University of Calgary Team Reference Document

May 22, 2023

Contents

1 General Tips

- For g++, `#include <bits/stdc++.h>` includes all standard headers.
- The constant π is usually built-in as `M_PI`.
- Use `ios_base::sync_with_stdio(false)` if you are using C++ streams. Big speedup in some cases.

2 Geometry

2.1 Basic 2D Geometry

Basic definitions

```
const double EP = 1e-9; // do not use for angles
typedef complex<double> PX;
const PX BAD(1e100,1e100);
```

Cross/dot product, same slope test

$$\left(\vec{a}\times\vec{b}\right)_z = a_x b_y - a_y b_x$$

```
double cp(PX a, PX b) {return (conj(a)*b).imag();}
double dp(PX a, PX b) {return (conj(a)*b).real();}
bool ss(PX a, PX b) {return fabs(cp(a,b)) < EP;}
```

Orientation: −1=CW, 1=CCW, 0=colinear

```
// Can be used to check if a point is on a line (0)
int ccw(PX a, PX b, PX c) {
    double r = cp(b-a, c-a);
    if (fabs(r) < EP) return 0;
    return r > 0 ? 1 : -1;
}
```

Check if x is on line segment from p_1 to p_2

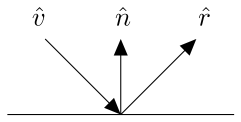
```
bool onSeg(PX p1, PX p2, PX x) {
    // Sometimes 1e-14 may be a better choice here
    return fabs(abs(p2-p1)-abs(x-p1)-abs(x-p2))<EP;
}
```

Angle between vectors (0 to π)

Use `fabs(arg(x/y))`

Reflect vector using line normal (unit vectors)

Orientation of $\hat{n}$ does not matter.
Use `r = n * n / (-v)`



Intersection point of two lines

```
PX lineIntersect(PX p1, PX v1, PX p2, PX v2) {
    // If exact same line, pick random point (p1)
    if (ss(v1, v2)) return ss(v1, p2-p1) ? p1 : BAD;
    return p1 + (cp(p2-p1,v2)/cp(v1,v2))*v1;
}
```

Point to line distance (x to $p + \vec{v}t$)

```
double ptToLine(PX p, PX v, PX x) {
    // Closest point on line: p + v*dp(v, x-p)
    return fabs(cp(v, x-p) / abs(v));
}
```

Line segment (a, b) to point p distance

```
double lsp_dist(PX a, PX b, PX p) {
    return dp(b-a, p-a) > 0 && dp(a-b, p-b) > 0 ?
           abs(cp(b-a, p-a) / abs(b-a)) :
           min(abs(a-p), abs(b-p));
}
```

Check if two line segments intersect (p_1p_2 and q_1q_2)

```
// WARNING UNTESTED - intersection point is tested
bool segIntersect(PX p1, PX p2, PX q1, PX q2) {
    int o1 = ccw(p1, p2, q1);
    int o2 = ccw(p1, p2, q2);
    int o3 = ccw(q1, q2, p1);
    int o4 = ccw(q1, q2, p2);
    if (o1 != o2 && o3 != o4) return true;

    // p1, p2 and q1 are colinear and q1 on p1p2
    if (o1 == 0 && onSeg(p1, p2, q1)) return true;

    // p1, p2 and q1 are colinear and q2 on p1p2
    if (o2 == 0 && onSeg(p1, p2, q2)) return true;

    // q1, q2 and p1 are colinear and p1 on q1q2
    if (o3 == 0 && onSeg(q1, q2, p1)) return true;

     // q1, q2 and p2 are colinear and p2 on q1q2
    if (o4 == 0 && onSeg(q1, q2, p2)) return true;

    return false; // Doesn't fall in above cases
}
```

Intersection point of two line segments (p_1p_2 and q_1q_2)

```
PX segIntersect(PX p1, PX p2, PX q1, PX q2) {
    // Handle special cases for colinear
    if (onSeg(p1, p2, q1)) return q1;
    if (onSeg(p1, p2, q2)) return q2;
    if (onSeg(q1, q2, p1)) return p1;
    if (onSeg(q1, q2, p2)) return p2;
    PX ip = lineIntersect(p1, p2-p1, q1, q2-q1);
    return (onSeg(p1, p2, ip) && onSeg(q1, q2, ip))
        ? ip : BAD;
}
```

Area of polygon (including concave)

```
// points must be in CW or CCW order
double area(vector<PX> const& P) {
    double a = 0.0;
    for (int i = 0; i < P.size(); i++)
        a += cp(P[i], P[(i+1)%P.size()]);
    return 0.5 * fabs(a);
}
```

Centroid of polygon (including concave)

```
// points must be in CW or CCW order
PX centroid(vector<PX> const& P) {
    PX c;
    double scale = 0.0;
    for (int i = 0; i < P.size(); i++) {
        int j = (i+1) % P.size();
        double x = cp(P[i], P[j]);
        c += (P[i] + P[j]) * x;
        scale += 3.0 * x;
    }
    return c / scale;
}
```

Cut convex polygon with straight line

Returns polygon on left side of $p + \vec{v}t$. Points can be in CW or CCW order. Do not use with duplicate points.

```
vector<PX> pts;
void cutPolygon(PX p, PX v) {
    auto P = pts; // make a copy
    pts.clear();
    for (int i = 0; i < P.size(); i++) {
        if (cp(v, P[i]-p) > EP) pts.push_back(P[i]);
        PX pr = P[(i+1)%P.size()],
           ip = lineIntersect(P[i], pr - P[i], p, v);
        if (ip != BAD && onSeg(P[i], pr, ip))
            pts.push_back(ip);
        // remove duplicate points
        while (pts.size() >= 2 && norm(
        pts[pts.size()-1] - pts[pts.size()-2]) < EP)
            pts.pop_back();
    }
}
```

2.2 Convex Hull (Floating Point)

Graham's scan. Complexity: $O(n \log n)$

```
vector<PX> pts;
void convexHull() {
    if (pts.empty()) return;
    int fi = 0;
    for (int i = 1; i < pts.size(); i++)
        if (pts[i].imag() + EP < pts[fi].imag() ||
           (fabs(pts[i].imag() - pts[fi].imag())<EP
           && pts[i].real() + EP < pts[fi].real()))
                fi = i;
    swap(pts[0], pts[fi]);
    sort(++pts.begin(), pts.end(), [](PX a, PX b) {
        PX v1 = a - pts[0], v2 = b - pts[0];
        double a1 = arg(v1), a2 = arg(v2);
        // Use smaller epsilon for angles
        if (fabs(a1 - a2) > 1e-14) return a1 < a2;
        return abs(v1) < abs(v2);
    });
    int M = 2;
    for (int i = 2; i < pts.size(); i++) {
        while (M > 1 && ccw(pts[M-2], pts[M-1],
            pts[i]) <= 0) M--;
        swap(pts[i], pts[M++]);
    }
    if (M < pts.size()) pts.resize(M);
}
```

Notes:

- All intermediate colinear points and duplicate points are discarded
- If all points are colinear, the algorithm will output the two endpoints of the line
- Works with any number of points including 0, 1, 2
- Works with line segments colinear to the starting point

Example usage

```
pts.clear();
pts.emplace_back(0.0, 0.0); // put all the points in
convexHull();
// pts now contains the convex hull in CCW order
// starting from lowest y point
```

Check if point is within polygon

```
// P must be a convex/concave polygon sorted CCW/CW
bool inPolygon(vector<PX> const& P, PX p) {
    double sum = 0.0;
    for (int i = 0; i < P.size(); i++){
        PX a = P[i], b = P[(i+1)%P.size()];
        // to exclude edges, MUST return false
        if (onSeg(a, b, p)) return true;
        sum += arg((a-p) / (b-p));
    }
    // Use 1e-14 for angle
    return fabs(fabs(sum) - 2.0*M_PI) < 1e-14;
}
```

2.3 Convex Hull (Integer)

Returns convex hull in CCW order starting from lowest x point instead of y point. Complexity: $O(n \log n)$
Note: Even for floating point, it might be better to use this line sweep method as it is more numerically stable.

```
typedef long long LL;
typedef pair<LL,LL> PT;
vector<PT> pts;

LL ccw(PT a, PT b, PT c) {
    return (b.first-a.first)*(c.second-a.second) -
           (b.second-a.second)*(c.first-a.first);
}

void convexHull() {
    vector<PT> b, t;
    sort(begin(pts), end(pts));
    for (PT pt : pts) {
        // If you want to keep intermediate colinear
        // points, use < and >
        while (b.size() >= 2 && ccw(b[b.size()-2],
              b[b.size()-1], pt) <= 0) b.pop_back();
        while (t.size() >= 2 && ccw(t[t.size()-2],
              t[t.size()-1], pt) >= 0) t.pop_back();
        b.push_back(pt);
        t.push_back(pt);
    }
    pts = b;
    for (int i = int(t.size()-2); i > 0; i--)
        pts.push_back(t[i]);
}
```

3 Graphs

3.1 Articulation Points and Bridges

Graph does not need to be connected. Tested only on bidirectional (undirected) graphs. Complexity: $O(V+E)$

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

VVI adj;
VI dfs_low, dfs_num;
int cnt;

void dfs(int i, int r, int p) { // (cur, root, parent)
    if (dfs_num[i] != -1) return;
    dfs_low[i] = dfs_num[i] = cnt++;
    int ap = i != r; // number of disconnected
                     // components if vertex is removed
    for (int j : adj[i])
    if (j != p) { // change cond if parallel edges
        if (dfs_num[j] == -1) {
            dfs(j, r, i);
            if (dfs_low[j] >= dfs_num[i]) ap++;
            if (dfs_low[j] > dfs_num[i]) {
                // (i,j) is a bridge
                // each **UNORDERED** pair
                // will occur exactly once
            }
            dfs_low[i] = min(dfs_low[i], dfs_low[j]);
        } else {
            dfs_low[i] = min(dfs_low[i], dfs_num[j]);
        }
    }
    if (ap >= 2) {
        // i is an articulation point
        // each vertex will only occur once
    }
}
```

Example usage:

```
// N is number of vertices
cnt = 0;
adj.assign(N, VI()); // fill adj
dfs_num.assign(N, -1);
dfs_low.resize(N); // initialization not necessary
for (int n = 0; n < N; n++) dfs(n, n, -1);
```

3.2 Bellman-Ford

Consider terminating the loop if no weight was modified in the loop. Complexity: $O(VE)$

```
let weight[V] = all infinity except weight[source] = 0
let parent[V] = all null

loop V-1 times
    for each edge (u,v) with weight w
        if weight[u] + w < weight[v]
            weight[v] = weight[u] + w
            parent[v] = u

// detecting negative weight cycles
for each edge (u,v) with weight w
    if weight[u] + w < weight[v]
    then graph has negative weight cycle
```

3.3 Cycle Detection in Directed Graph

We can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the graph.

3.4 Dijkstra's

Remember to consider using Floyd-Warshall or $O(V^2)$ version of Dijkstra's. Complexity: $O((E+V)\log V)$

```
typedef pair<int,int> PT;
typedef vector<PT> VPT;
vector<VPT> adj;
int dist[N]; // N = number of nodes
priority_queue<PT, vector<PT>, greater<PT>> dja;
dja.emplace(0, START_NODE);
fill(dist, dist+N, INT_MAX);
dist[START_NODE] = 0;
while (!dja.empty()) {
    PT pt = dja.top(); dja.pop();
    if (pt.first != dist[pt.second]) continue;
    for (PT ps : adj[pt.second]) {
        if (pt.first + ps.second < dist[ps.first]) {
            dist[ps.first] = pt.first + ps.second;
            dja.emplace(dist[ps.first], ps.first);
        }
    }
}
```

3.5 Eulerian Cycle

This non-recursive version works with large graphs. Complexity: $O(E\log E)$

```
vector<multiset<int>> adj;
vector<int> cyc;
void euler(int n) {
    stack<int> stk;
    stk.push(n);
    while (!stk.empty()) {
        n = stk.top();
        if (adj[n].empty()) {
            cyc.push_back(n);
            stk.pop();
        } else {
            auto it = adj[n].begin();
            int m = *it;
            adj[n].erase(it);
            adj[m].erase(adj[m].find(n));
            stk.push(m);
        }
    }
}
```

Example usage:

```
adj[0].insert(1); adj[1].insert(0); // 0 to 1
adj[0].insert(1); adj[1].insert(0); // another one
adj[0].insert(0); adj[0].insert(0); // loop on 0
cyc.clear();
euler(0); // find eulerian cycle starting from 0
// cyc contains complete cycle including endpoints
// e.g. 0, 0, 1, 0
```

3.6 Floyd-Warshall

Negative on diagonal means a vertex is in a negative cycle. Complexity: $O(V^3)$

```
let dist[V][V] be initialized to
    dist[v][v] = 0
    dist[u][v] = weight of edge else infinity
for k from 1 to V
    for i from 1 to V
        for j from 1 to V
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] = dist[i][k] + dist[k][j]
```

3.7 Bipartite Graphs

3.7.1 Maximum Bipartite Matching

Complexity: $O(VE)$

```
vector<vector<int>> adj;
vector<char> visL, visR;
vector<int> matchR;
bool bpm(int l) {
    if (visL[l]) return false;
    visL[l] = true;
    for (int r : adj[l]) {
        if (matchR[r] == l) continue;
        visR[r] = true;
        if (matchR[r]<0 || bpm(matchR[r])) {
            matchR[r] = l;
            return true;
        }
    }
    return false;
}
```

Example usage:

```
int ans = 0; // cardinality
vector<int> U; // unmatched left vertices
matchR.assign(R, -1);
for (int l = 0; l < L; l++) {
    visL.assign(L, 0);
    visR.assign(R, 0);
    if (bpm(l)) ans++; else U.push_back(l);
}
```

Minimum vertex cover is $(L - Z) \cup (R \cap Z)$

```
visL.assign(L, 0);
visR.assign(R, 0);
for (int l : U) bpm(l);
for (int l = 0; l < L; l++) {
    if (!visL[l]) then l is in vertex cover
}
for (int r = 0; r < R; r++) {
    if (visR[r]) then r is in vertex cover
}
```

3.7.2 Stable Marriage/Matching

Only tested with equal numbers of men and women.
Complexity: $O(MW)$

```
typedef vector<int> VI;
vector<VI> mPref, wPref;
VI wPartner;
void stableMarriage() {
    int M = mPref.size();
    VI pr(M), fm(M);
    iota(begin(fm), end(fm), 0);
    wPartner.assign(wPref.size(), -1);
    while (!fm.empty()) {
        int m = fm.back();
        int w = mPref[m][pr[m]++];
        if (wPartner[w] == -1 || wPref[w][m]
        < wPref[w][wPartner[w]]) {
            fm.pop_back();
            if (wPartner[w] != -1)
                fm.push_back(wPartner[w]);
            wPartner[w] = m;
        }
    }
}
```

Example usage:

```
mPref.clear(); wPref.clear();

// Man 0 ranks women 2, 0, 1 (best to worst)
mPref.push_back(VI{2,0,1});

// Woman 0 ranks men 1, 2, 0 (best to worst)
wPref.push_back(VI{2,0,1});

stableMarriage(); // matching is in wPartner
```

3.7.3 Notes

- Konig's theorem: In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.
- The complement of a minimum vertex cover is a maximum independent set.
- For minimum/maximum weight bipartite matching, use min cost max flow or Hungarian algorithm.
- For minimum/maximum weight vertex cover, reduce to minimum cut (max flow).

3.8 Max Flow (with Min Cut)

Dinic's algorithm.
Complexity: $O\left(\min\left(V^2E, fE\right)\right)$ where f is the maximum flow
For unit capacities: $O\left(\min\left(V^{2/3}, E^{1/2}\right)E\right)$
For bipartite matching: $O\left(\sqrt{V}E\right)$

```
typedef long long LL;
const int SOURCE=0, SINK=1; // change if necessary

struct edge {
    int to, idx;
    LL cap;
};
vector<vector<edge>> adj;
vector<int> lvl, ptr;

LL totalflow;

LL dfs(int n, LL f) {
    if (n == SINK) { totalflow += f; return f; }
    if (lvl[n] == lvl[SINK]) return 0;
    while (ptr[n] < (int)adj[n].size()) {
        edge& e = adj[n][ptr[n]];
        ptr[n]++;
        if (lvl[e.to] == lvl[n]+1 && e.cap > 0) {
            LL nf = dfs(e.to, min(f, e.cap));
            if (nf) {
                e.cap -= nf;
                adj[e.to][e.idx].cap += nf;
                return nf;
            }
        }
    }
    return 0;
}
bool runMaxFlow() {
    lvl.assign(adj.size(), -1);
    ptr.assign(adj.size(), 0);
    lvl[SOURCE] = 0;
    queue<int> bfs;
    bfs.push(SOURCE);
    while (!bfs.empty()) {
        int t = bfs.front();
        bfs.pop();
        for (edge& e : adj[t]) {
            if (lvl[e.to] != -1 || e.cap <= 0) continue;
            lvl[e.to] = lvl[t]+1;
            bfs.push(e.to);
        }
    }
    if (lvl[SINK] == -1) return false;
    while (dfs(SOURCE, 1LL<<60)) {}
    return true;
}
void initMaxFlow(int nodes) {
    totalflow = 0; adj.clear(); adj.resize(nodes);
}
void addEdge(int a, int b, LL w) {
    adj[a].push_back(edge{b, (int)adj[b].size(), w});
    adj[b].push_back(edge{a, (int)adj[a].size()-1, 0});
}
```

Example usage

```
initMaxFlow(desired number of nodes); // nodes 0 to N-1
addEdge(0, 3, 123); // 0 to 3 with capacity 123
while (runMaxFlow()) {}
// The max flow is now in totalflow
// The min cut: Nodes where lvl[i] == -1 belong to the T
// component, otherwise S
```

3.9 Min Cost Max Flow

Edmonds-Karp with Bellman-Ford algorithm. Complexity: $O\left(\min\left(V^2E^2, fVE\right)\right)$ where f is the maximum flow

```
const int NODES = 101 // maximum number of nodes
typedef long long LL;
typedef pair<int,int> PT;

vector<vector<int>> adj;
LL cap[NODES][NODES], cost[NODES][NODES], flow[NODES][NODES];

LL totalflow, totalcost;
bool runMCMF(int source, int sink) {
    vector<LL> mf(NODES), weight(NODES, 1LL<<60); // must be larger than longest path
    vector<int> parent(NODES, -1);
    weight[source] = 0;
    mf[source] = 1LL<<60; // value must be larger than max flow
    for (int i = 0, lm = 0; i < NODES-1 && lm == i; i++) {
        for (int u = 0; u < NODES; u++) {
            for (int v : adj[u]) {
                if (!cap[u][v] && !flow[v][u]) continue;
                LL w = (flow[v][u]) ? -cost[v][u] : cost[u][v];
                if (weight[u] + w < weight[v]) {
                    weight[v] = weight[u] + w;
                    parent[v] = u;
                    mf[v] = min(mf[u], (flow[v][u]) ? flow[v][u] : cap[u][v]);
                    lm = i+1;
                }
            }
        }
    }
    LL f = mf[sink];
    if (!f) return false;
    for (int j = sink; j != source;) {
        int p = parent[j];
        if (flow[j][p]) {
            cap[j][p] += f;
            flow[j][p] -= f;
        } else {
            cap[p][j] -= f;
            flow[p][j] += f;
        }
        totalcost += f * (weight[j] - weight[p]);
        j = p;
    }
    totalflow += f;
    return true;
}
void initMCMF() {
    totalflow = totalcost = 0;
    adj.clear(); adj.resize(NODES);
    memset(cap, 0, sizeof cap);
    memset(cost, 0, sizeof cost);
    memset(flow, 0, sizeof flow);
}
void addEdge(int a, int b, LL w, LL c) {
    adj[a].push_back(b);
    adj[b].push_back(a); // this line is necessary even without bidirectional edges
    cap[a][b] = w; // set cap[b][a] and cost[b][a] to the same to get bidirectional edges
    cost[a][b] = c;
}
```

Example usage

```
initMCMF();
addEdge(0, 3, 123, 5); // adds edge fron 0 to 3 with capacity 123 and cost 5
while (runMCMF(source, sink)) {}
// The max flow is now in totalflow and total cost in totalcost
```

3.10 Strongly Connected Components

Tarjan's algorithm. Note that Tarjan's algorithm generates SCCs in reverse topological order, while Kosaraju's algorithm generates in topological order. Complexity: $O(V+E)$

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

VVI adj;
VI dfsLow, dfsNum, sccNum, sccStack;
vector<char> vis;
int cnt, numScc;

void dfs(int i) {
    if (dfsNum[i] != -1) return;
    dfsLow[i] = dfsNum[i] = cnt++;
    sccStack.push_back(i);
    vis[i] = true;
    for (int j : adj[i]) {
        dfs(j);
        if (vis[j])
            dfsLow[i] = min(dfsLow[i], dfsLow[j]);
    }
    if (dfsLow[i] == dfsNum[i]) {
        int j;
        do {
            j = sccStack.back();
            sccStack.pop_back();
            vis[j] = false;
            sccNum[j] = numScc;
        } while (i != j);
        numScc++;
    }
}

void scc() {
    int N = adj.size();
    cnt = numScc = 0;
    dfsLow.resize(N); // init not necessary
    dfsNum.assign(N, -1);
    sccNum.resize(N); // init not necessary
    vis.assign(N, false);
    for (int n = 0; n < N; n++) dfs(n);
}
```

3.10.1 2-SAT

```
vector<char> truthValues;

int VAR(int i) {return 2*i;}
int NOT(int i) {return i^1;}
int NVAR(int i) {return NOT(VAR(i));}

void addCond(int c1, int c2) {
    adj[NOT(c1)].push_back(c2);
    adj[NOT(c2)].push_back(c1);
}
void init2SAT(int numVars) {
    adj.assign(2*numVars, VI());
    truthValues.resize(numVars); // init not necessary
}
bool run2SAT() {
    scc();
    for (int i = 0; i < adj.size(); i += 2) {
        if (sccNum[i] == sccNum[i+1]) return false;
        // If SCC is computed with Kosaraju's, use > instead
        truthValues[i/2] = sccNum[i] < sccNum[i+1];
    }
    return true;
}
```

Example usage

```
init2SAT(N); // variables from 0 to N-1
addCond(VAR(4), NVAR(0)); // v4 or not v0
if (run2SAT()) {
    // there is a solution
    // truth values are in truthValues[0 to N-1]
}
```

3.11 Tree Algorithms

3.11.1 Lowest Common Ancestor

LCA can be solved using RMQ of the dfs tree (using sparse table or segment tree), or the following code using this binary-search like method. Complexity: $O(n \log n)$ preprocessing and $O(\log n)$ per query

```
int N,            // number of vertices
    T[100000],    // parent of each vertex (parent of root should be itself)
    L[100000],    // depth of each node from root (calculate with dfs or something)
    P[100000][17]; // P[i][j] is the 2^j parent of i, or root if nonexistent [N][floor(log2(N))+1]

void lcaBuild() {
    for (int n = 0; 1<<n <= N; n++)
        for (int i = 0; i < N; i++)
            P[i][n] = (n) ? P[P[i][n-1]][n-1] : T[i];
}

int lcaQuery(int p, int q) {
    if (L[p] < L[q]) swap(p, q); // ensure p is deeper in tree
    while (L[p] > L[q]) p = P[p][31 - __builtin_clz(L[p] - L[q])]; // get p on same level as q
    if (p == q) return p; // special case if p/q is the LCA
    for (int j = 31 - __builtin_clz(N); j >= 0; j--)
        if (P[p][j] != P[q][j]) {p = P[p][j]; q = P[q][j];}
    return T[p];
}
```

3.11.2 Eccentricity

The maximum distances from every vertex are stored in maxd. Complexity: $O(V)$

```
vector<vector<int>> adj;
int N, edged[N], maxd[N]; // N = number of vertices
int dfs1(int n, int p) {
    edged[n] = 0;
    for (int m : adj[n]) if (m != p)
        edged[n] = max(edged[n], 1+dfs1(m, n));
    return edged[n];
}
void dfs2(int n, int p, int pd) {
    int maxEdged[2] = {pd, 0}, nwmg = 1;
    for (int m : adj[n]) if (m != p) {
        if (edged[m] + 1 > maxEdged[0]) {
            maxEdged[1] = maxEdged[0];
            maxEdged[0] = edged[m] + 1;
            nwmg = 1;
        } else if (edged[m] + 1 == maxEdged[0]) {
            nwmg++;
        } else if (edged[m] + 1 > maxEdged[1]) {
            maxEdged[1] = edged[m] + 1;
        }
    }
    for (int m : adj[n]) if (m != p) {
        int npd = maxEdged[0];
        if (npd == edged[m] + 1 && nwmg == 1)
            npd = maxEdged[1];
        dfs2(m, n, npd+1);
    }
    maxd[n] = max(pd, maxEdged[0]);
}
```

Example usage (goes through each connected component):

```
memset(edged, -1, sizeof edged);
for (int n = 0; n < N; n++) if (edged[n] == -1) {
    dfs1(n, -1);
    dfs2(n, -1, 0);
}
```

3.11.3 Number of times an edge is used in a path between two vertices

Complexity: $O(V)$

```
typedef pair<int,int> PT;
vector<PT> adj[N]; // pairs of (vertex, edge ID)
LL usage[E]; // usage of each edge ID is here
bool visited[N];
int dfs(int n){
    visited[n] = true;
    int nn = 1;
    for (auto pt : adj[n]) {
        if (visited[pt.first]) continue;
        int x = dfs(pt.first);
        usage[pt.second] += LL(N - x) * LL(x);
        nn += x;
    }
    return nn;
}
```

3.11.4 Heavy-Light Decomposition

This code provides a basic framework for HLD.

```
vector<vector<pair<int, int>>> adj;
const int MAXN = 10000;
int nChain     [MAXN], // Chain number of vertex
    nChainIndex[MAXN], // Index in chain of vertex
    parent     [MAXN], // Parent of a vertex
    parentLen  [MAXN], // Length of edge from parent
    tSize      [MAXN]; // Subtree size of a vertex
VI cDepth,  // Depth of a chain (root is 0)
   cLength, // Length of a chain
   cParent; // Vertex number of chain parent

void dfs(int n, int p) {
    tSize[n] = 1;
    for (auto const& m : adj[n]) if (m.first != p) {
        parent[m.first] = n;
        parentLen[m.first] = m.second;
        dfs(m.first, n);
        tSize[n] += tSize[m.first];
    }
}
void hld(int n, int p) {
    nChainIndex[n] = cLength.back()++;
    nChain[n] = (int)cLength.size()-1;

    // Find largest child
    int h = -1;
    for (auto const& m : adj[n])
    if (m.first != p &&
            (h == -1 || tSize[m.first] > tSize[h]))
        h = m.first;

    if (h == -1) return;
    hld(h, n);

    // Process other children
    for (auto const& m : adj[n])
    if (m.first != p && m.first != h) {
        cLength.push_back(0);
        cParent.push_back(n);
        cDepth.push_back(1+cDepth[nChain[n]]);
        hld(m.first, n);
    }
}
```

Example usage:

```
cDepth.assign(1, 0);
cLength.assign(1, 0);
cParent.assign(1, -1);
dfs(0, -1);
hld(0, -1);
```

3.11.5 Notes

- The diameter of a tree (longest distance between two vertices) can be found by choosing any vertex, then finding a furthest vertex v_1, then finding a furthest vertex v_2. The distance between v_1 and v_2 is the diameter, and the centers (1 or 2) are median elements of that path.
- The radius of a tree (longest distance from the best root) is $\left\lfloor \frac{diameter}{2} \right\rfloor$

4 Sequences and Strings

4.1 AVL Tree

Creating your own BST can be useful in certain situations; e.g. to find the kth element in a set in $O(\log n)$.

```
struct node {
    node *l, *r;
    int nodes, height, val;
    node(int v)
    : l(0), r(0), nodes(1), height(1), val(v) {}
} *root;

int height(node *n) {return (n) ? n->height : 0;}
int nodes(node *n) {return (n) ? n->nodes : 0;}
int gb(node *n)
    {return (n) ? height(n->l) - height(n->r) : 0;}

void updHeight(node *n) {
    n->height = max(height(n->l), height(n->r)) + 1;
    n->nodes = nodes(n->l) + nodes(n->r) + 1;
}

void leftRotate(node *&n) {
    node *nr = n->r;
    n->r = nr->l;
    nr->l = n;
    n = nr;
    updHeight(n->l);
    updHeight(n);
}
void rightRotate(node *&n) {
    node *nr = n->l;
    n->l = nr->r;
    nr->r = n;
    n = nr;
    updHeight(n->r);
    updHeight(n);
}

void fix(node *&n) {
    if (!n) return;
    updHeight(n);
    if (gb(n) > 1) {
        if (gb(n->l) < 0) leftRotate(n->l);
        rightRotate(n);
    } else if (gb(n) < -1) {
        if (gb(n->r) > 0) rightRotate(n->r);
        leftRotate(n);
    }
}

void insert(node *&n, int val) {
    if (!n) n = new node(val);
    else if (val < n->val) insert(n->l, val);
    else if (val > n->val) insert(n->r, val);
    fix(n);
}

int predec(node *&n) {
    int ret;
    if (n->r) ret = predec(n->r);
    else {
        node *x = n;
        n = x->l;
        ret = x->val;
        delete x;
    }
    fix(n);
    return ret;
}

void remove(node *&n, int val) {
    if (!n) return;
    if (val < n->val) remove(n->l, val);
    else if (val > n->val) remove(n->r, val);
    else if (n->l) n->val = predec(n->l);
    else {
        node *x = n;
        n = x->r;
        delete x;
    }
    fix(n);
}
```

Example: in-order traversal

```
void inorder(node *n) {
    if (!n) return;
    inorder(n->l);
    cout << n->val << endl;
    inorder(n->r);
}
```

Example: get kth element in set (zero-based)

```
int kth(node *n, int k) {
    if (!n) return 2000000000;
    if (k < nodes(n->l)) return kth(n->l, k);
    else if (k > nodes(n->l))
        return kth(n->r, k - nodes(n->l) - 1);
    return n->val;
}
```

Example: count number of elements strictly less than x

```
int count(node *n, int x) {
    if (!n) return 0;
    if (x <= n->val) return count(n->l, x);
    return 1 + nodes(n->l) + count(n->r, x);
}
```

Example: delete tree

```
void del(node *&n) {
    if (!n) return;
    del(n->l);
    del(n->r);
    delete n;
    n = nullptr;
}
```

4.2 Aho-Corasick and Trie

The Aho-Corasick string matching algorithm locates elements of a finite set of strings (the ”dictionary”) within an input text. Complexity: Linear in length of patterns plus searched text

```
const int AS = 256; // alphabet size
struct node {
    int match = -1, child[AS] = {},
        suffix = 0, dict = 0;
};
vector<node> nodes;
vector<string> dict; // do not add duplicates

void acMatch(string const& s) {
    for (size_t i = 0, n = 0; i < s.size(); i++) {
        char c = s[i];
        while (n && !nodes[n].child[c])
            n = nodes[n].suffix;
        n = nodes[n].child[c];
        for (int m = n; m; m = nodes[m].dict) {
            if (nodes[m].match >= 0) {
                // Replace with whatever you want
                cout << "Matched␣" << nodes[m].match
                     << "␣at␣" << i << endl;
            }
        }
    }
}
```

Example usage:

```
dict = {"bc", "abc"}; // do not add duplicates
acBuild();
acMatch("abcabc");
```

Example output:

```
Matched 1 at 2
Matched 0 at 2
Matched 1 at 5
Matched 0 at 5
```

```
void acBuild() {
    // Build trie
    nodes.assign(1, node()); // create root
    for (size_t i = 0; i < dict.size(); i++) {
        int n = 0;
        for (char c : dict[i]) {
            if (!nodes[n].child[c]) {
                nodes[n].child[c] = nodes.size();
                nodes.emplace_back();
            }
            n = nodes[n].child[c];
        }
        nodes[n].match = i;
    }

    // Build pointers to longest proper suffix and dict
    queue<int> bfs;
    bfs.push(0);
    while (!bfs.empty()) {
        int n = bfs.front();
        bfs.pop();
        for (int i = 0; i < AS; i++)
        if (nodes[n].child[i]) {
            int m = nodes[n].child[i],
                v = nodes[n].suffix;
            while (v && !nodes[v].child[i])
                v = nodes[v].suffix;
            int s = nodes[v].child[i];
            if (s != m) {
                nodes[m].suffix = s;
                nodes[m].dict = (nodes[s].match >= 0)
                    ? s : nodes[s].dict;
            }
            bfs.push(m);
        }
    }
}
```

4.3 KMP and Z-function

Knuth-Morris-Pratt algorithm. Complexity: $O(m+n)$

This function returns a vector containing the zero-based index of the start of each match of K in S. It works with strings, vectors, and pretty much any array-indexed data structure that has a size method. Matches may overlap.

For GNU C++, `strstr()` uses KMP, but `string.find()` in C++ and `String.indexOf()` in Java do not.

Z-function complexity: $O(n)$. z[i] is the length of the longest common prefix between s and the suffix of s starting at i.

```
template<class T>
vector<int> KMP(T const& S, T const& K) {
    vector<int> b(K.size() + 1, -1);
    vector<int> matches;

    // Preprocess
    for (int i = 1; i <= K.size(); i++) {
        int pos = b[i - 1];
        while (pos != -1 && K[pos] != K[i - 1])
            pos = b[pos];
        b[i] = pos + 1;
    }

    // Search
    int sp = 0, kp = 0;
    while (sp < S.size()) {
        while (kp != -1 && (kp == K.size() || K[kp] != S[sp])) kp = b[kp];
        kp++; sp++;
        if (kp == K.size()) matches.push_back(sp - K.size());
    }

    return matches;
}
```

```
vector<int> z;
void calcZ(string const& s) {
    int n = s.size(), l = -1, r = -1;
    z.assign(n, 0);
    for (int i = 1; i < n; i++) {
        if (i <= r) z[i] = min(z[i-l], r-i+1);
        while (i+z[i] < n && s[i+z[i]] == s[z[i]]) z[i]++;
        if (i+z[i]-1 > r) {
            l = i;
            r = i+z[i]-1;
        }
    }
}
```

Example of KMP preprocessing array b[i] and Z-function z[i]:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K[i]	f	i	x	p	f	i	x	f	i	x	p	s	u	f	i	x	\0
b[i]	-1	0	0	0	0	1	2	3	1	2	3	4	0	0	1	2	3
z[i]	0	0	0	0	3	0	0	4	0	0	0	0	0	3	0	0	

4.4 Longest Common Subsequence

Note that if characters are never repeated in at least one string, LCS can be reduced to LIS. Complexity: $O(nm)$

```
template<class T>
int LCS(T const& A, T const& B) {
    int dp[][] = {}; // set size appropriately
    for (int a = 0; a < A.size(); a++) {
        for (int b = 0; b < B.size(); b++) {
            if (a) dp[a][b] = max(dp[a][b], dp[a-1][b]);
            if (b) dp[a][b] = max(dp[a][b], dp[a][b-1]);
            if (A[a] == B[b])
                dp[a][b] = max(dp[a][b], ((a && b) ? dp[a-1][b-1] : 0) + 1);
        }
    }
    return dp[A.size() - 1][B.size() - 1];
}
```

4.5 Longest Increasing Subsequence

Complexity: $O(n \log k)$ where k is the length of the LIS

```
vector<int> L; // L[x] = smallest end of length x LIS
for each x in sequence {
    auto it = lower_bound(L.begin(), L.end(), x);
    if (it == L.end()) L.push_back(x); else *it = x;
}
// Length of LIS is L.size()
```

4.6 Fenwick Tree / Binary Indexed Tree

This implements a D-dimensional Fenwick tree with indexes $[1, N-1]$. Complexity: $O\left(\log^D N\right)$ per operation

```
template<int N, int D=1>
class FenwickTree {
    vector<int> tree;
    int isum(int ps) {return tree[ps];}
    template<class... T>
    int isum(int ps, int n, T... tail) {
        int a = 0;
        while (n) {
            a += isum(ps*N + n, tail...);
            n -= (n & -n);
        }
        return a;
    }
    void iupd(int u, int ps) {tree[ps] += u;}
    template<class... T>
    void iupd(int u, int ps, int n, T... tail) {
        while (n < N) { // TODO: check cond
            iupd(u, ps*N + n, tail...);
            n += (n & -n);
        }
    }
public:
    FenwickTree() : tree(pow(N, D)) {}
    template<class... T, class = class enable_if<sizeof...(T)==D>::type>
    int sum(T... v) {return isum(0, v...);}
    template<class... T, class = class enable_if<sizeof...(T)==D>::type>
    void upd(int u, T... v) {iupd(u, 0, v...);}
};
```

Example usage

```
FenwickTree<130> t; // creates 1D fenwick tree with indexes [1,129]
t.upd(5, 7); // adds 5 to index 7
t.sum(14); // gets sum of all points [1, 14]

FenwickTree<130, 3> t; // creates 3D fenwick tree with indexes [1,129]
t.upd(5, 7, 8, 9); // adds 5 to the point (7, 8, 9)
t.sum(14, 15, 16); // gets sum of all points [(1, 1, 1), (14, 15, 16)]
```

Simple 1D tree (remember, first index is 1)

```
typedef long long LL;
const int N = 100002;
LL f1[N], f2[N];

LL sum(LL *f, int n) {
    LL a = 0;
    while (n) {
        a += f[n];
        n -= (n & -n);
    }
    return a;
}
```

```
void upd(LL *f, int n, LL v) {
    while (n < N) {
        f[n] += v;
        n += (n & -n);
    }
}

// only required for range queries
// with range updates
LL rsum(int n) {
    return sum(f1, n) * n - sum(f2, n);
}
```

To get sum from $[p, q]$:

```
rsum(q) - rsum(p-1)
```

To add v to $[p, q]$:

```
upd(f1, p, v);
upd(f1, q+1, -v);
upd(f2, p, v*(p-1));
upd(f2, q+1, -v*q);
```

4.7 Sparse Table

Solves static range min/max query with $O(n \log n)$ preprocessing and $O(1)$ per query. This code does range minimum query.

```
int N, A[1000000], spt[1000000][20]; // spt[N][floor(log2(N))+1]

void sptBuild() {
    for (int n = 0; 1<<n <= N; n++)
        for (int i = 0; i+(1<<n) <= N; i++)
            spt[i][n] = (n) ? min(spt[i][n-1],
                                  spt[i+(1<<(n-1))][n-1]) : A[i];
}

int sptQuery(int i, int j) {
    int n = 31 - __builtin_clz(j-i+1); // floor(log2(j-i+1))
    return min(spt[i][n], spt[j+1-(1<<n)][n]);
}
```

Example usage

```
N = 10; // size of array
A = {1, 5, -3, 7, -2, 1, 6, -8, 4, -2};
sptBuild();
sptQuery(0, 9); // returns -8
sptQuery(1, 1); // return 5
sptQuery(1, 4); // returns -3
sptQuery(5, 8); // returns -8
```

4.8 Segment Tree

The size of the segment tree should be 4 times the data size. Building is $O(n)$. Querying and updating is $O(\log n)$.

4.8.1 Example 1 (no range updates)

This segment tree finds the maximum subsequence sum in an arbitrary range.

```
int A[50000];

struct node {
    int bestPrefix, bestSuffix, bestSum, sum;
    void merge(node& ls, node& rs) {
        bestPrefix
            = max(ls.bestPrefix, ls.sum + rs.bestPrefix);
        bestSuffix
            = max(rs.bestSuffix, rs.sum + ls.bestSuffix);
        bestSum
            = max(ls.bestSuffix + rs.bestPrefix,
              max(ls.bestSum, rs.bestSum));
        sum = ls.sum + rs.sum;
    }
} seg[200000];

void segBuild(int n, int l, int r) {
    if (l == r) {
        seg[n].bestPrefix = seg[n].bestSuffix
            = seg[n].bestSum = seg[n].sum = A[l];
        return;
    }
    int m = (l+r)/2;
    segBuild(2*n+1, l, m);
    segBuild(2*n+2, m+1, r);
    seg[n].merge(seg[2*n+1], seg[2*n+2]);
}

node segQuery(int n, int l, int r, int i, int j) {
    if (i <= l && r <= j) return seg[n];
    int m = (l+r)/2;
    if (m < i) return segQuery(2*n+2, m+1, r, i, j);
    if (m >= j) return segQuery(2*n+1, l, m, i, j);
    node ls = segQuery(2*n+1, l, m, i, j);
    node rs = segQuery(2*n+2, m+1, r, i, j);
    node a;
    a.merge(ls, rs);
    return a;
}

void segUpdate(int n, int l, int r, int i) {
    if (i < l || i > r) return;
    if (i == l && l == r) {
        seg[n].bestPrefix = seg[n].bestSuffix
            = seg[n].bestSum = seg[n].sum = A[l];
        return;
    }
    int m = (l+r)/2;
    segUpdate(2*n+1, l, m, i);
    segUpdate(2*n+2, m+1, r, i);
    seg[n].merge(seg[2*n+1], seg[2*n+2]);
}
```

4.8.2 Example 2 (with range updates)

This segment tree stores a series of booleans and allows swapping all booleans in any range.

```
struct node {
    int sum;
    bool inv;
    void apply(int x) {
        sum = x - sum;
        inv = !inv;
    }
    void split(node& ls, node& rs, int l, int m, int r) {
        if (inv) {
            ls.apply(m-l+1);
            rs.apply(r-m);
            inv = false;
        }
    }
    void merge(node& ls, node& rs) {
        sum = ls.sum + rs.sum;
    }
} seg[200000];

node segQuery(int n, int l, int r, int i, int j) {
    if (i <= l && r <= j) return seg[n];
    int m = (l+r)/2;
    seg[n].split(seg[2*n+1],seg[2*n+2],l,m,r);
    if (m < i) return segQuery(2*n+2, m+1, r, i, j);
    if (m >= j) return segQuery(2*n+1, l, m, i, j);
    node ls = segQuery(2*n+1, l, m, i, j);
    node rs = segQuery(2*n+2, m+1, r, i, j);
    node a;
    a.merge(ls, rs);
    return a;
}

void segUpdate(int n, int l, int r, int i, int j) {
    if (i > r || j < l) return;
    if (i <= l && r <= j) {
        seg[n].apply(r-l+1);
        return;
    }
    int m = (l+r)/2;
    seg[n].split(seg[2*n+1],seg[2*n+2],l,m,r);
    segUpdate(2*n+1, l, m, i, j);
    segUpdate(2*n+2, m+1, r, i, j);
    seg[n].merge(seg[2*n+1], seg[2*n+2]);
}
```

Example usage:

```
N = size of list;
segBuild(0, 0, N-1);
segQuery(0, 0, N-1, i, j); // queries range [i, j]
segUpdate(0, 0, N-1, i, j); // updates range [i, j] (you may need to add parameters)
```

4.9 Suffix Array

4.9.1 Notes

- Terminating character ($) is not required (unlike CP book), but it is useful to compute the longest common substring of multiple strings
- Use slow version if possible as it is shorter

4.9.2 Initialization

Complexity: $O\left(n \log^2 n\right)$

```
typedef vector<int> VI;

VI sa, ra, lcp;
string s;

void saInit() {
    int l = s.size();
    sa.resize(l);
    iota(sa.begin(), sa.end(), 0);
    ra.assign(s.begin(), s.end());
    for (int k = 1; k < l; k *= 2) {
        // To use radix sort, replace sort() with:
        // csort(l, k); csort(l, 0);
        sort(sa.begin(), sa.end(), [&](int a, int b){
            if (ra[a] != ra[b]) return ra[a] < ra[b];
            int ak = a+k < l ? ra[a+k] : -1;
            int bk = b+k < l ? ra[b+k] : -1;
            return ak < bk;
        });
        VI ra2(l); int x = 0;
        for (int i = 1; i < l; i++) {
            if (ra[sa[i]] != ra[sa[i-1]] ||
                sa[i-1]+k >= l ||
                ra[sa[i]+k] != ra[sa[i-1]+k]) x++;
            ra2[sa[i]] = x;
        }
        ra = ra2;
    }
}
```

4.9.3 Initialization (slow)

Complexity: $O\left(n^2 \log n\right)$

```
void saInit() {
    int l = s.size();
    sa.resize(l);
    iota(sa.begin(), sa.end(), 0);
    sort(sa.begin(), sa.end(), [](int a, int b) {
        return s.compare(a, -1, s, b, -1) < 0;
    });
}
```

4.9.4 Example suffix array

i	sa[i]	lcp[i]	Suffix
0	0	0	abacabacx
1	4	4	abacx
2	2	1	acabacx
3	6	2	acx
4	1	0	bacabacx
5	5	3	bacx
6	3	0	cabacx
7	7	1	cx
8	8	0	x

4.9.5 Longest Common Prefix array

Complexity: $O(n)$

```
void saLCP() {
    int l = s.size();
    lcp.resize(l);
    VI p(l), rsa(l);
    for (int i = 0; i < l; i++) {
        p[sa[i]] = (i) ? sa[i-1] : -1;
        rsa[sa[i]] = i;
    }
    int x = 0;
    for (int i = 0; i < l; i++) {
        // Note: The $ condition is optional and is
        // useful for finding longest common substring
        while (p[i] != -1 && p[i]+x < l &&
            s[i+x] == s[p[i]+x] && s[i+x] != '$') x++;
        lcp[rsa[i]] = x;
        if (x) x--;
    }
}
```

4.9.6 String matching

Returns a vector containing the zero-based index of the start of each match of `m` in `s`. Complexity: $O(m \log n)$

```
VI saFind(string const& m) {
    auto r = equal_range(sa.begin(), sa.end(), -1,
    [&](int i, int j) {
        int a = 1;
        if (i == -1) {swap(i, j); a = -1;}
        return a*s.compare(i, m.size(), m) < 0;
    });
    VI occ(r.first, r.second);
    sort(occ.begin(), occ.end()); // optional
    return occ;
}
```

4.9.7 Optional counting sort

Improves `saInit()` performance to $O(n \log n)$
Usually not necessary, about 4x speed up on a 1M string. However reduces performance in some cases. Not recommended.

```
void csort(int l, int k) {
    int m = max(300, l+1);
    VI c(m), sa2(l);
    for (int i = 0; i < l; i++) c[i+k<l ? ra[i+k]+1 : 0]++;
    for (int s = 0, i = 0; i < m; i++) {
        swap(c[i], s); s += c[i];
    }
    for (int i = 0; i < l; i++)
        sa2[c[sa[i]+k<l ? ra[sa[i]+k]+1 : 0]++] = sa[i];
    sa = sa2;
}
```

4.9.8 Example usage

```
s = "abacabacx";
saInit(); // Now sa[] is filled
saLCP();  // Now lcp[] is filled
```

5 Math and Other Algorithms

5.1 Cycle-Finding (Floyd's)

Sequence starts at x_0, x_μ is start of cycle, λ is cycle length. Complexity: $O(\mu + \lambda)$

```
int t = f(X0), h = f(f(X0)), mu = 0, lambda = 1;
while (t != h) { t = f(t); h = f(f(h)); }
h = X0;
while (t != h) { t = f(t); h = f(h); mu++; }
h = f(t);
while (t != h) { h = f(h); lambda++; }
```

5.2 Exponentiation by Squaring

Computes x^n. Complexity: $O(\log n)$ assuming multiplication and division are constant time.

```
result = 1
while n is nonzero
    if n is odd
        result *= x
        n-= 1
    x *= x
    n /= 2
```

5.3 Extended Euclidean and Modular Inverse

Complexity: $O(\log(\min(a, b)))$

```
int x, y, d;
void gcd(int a, int b) {
    if (b == 0) {x = 1; y = 0; d = a; return;}
    gcd(b, a % b);
    x -= y * (a / b);
    swap(x, y);
}
```

Finds $d = \gcd(a, b)$ and solves the equation $ax + by = d$. The equation $ax + by = c$ has a solution iff c is a multiple of $d = \gcd(a, b)$. If (x, y) is a solution, all other solutions have the form $(x + k\frac{b}{d}, y - k\frac{a}{d}), k \in \mathbb{Z}$.
To get modular inverse of a modulo m, do `gcd(a, m)` and the inverse is x (assuming inverse exists).
It is guaranteed that (x, y) is one of the two minimal pairs of Bézout coefficients. $|x| < \frac{b}{d}$ and $|y| < \frac{a}{d}$.

5.4 Fast Fourier Transform

Optimized Russian version. Complexity: $O(n \log n)$

```
typedef complex<double> base;

void fft (vector<base>& a, bool invert) {
    int n = (int) 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*M_PI/len * (invert ? -1 : 1);
        base wlen (cos(ang), sin(ang));
        for (int i=0; i<n; i+=len) {
            base w (1);
            for (int j=0; j<len/2; ++j) {
                base 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 (int i=0; i<n; ++i)
            a[i] /= n;
}
```

```
void multiply (const vector<int>& a,
               const vector<int>& b,
               vector<int>& res) {
    vector<base> fa(a.begin(), a.end()),
                 fb(b.begin(), b.end());
    size_t n = 1;
    while (n < max (a.size(), b.size())) n <<= 1;
    n <<= 1;
    fa.resize (n); fb.resize (n);

    fft (fa, false),  fft (fb, false);
    for (size_t i=0; i<n; ++i)
        fa[i] *= fb[i];
    fft (fa, true);

    res.resize (n);
    for (size_t i=0; i<n; ++i)
        res[i] = int (fa[i].real() + 0.5);
}
```

5.4.1 Fast Polynomial Multiplication (Integer)

A bit faster than FFT. Polynomials must be nonempty arrays in the range $[0, m)$ where $m = 2\,013\,265\,921 = 2^{31} - 2^{27} + 1$, a prime. Negative coefficients are not allowed. Complexity: $O(n \log n)$

```
void transform(const vector<int>& a, vector<int>& tA,
               int logN, int primitiveRoot) {
    tA.resize(1 << logN);
    for (int j = 0; j < (int)a.size(); j++) {
        unsigned int k = j << (32 - logN); // MUST be unsigned
        k = ((k >> 1) & 0x55555555) | ((k & 0x55555555) << 1);
        k = ((k >> 2) & 0x33333333) | ((k & 0x33333333) << 2);
        k = ((k >> 4) & 0x0f0f0f0f) | ((k & 0x0f0f0f0f) << 4);
        k = ((k >> 8) & 0x00ff00ff) | ((k & 0x00ff00ff) << 8);
        tA[(k >> 16) | (k << 16)] = a[j];
    }
    int root[LOG_MAX_LENGTH];
    root[LOG_MAX_LENGTH - 1] = primitiveRoot;
    for (int i = LOG_MAX_LENGTH - 1; i > 0; i--) {
        root[i - 1] = addMultiply(0, root[i], root[i]);
    }
    for (int i = 0; i < logN; i++) {
        int twiddle = 1;
        for (int j = 0; j < (1 << i); j++) {
            for (int k = j; k < (int)tA.size(); k += 2 << i) {
                int x = tA[k];
                int y = tA[k + (1 << i)];
                tA[k] = addMultiply(x, twiddle, y);
                tA[k + (1 << i)] =
                    addMultiply(x, MODULUS - twiddle, y);
            }
            twiddle = addMultiply(0, root[i], twiddle);
        }
    }
}
```

```
const int LOG_MAX_LENGTH = 27;
const int MODULUS = 2013265921;
const int PRIMITIVE_ROOT = 137;
const int PRIMITIVE_ROOT_INVERSE = 749463956;

int addMultiply(int x, int y, int z) {
    return (int)((x + y * (LL)z) % MODULUS);
}

void multiply (const vector<int>& a,
               const vector<int>& b,
               vector<int>& res) {
    int minN = int(a.size() - 1 + b.size());
    int logN = 0;
    while ((1 << logN) < minN) {
        logN++;
    }
    vector<int> tA, tB, nC;
    transform(a, tA, logN, PRIMITIVE_ROOT);
    transform(b, tB, logN, PRIMITIVE_ROOT);
    for (int j = 0; j < (int)tA.size(); j++)
        tA[j] = addMultiply(0, tA[j], tB[j]);
    transform(tA, nC, logN, PRIMITIVE_ROOT_INVERSE);

    res.resize(minN);
    int nInverse = MODULUS - ((MODULUS - 1) >> logN);
    for (int j = 0; j < minN; j++)
        res[j] = addMultiply(0, nInverse, nC[j]);
}
```

5.5 Gauss-Jordan Elimination

This code tries to choose pivots to minimize error. Complexity: $O(N^3)$

```
const double EP = 1e-9;
void rref(vector<vector<double> >& mat) {
    int R = mat.size(), C = mat[0].size();
    for (int i = 0; i < min(R,C); i++) {
        int rr = i;
        for (int r = i; r < R; r++)
            if (mat[r][i] > mat[rr][i]) rr = r;
        if (fabs(mat[rr][i]) < EP) continue;
        swap(mat[rr], mat[i]);
        for (int c = C-1; c >= i; c--)
            mat[i][c] /= mat[i][i];
        for (int r = 0; r < R; r++) if (r != i)
            for (int c = C-1; c >= i; c--)
                mat[r][c] -= mat[i][c] * mat[r][i];
    }
}
```

5.6 Union-Find Disjoint Sets

Complexity: $O(1)$ per operation. Note: $O(\log n)$ if one of union-by-rank or path compression is omitted

```
// ds[x] is parent of x, and dr[x] is rank
// rank is height of tree without path compression
vector<int> ds, dr;
int findSet(int i) {
    return ds[i] == i ? i : (ds[i] = findSet(ds[i]));
}
void unionSet(int i, int j) {
    int x = findSet(i), y = findSet(j);
    if (x == y) return; // Sometimes necessary if you are
                        // calculating additional info.
    if (dr[x] < dr[y]) ds[x] = y;
    else if (dr[x] > dr[y]) ds[y] = x;
    else {ds[x] = y; dr[y]++;}
}
bool sameSet(int i, int j) {
    return findSet(i) == findSet(j);
}
```

Example initialization:

```
dr.assign(N, 0);
ds.resize(N);
iota(begin(ds), end(ds), 0);
```

5.7 Sieve, Prime Factorization, Totient

This sieve stores the smallest prime divisor (sp). Use 64-bit to avoid overflowing `i*i`. If the prime factorization of n is $\prod_{i=1}^{k} p_i^{m_i}$, the factoring functions return a sorted list of (p_i, m_i). The number of divisors in n is $\prod_{i=1}^{k}(m_i+1)$, and the sum of all divisors is $\prod_{i=1}^{k} \frac{p_i^{m_i+1}-1}{p_i-1}$. To find Euler's totient function, use product over distinct prime numbers dividing n.

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

Sieve: $O(n \log\log n)$

```
typedef vector<pair<int, int>> VP;
typedef long long LL;
const int MAX_P = 70000;

vector<int> primes;
int sp[MAX_P];

void sieve() {
    for (LL i = 2; i < MAX_P; i++) {
        if (sp[i]) continue;
        sp[i] = i;
        primes.push_back(i);
        for (LL j=i*i;j<MAX_P;j+=i)
            if(!sp[j]) sp[j] = i;
    }
}
```

Prime factorization: $O\left(\frac{\sqrt{n}}{\log n}\right)$
Works for $n <$ MAX_P^2

```
VP primeFactorize(int n) {
    VP f;
    for (int p : primes) {
        if (LL(p)*p > n) break;
        int a = 0;
        while (n % p == 0) {
            n /= p; a++;
        }
        if (a) f.emplace_back(p, a);
    }
    if (n != 1) f.emplace_back(n, 1);
    return f;
}
```

Prime factorization: $O(\log n)$
Works for $n <$ MAX_P

```
VP primeFactorize(int n) {
    VP f;
    while (n != 1) {
        int a = 0, p = sp[n];
        while (n % p == 0) {
            n /= p; a++;
        }
        f.emplace_back(p, a);
    }
    return f;
}
```

Pollard's rho algorithm

Inputs: n, the integer to be factored and $f(x)$, a pseudo-random function modulo n ($f(x) = x^2 + c$, $c \neq 0$, $c \neq -2$ works fine)

Output: a non-trivial factor of n, or failure.

```
1. x = 2; y = 2; d = 1;
2. While d == 1:
 1. x = f(x)
 2. y = f(f(y))
 3. d = GCD(|x - y|, n)
3. If d == n, return failure.
4. Else, return d.
```

Note that this algorithm will return failure for all prime n, but it can also fail for composite n. In that case, use a different $f(x)$ and try again.

5.8 Simplex

Working version (copied from Stanford notebook). Complexity: Exponential in worst case, quite good on average

Two-phase simplex algorithm for solving linear programs. Maximize $c^T x$ subject to $Ax \leq b$ and $x \geq 0$. A should be an $m \times n$ matrix, b should be an m-dimensional vector, and c should be an n-dimensional vector. The optimal solution will be in vector x. It returns the value of the optimal solution (infinity if unbounded above, nan if infeasible).

```
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) {
    double inv = 1.0 / D[r][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] * inv;
    for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] *= inv;
    for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    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] < EPS) 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; // VALUE: 1.29032
  cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
  for (size_t i = 0; i < x.size(); i++) cerr << "␣" << x[i];
  cerr << endl;
  return 0;
}
```

6 Tricks for Bit Manipulation

6.1 GCC Builtins and Other Tricks

For these builtins, you can append `l` or `ll` to the function names to get the `long` or `long long` version.

`int __builtin_ffs(int x)`	Returns one plus the index of the least significant 1-bit of x. Returns 0 if $x = 0$.
`int __builtin_clz(unsigned int x)`	Returns the number of leading 0-bits in $x \neq 0$, starting at the most significant bit position. $\lfloor \log_2(n) \rfloor =$ `31 - __builtin_clz(n)`
`int __builtin_ctz(unsigned int x)`	Returns the number of trailing 0-bits in $x \neq 0$, starting at the most significant bit position.
`int __builtin_clrsb(int x)`	Returns the number of leading redundant sign bits in x, i.e. the number of bits following the most significant bit that are identical to it. There are no special cases for 0 or other values.
`int __builtin_popcount(unsigned int x)`	Returns the number of 1-bits in x. (Slow on x86 without SSE4 flag)
`int __builtin_parity(unsigned int x)`	Returns the parity of x, i.e. the number of 1-bits in x modulo 2.
`uintN_t __builtin_bswapN(uintN_t x)`	Returns x with the order of the bytes reversed. $N = 16, 32, 64$
`(x & (x - 1)) == 0`	Checks if x is a power of 2 (only one bit set). Note: 0 is edge case.
`(x + y - 1) / y`	Finds $\left\lceil \frac{x}{y} \right\rceil$ (positive integers only)

6.2 Lexicographically Next Bit Permutation

Suppose we have a pattern of N bits set to 1 in an integer and we want the next permutation of N 1 bits in a lexicographical sense. For example, if N is 3 and the bit pattern is 00010011, the next patterns would be 00010101, 00010110, 00011001, 00011010, 00011100, 00100011, and so forth. The following is a fast way to compute the next permutation.

```
int bs = 0b11111; // whatever is first bit permutation
int t = bs | (bs - 1); // t gets v's least significant 0 bits set to 1
// Next set to 1 the most significant bit to change,
// set to 0 the least significant ones, and add the necessary 1 bits.
bs = (t + 1) | (((~t & -~t) - 1) >> (__builtin_ctz(bs) + 1));
```

6.3 Loop Through All Subsets

For example, if **bs** $= 10110$, loop through **bt** $= 10100, 10010, 10000, 00110, 00100, 00010$

```
for (int bt = (bs-1) & bs; bt; bt = (bt-1) & bs) {
    int bu = bt ^ bs; // contains the opposite subset of bt (e.g. if bt = 10000, bu = 00110)
}
```

6.4 Parsing and Printing __int128

GCC supports `(unsigned) __int128` type on most platforms (notable exception is Windows). However, it does not currently support printing and parsing of those types.

```
string printint128(__int128 a) { // prints as decimal
    if (!a) return "0";
    string s;
    while (a) {
        s = char(llabs(a % 10) + '0') + s;
        if (-10 < a && a < 0) s = '-' + s;
        a /= 10;
    }
    return s;
}

__int128 parseint128(string s) { // parses decimal number
    __int128 a = 0, sgn = 1;
    for (char c : s) {
        if (c == '-') sgn *= -1; else a = a * 10 + sgn * (c - '0');
    }
    return a;
}
```

7 Math Formulas and Theorems

Arithmetic series

Arithmetic series: $a_n = a_1 + (n-1)d$
The sum of the first n terms of an arithmetic series is: $S_n = \frac{n(a_1+a_n)}{2} = \frac{n}{2}\left[2a_1 + (n-1)d\right]$

Catalan numbers (and Super Catalan)

Starting from $n = 0$, C_n: $1, 1, 2, 5, 14, 42, 132, 429, 1430$, S_n: $1, 1, 3, 11, 45, 197, 903, 4279$

$$C_0 = 1, C_n = \frac{1}{n+1}\binom{2n}{n} = \sum_{i=0}^{n-1} C_i C_{n-1-i} = \frac{2(2n-1)}{n+1}C_{n-1}, S_n = \frac{1}{n}\left(\left(6n-9\right)S_{n-1} - \left(n-3\right)S_{n-2}\right)$$

Chinese remainder theorem

Suppose $n_1 \cdots n_k$ are positive integers that are pairwise coprime. Then, for any series of integers $a_1 \cdots a_k$, there are an infinite number of solutions x where

$$\begin{cases} x &= a_1 \pmod{n_1} \\ & \cdots \\ x &= a_k \pmod{n_k} \end{cases}$$

All solutions x are congruent modulo $N = n_1 \cdots n_k$.

Ellipse

Equation is $\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1$, area is πab, distance from center to either focus is $f = \sqrt{a^2 - b^2}$.

Euler/Fermat little's theorem

For any prime p and integer a, $a^p \equiv a \pmod{p}$. If a is not divisible by p, then $a^{p-1} \equiv 1 \pmod{p}$ and a^{p-2} is the modular inverse of a modulo p. More generally, for any coprime n and a, $a^{\varphi(n)} \equiv 1 \pmod{n}$. $\varphi(n)$ is Euler's totient function, the number positive integers up to a given integer n that are relatively prime to n. If $\gcd(m, n) = 1$, then $\varphi(mn) = \varphi(m)\varphi(n)$ (multiplicative property). For all n and m, and $e \geq \log_2(m)$, it holds that $n^e \pmod{m} \equiv n^{\varphi(m)+e \mod \varphi(m)} \pmod{m}$. Starting from $n = 1$, $\varphi(n)$ values: 1, 1, 2, 2, 4, 2, 6, 4, 6.

Factoring

$a^2 - b^2 = (a+b)(a-b)$, $a^3 - b^3 = (a-b)(a^2 + ab + b^2)$, $a^3 + b^3 = (a+b)(a^2 - ab + b^2)$

Fermat's last theorem

No three positive integers a, b, and c can satisfy the equation $a^n + b^n = c^n$ for any integer value of n greater than 2.

Geometric series

The sum of the first n terms of a geometric series is: $1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$
If the absolute value of r is less than one, the series converges as n goes to infinity: $\frac{1}{1-r}$

Great-circle distance

Let ϕ_1, λ_1 and ϕ_2, λ_2 be the geographical latitude and longitude of two points 1 and 2, and R be sphere radius. This Haversine Formula provides higher numerical precision.

$$a = \sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)$$

$$\text{Great-circle distance} = 2 \times R \times \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right)$$

Lagrange multiplier

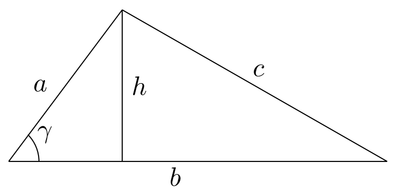
To maximize/minimize $f(x, y)$ subject to $g(x, y) = 0$, you may be able to set partial derivatives of $\mathcal{L}$ to zero, where $\mathcal{L}(x, y, \lambda) = f(x, y) - \lambda \cdot g(x, y)$

Sum formulas

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$

Triangles

Area $= \frac{1}{2}bh = \frac{1}{2}ab\sin\gamma = \sqrt{s(s-a)(s-b)(s-c)}$ where s is the semiperimeter
Radius of incircle: $r = \frac{\text{Area}}{s}$
Law of sines: Diameter of circumcircle $= \frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$
Law of cosines: $c^2 = a^2 + b^2 - 2ab\cos C$ / $\cos C = \frac{a^2+b^2-c^2}{2ab}$



7.1 Dynamic programming

7.1.1 Convex hull optimization

Suppose that you have a large set of linear functions $y = m_i x + b_i$. Each query consists of a value of x and asks for the minimum value of y that can be obtained if we select one of the linear functions and evaluate it at x. Sort the lines in descending order by slope, and add them one by one. Suppose l_1, l_2, and l_3 are the second line from the top, the line at the top, and the line to be added, respectively. Then, l_2 becomes irrelevant if and only if the intersection point of l_1 and l_3 is to the left of the intersection of l_1 and l_2. The overall shape of the lines will be concave downwards.

```
// Represent lines as PT(m, b) where y = mx + b
typedef pair<LL,LL> PT;
double ipl(PT l1, PT l2) {
    return double(l1.second - l2.second) / double(l2.first - l1.first);
}

vector<PT> lines;
// To add a line to the data structure
PT line; // The new line to add
while (lines.size() >= 2 && ipl(line, lines[lines.size()-2]) < ipl(lines[lines.size()-2], lines[lines.size()-1]))
    lines.pop_back();
lines.push_back(line);

// To find the lowest point at a certain x
auto it = lower_bound(begin(lines), end(lines), <the x>, [&](const PT& l, LL x){
    double xv = (l == lines[SZ(lines)-1]) ? 1e100 : ipl(l, *(&l + 1));
    return xv < x;
});
```

7.1.2 Knuth-Yao optimization

This optimization converts certain $O(n^3)$ DP recurrences into $O(n^2)$.

Convex quadrangle inequality: $f(i,k) + f(j,l) \geq f(i,l) + f(j,k)$ for $i \leq j \leq k \leq l$

Concave quadrangle inequality: $f(i,k) + f(j,l) \leq f(i,l) + f(j,k)$ for $i \leq j \leq k \leq l$

If $A[i][j]$ is the smallest k that gives the optimal answer (or largest k, doesn't matter), then $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$. Let cost function $c(i,j) = w(i,j) + \min[c(i,k-1) + c(k,j)]$ (or max). For maximization, w must be monotone and satisfy convex QI. For minimization, w must be monotone and satisfy concave QI. $f(i,j) = a_i + a_{i+1} + \cdots + a_j$ satisfies both convex and concave QI. $f(i,j) = a_i a_{i+1} \cdots a_j$ satisfies concave QI if all $a_k \geq 1$.

7.2 Game theory

Applicable to impartial games under the normal play convention.

- Grundy number: Represents Nim pile size. $g(x) = \min(n \geq 0 : n \neq g(y)\, \forall y \in f(x))$
- Remoteness: Moves left if winner forces a win as soon as possible and loser tries to lose as slowly as possible. $r(x) = 0$ if terminal, $1 +$ least even $r(k), k \in f(x)$ if such exists, otherwise $1 +$ greatest odd $r(k), k \in f(x)$.
- Suspense function: Moves left if winner tries to play as long as possible and loser tries to lose as soon as possible. $s(x) = 0$ if terminal, $1 +$ greatest even $r(k), k \in f(x)$ if such exists, otherwise $1 +$ least odd $r(k), k \in f(x)$.

Losing conditions:

- Ordinary sum of games: The player whose turn it is must choose one of the games and make a move in it. A player who is not able to move in all the games loses. $g_1 \oplus g_2 \oplus \cdots \oplus g_n = 0$.
- Union of games: The player whose turn it is must choose at least one of these games and make one move in every chosen one. A player who is not able to move loses. $\forall i\ g_i = 0$.
- Selective compound: The player whose turn it is must choose at least one of these subgames, but he cannot choose all of them and then make one move in every chosen one. A player who is not able to move loses. $g_1 = g_2 = ... = g_n$.
- Conjunctive compound: The player whose turn it is must make a move in every subgame. A player who is not able to move loses. $\min(r_1, r_2, \cdots, r_n)$ is even.
- Continued conjunctive compound: The player whose turn it is must make a move in every subgame he can and the game ends and a player loses only if he cannot move anywhere. $\max(s_1, s_2, \cdots, s_n)$ is even.

7.3 Prime numbers

(all primes up to 547, and selected ones thereafter) 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 577 607 641 661 701 739 769 811 839 877 911 947 983 1019 1049 1087 1109 1153 1193 1229 1277 1297 1321 1381 1429 1453 1487 1523 1559 1597 1619 1663 1699 1741 1783 1823 1871 1901 1949 1993 2017 2063 2089 2131 2161 2221 2267 2293 2339 2371 2393 2437 2473 2539 2579 2621 2663 2689 2713 2749 2791 2833 2861 2909 2957 3001 3041 3083 3137 3187 3221 3259 3313 3343 3373 3433 3467 3517 3541 3581 3617 3659 3697 3733 3779 3823 3863 3911 3931 4001 4021 4073 4111 4153 4211 4241 4271 4327 4363 4421 4457 4507 4547 4591 4639 4663 4721 4759 4799 4861 4909 4943 4973 5009 5051 5099 5147 5189 5233 5281 5333 5393 5419 5449 5501 5527 5573 5641 5659 5701 5743 5801 5839 5861 5897 5953 6029 6067 6101 6143 6199 6229 6271 6311 6343 6373 6427 6481 6551 6577 6637 6679 6709 6763 6803 6841 6883 6947 6971 7001 7043 7109 7159 7211 7243 7307 7349 7417 7477 7507 7541 7573 7603 7649 7691 7727 7789 7841 7879 13763 19213 59263 77339 117757 160997 287059 880247 2911561 4729819 9267707 9645917 11846141 23724047 39705719 48266341 473283821 654443183 793609931 997244057 8109530161 8556038527 8786201093 9349430521 70635580657 73695102059 79852211099 97982641721 219037536857 273750123551 356369453281 609592207993 2119196502847 3327101349167 4507255137769 7944521201081 39754306037983 54962747021723 60186673819997 98596209151961