + 1, null);
        printf("%d ", root->val);
    }
}

```

### DP Convex Hull Optimization

```

public class ConvexHullOptimization {

    long[] A = new long[1000000];
    long[] B = new long[1000000];
    int len;
    int ptr;

    // a descends
    public void addLine(long a, long b) {
        // intersection of (A[len-2],B[len-2]) with (A[len-1],B[len-1]) must
        // lie to the left of intersection of (A[len-1],B[len-1]) with (a,b)
        while (len >= 2 && (B[len - 2] - B[len - 1]) * (a - A[len - 1]) >=
            (B[len - 1] - b) * (A[len - 1] - A[len - 2])) {

```

```

        --len;
    }
    A[len] = a;
    B[len] = b;
    ++len;
}

// x ascends
public long minValue(long x) {
    ptr = Math.min(ptr, len - 1);
    while (ptr + 1 < len && A[ptr + 1] * x + B[ptr + 1] <= A[ptr] * x + B[ptr]) {
        ++ptr;
    }
    return A[ptr] * x + B[ptr];
}

// Usage example
public static void main(String[] args) {
    ConvexHullOptimization h = new ConvexHullOptimization();
    h.addLine(3, 0);
    h.addLine(2, 1);
    h.addLine(3, 2);
    h.addLine(0, 6);
    System.out.println(h.minValue(0));
    System.out.println(h.minValue(1));
    System.out.println(h.minValue(2));
    System.out.println(h.minValue(3));
}
}

```

## Geometry

### Point, Segment, Line, Circle

```

double _acos(double x) {
    double ret = acos(x);
    if (ret == ret) return ret;
    if (x < 0) return acos(-1.0);
    return acos(1.0);
}

#define acos  acos
#define sqr(x) ((x)*(x))

const double PI = acos(-1);
const double EPS = 1e-9;
const double INF = 1e300;

struct point{
    double x, y;
    point() { x = y = 0; }
    point(double x, double y) : x(x), y(y) {}
}

```

```

};

struct segment {
    point p1, p2;
    segment() {p1 = p2 = point(0,0);}
    segment(point p1, point p2) : p1(p1), p2(p2) {}
};

/** basic operators and functions of point and segment */
/* complexity: constant */
double cross(const point &p1, const point &p2) {
    /* returns z-component of cross product of two points (vectors) */
    return p1.x * p2.y - p1.y * p2.x;
}

double dot(const point &p1, const point &p2) {
    /* returns dot product of two points (vectors) */
    return p1.x * p2.x + p1.y * p2.y;
}

double getAngle(const point &p1, const point &p2) {
    /* returns angle formed by two vectors. WARNING: undirected angle */
    return fabs(acos(dot(p1,p2) / dist(p1,point(0,0)) / dist(p2,point(0,0))));
}

double getAngle(const point &p1, const point &center, const point &p2) {
    /* returns angle formed by three points. WARNING: undirected angle */
    return getAngle(p1 - center, p2 - center);
}

double distToSegment(const point &p, const segment &s) {
    /* returns distance of a point to a segment */
    if (getAngle(s.p2, s.p1, p) > PI/2 + EPS || getAngle(s.p1, s.p2, p) > PI/2 + EPS) return min(dist(p,s.p1), dist(p,s.p2));
    return fabs(cross(s.p1 - p, s.p2 - p)) / dist(s.p1, s.p2);
}

double distToLine(const point &p, const segment &s){
    /* returns distance of a point to a line (its orthogonal projection) */
    return fabs(cross(s.p1 - p, s.p2 - p)) / dist(s.p1, s.p2);
}

point rotate(const point &p, const double &alpha) {
    /* rotates a point with respect to the origin. alpha in radians */
    return point(p.x * cos(alpha) - p.y * sin(alpha), p.x * sin(alpha) + p.y * cos(alpha));
}

point rotate(const point &p, const point &center, const double &alpha){
    /* rotates a point with respect to point center. alpha in radians */
    return center + rotate(p - center, alpha);
}

point rescale(const point &p, const double s) {
    return point(p.x * s, p.y * s);
}

point dilate(const point &p, const double Factor){
    return rescale(p, Factor);
}

point dilate(const point &p, const point &center, double factor){
    return dilate(p - center, factor) + center;
}

```

```

}
bool isRightTurn(const point &p1, const point &p2, const point &p3){
    return cross(p2 - p1, p3 - p2) <= 0;
    /* straight returns true */
}
bool isOnSameSide(const point &p1, const point &p2, const segment &s){
    double z1 = cross(s.p2 - s.p1, p1 - s.p1);
    double z2 = cross(s.p2 - s.p1, p2 - s.p1);
    return (z1 + EPS < 0 && z2 + EPS < 0) || (0 < z1 - EPS && 0 < z2 - EPS)
    || fabs(z1) < EPS || fabs(z2) < EPS;
    /* on segment returns true */
}
bool isOnLine(const point &p, const segment &l){
    return fabs((l.p1.y - p.y) * (l.p2.x - p.x) - (l.p2.y - p.y) * (l.p1.x - p.x)) < EPS;
}
bool isOnSegment(const point &p, const segment &s){
    return fabs(dist(p, s.p1) + dist(p, s.p2) - dist(s.p1, s.p2)) < EPS;
}
bool isIntersecting(const segment &s1, const segment &s2){
    return !(isOnSameSide(s1.p1,s1.p2,s2) || isOnSameSide(s2.p1,s2.p2,s1)) ||
    isOnSegment(s1.p1,s2) || isOnSegment(s1.p2,s2) || isOnSegment(s2.p1,s1) ||
    isOnSegment(s2.p2,s1);
}
bool isParallel(const segment &s1, const segment &s2){
    return fabs((s1.p1.y-s1.p2.y)*(s2.p1.x-s2.p2.x)-(s2.p1.y-s2.p2.y)*(s1.p1.x-s1.p2.x)) < EPS;
}
point intersection(const segment &s1, const segment &s2){
    /* assumes !isParallel(s1,s2) */
    double x1 = s1.p1.x - s1.p2.x;
    double x2 = s2.p1.x - s2.p2.x;
    double y1 = s1.p1.y - s1.p2.y;
    double y2 = s2.p1.y - s2.p2.y;
    double cross1 = cross(s1.p1, s1.p2);
    double cross2 = cross(s2.p1, s2.p2);
    return point ((cross1 * x2 - cross2 * x1) / (x1 * y2 - x2 * y1), (cross1 * y2 - cross2 * y1) / (x1 * y2 - x2 * y1));
}
point projection(const point &p, const segment &s){
    /* projects p onto line s */
    return rescale(s.p2 - s.p1, dot(p - s.p1, s.p2 - s.p1) / sqr(length(s)))
    + s.p1;
}

/** introducing circle */
struct circle {
    point center;
    double r;
    circle() { center = point(0, 0); r = 0; }
    circle(point p, double r) : center(p), r(r) {}
};
vector<point> intersectionLineCircle(const segment &l, const circle &c){

```

```

    vector<point> res;
    double dx = l.p2.x - l.p1.x;
    double dy = l.p2.y - l.p1.y;
    double dr = length(l);
    double d = cross(l.p1 - c.center, l.p2 - c.center);
    if (sqr(c.r) * sqr(dr) - sqr(d) + EPS < 0) return res;
    double det = sqrt(fabs(sqr(c.r) * sqr(dr) - sqr(d)));
    double sdx = dy < 0 ? -dx : dx;
    double sdy = fabs(dy);
    res.push_back(c.center + point((d*dy + sdx * det)/sqr(dr), (-d*dx + sdy * det)/sqr(dr)));
    if (det > EPS) res.push_back(c.center + point((d*dy - sdx * det)/sqr(dr), (-d*dx - sdy * det)/sqr(dr)));
    return res;
}
vector<point> intersectionSegmentCircle(const segment &s, const circle &c){
    vector<point> res, _res = intersectionLineCircle(s,c);
    for (vector<point>::iterator it = _res.begin(); it != _res.end(); ++it){
        if (isOnSegment(*it,s)) res.push_back(*it);
    }
    return res;
}

```

### Polygons (Area, Orientation)

```

/** introducing polygon */
typedef vector<point> polygon;

/** Check position of a point with respect to a polygon */
/* complexity : O(N) */
bool isPointInsidePolygon(point p, polygon poly){
    /* ray casting to the right */
    segment ray (p,p+point(1,0));
    int n = (int)poly.size();
    /* counts the number of intersections */
    int nIntersection = 0;
    for (int i = 0; i < n; ++i){
        segment side(poly[i],poly[(i+1)%n]);
        if (isOnSegment(p,side)) return false;
        if (isParallel(ray,side)) continue;
        point x = intersection(ray,side);
        if (isOnSegment(x,side) && dot(x-p,ray.p2-p) > 0){
            /* special case: x is one of vertices of sides */
            if (x == side.p1){
                if (isRightTurn(p,x,side.p2)) nIntersection ++;
            }
            else if (x == side.p2){
                if (isRightTurn(p,x,side.p1)) nIntersection ++;
            }
            else nIntersection ++;
        }
    }
    return nIntersection % 2 == 1;
}

```

```
}

```

## Convex Hull

```
/** Convex Hull | monotone chain algorithm */
/* complexity : O(N log N) */
polygon convexHull(polygon p){
    int m = 0, n = p.size();
    polygon hull(2*n);
    sort(p.begin(), p.end());
    for (int i = 0; i < n; ++i){
        while (m >= 2 && isRightTurn(hull[m-2], hull[m-1], p[i])) --m;
        hull[m++] = p[i];
    }
    for (int i = n-1, t = m+1; i >= 0; --i){
        while (m >= t && isRightTurn(hull[m-2], hull[m-1], p[i])) --m;
        hull[m++] = p[i];
    }
    hull.resize(m);
    return hull;
}
```

## Delaunay Triangulation

```
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT: x[] = x-coordinates
//        y[] = y-coordinates
//
// OUTPUT: triples = a vector containing m triples of indices
//           corresponding to triangle vertices

#include <vector>
using namespace std;

typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
    int n = x.size();
    vector<T> z(n);
    vector<triple> ret;

    for (int i = 0; i < n; i++)
        z[i] = x[i] * x[i] + y[i] * y[i];
}
```

```
for (int i = 0; i < n-2; i++) {
    for (int j = i+1; j < n; j++) {
        for (int k = i+1; k < n; k++) {
            if (j == k) continue;
            double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
            double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
            double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
            bool flag = zn < 0;
            for (int m = 0; flag && m < n; m++)
                flag = flag && ((x[m]-x[i])*xn +
                               (y[m]-y[i])*yn +
                               (z[m]-z[i])*zn <= 0);
            if (flag) ret.push_back(triple(i, j, k));
        }
    }
    return ret;
}

int main() {
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);

    //expected: 0 1 3
    //           0 3 2

    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}
```

## Miscellaneous

### Graph Theorems

**Erdos-Gallai.** A sequence of nonnegative integers  $d_1 \geq \dots \geq d_n$  is a sequence of degree of an undirected graph iff  $\sum d_i$  is even and  $\sum_{i=1}^k d_i \leq k(k-1) + \sum_{i=k+1}^n \min(d_i, k)$

**Fulkerson-Chen-Anstee.** A sequence  $((a_1, b_1), \dots, (a_n, b_n))$  of nonnegative integer pairs with  $a_1 \geq \dots \geq a_n$  is a sequence of (in, outdeg) of a directed graph iff  $\sum a_i = \sum b_i$  and  $\sum_{i=1}^k a_i \leq \sum_{i=1}^k \min(b_i, k-1) + \sum_{i=k+1}^n \min(b_i, k)$

**Lindstrom-Gessel-Viennot.** The number of non-intersecting path from  $A$  to  $B$  in a directed acyclic graph is equal to the determinant of ... (elements  $(i, j)$  of matrix denotes the number of ways to go from  $A_i$  to  $B_j$ ).

**Koenig's.** In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.

**Brook's.** For any connected undirected graph  $G$  with maximum degree  $\Delta$ , the chromatic number of  $G$  is at most  $\Delta$  unless  $G$  is a complete graph or an odd cycle, in which case the chromatic number is  $\Delta + 1$ .

## Combinatorics

**Lucas Theorem.**  $\binom{n}{m} = \prod_{i=0}^k \binom{n_i}{m_i} \pmod{p}$  where  $n = \overline{n_k n_{k-1} \dots n_0}$  and  $m = \overline{m_k m_{k-1} \dots m_0}$  in base  $p$ .

**Stirling Number of the First Kind.**  $s(n, k)$  denotes the number of  $n$ -permutation with  $k$  cycles.  $s(n+1, k) = ns(n, k) + s(n, k-1)$ .

**Stirling Number of the Second Kind.**  $S(n, k)$  denotes the number of partition a set of  $n$  into  $k$  non-empty subsets.  $S(n+1, k) = kS(n, k) + S(n, k-1)$ .  $S(n, k) = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n$ .

**Gambler's Ruin.** Two players with  $n_1$  and  $n_2$  points each are playing, each turn P1 has probability of winning  $p$  and P2 has probability  $q = 1 - p$ . The probability of P1 losing all his points is  $\left(1 - \left(\frac{p}{q}\right)^{n_2}\right) / \left(1 - \left(\frac{p}{q}\right)^{n_1+n_2}\right)$ .

## Notes

**std::lower\_bound.** Returns an iterator pointing to the first element in the range  $[first, last)$  which **does not compare less than** val.

**std::upper\_bound.** Returns an iterator pointing to the first element in the range  $[first, last)$  which **compares greater** than val.