Reference Material

HaKings

Contents

1 Header

```
#include <bits/stdc++.h>
#define _ ios_base::sync_with_stdio(0), cin.tie(0), cin.tie(0), cout.tie(0), cout.precision(15);
#define INF 1000000000
#define FOR(i, a, b) for(int i=int(a); i<int(b); i++)
#define FORC(cont, it) for(decltype((cont).begin()) it=(cont).begin(); it!=(cont).end(); it++)
#define pb push_back
#define mp make_pair
#define all(x) (x).begin(), (x).end()
using namespace std; typedef long long ll; typedef pair<int, int> ii; typedef vector<int> vi; typedef vector<ii> vii; typedef vector<vi> vvi;
```

---

2 Graph Theory

2.1 Strongly Connected Components

$$O(V + E)$$

Partitions the vertices of a directed graph into strongly connected components.
A strongly connected component is a subset of a graph where every vertex is reachable from every other vertex.
Returns V where V[i] is the index of the component of node i.

```
vi low1, num1, components;
int counter1, SCCindex;
vector<bool> visited;
stack<int> S;

void dfs(Graph &g, int cv) {
  low1[cv] = num1[cv] = counter1++;
  S.push(cv);
  visited[cv] = true;
  FORC(g.edges[cv], edge) {
    if(num1[edge->to] == -1)
      dfs(g, edge->to);
    if(visited[edge->to])
      low1[cv] = min(low1[cv], low1[edge->to]);
  }
  if(low1[cv] == num1[cv]) {
    int index = SCCindex++;
    while(true) {
      int v = S.top(); S.pop(); visited[v] = 0;
      components[v] = index;
      if (cv == v)
```

```
        break;
    }
  }
}

vi stronglyConnectedComponents(Graph &g) {
  counter1 = 0, SCCindex = 0;
  visited = vector<bool>(g.V, 0);
  num1 = vi(g.V, -1), low1 = vi(g.V, 0), components = vi(g.V, 0);
  S = stack<int>();
  FOR(i, 0, g.V)
    if(num1[i] == -1)
      dfs(g, i);
  return components;
}
```

2.2 Articulation Points

$$O(V + E)$$

Finds all articulation points and bridges in a graph.
An articulation point is a vertex whose removal would disconnect the graph.
An bridge is a vertex whose removal disconnects the graph.

```
vi low2, num2, parent, strongPoints;
int counter2, root, rootChildren;
void dfs1(Graph &g, int v) {
  low2[v] = num2[v] = counter2++;
  FORC(g.edges[v], edge) {
    if(num2[edge->to] == -1) {
      parent[edge->to] = v;
      if(v == root) rootChildren++;
      dfs1(g, edge->to);
      if(low2[edge->to] >= num2[v]) strongPoints[v] = true;
      if(low2[edge->to] > num2[v]) edge->strong = g.edges[edge->to][edge->
          backEdge].strong = true;
      low2[v] = min(low2[v], low2[edge->to]);
    } else if(edge->to != parent[v])
      low2[v] = min(low2[v], num2[edge->to]);
  }
}

vi articulationPointsAndBridges(Graph &g) {
  counter2 = 0;
  num2 = vi(g.V, -1), low2 = vi(g.V, 0), parent = vi(g.V, -1), strongPoints =
      vi(g.V, 0);
  FOR(i, 0, g.V)
```

```
    if(num2[i] == -1) {
      root = i, rootChildren = 0;
      dfs1(g, i);
      strongPoints[root] = rootChildren > 1;
    }
  return strongPoints;
}
```

2.3 Eulerian Path

$$O(V + E)$$

Partitions the vertices of a directed graph into strongly connected components.
A strongly connected component is a subset of a graph where every vertex is reachable from every other vertex.
Returns V where V[i] is the index of the component of node i.

```
vi low1, num1, components;
int counter1, SCCindex;
vector<bool> visited;
stack<int> S;

void dfs(Graph &g, int cv) {
  low1[cv] = num1[cv] = counter1++;
  S.push(cv);
  visited[cv] = true;
  FORC(g.edges[cv], edge) {
    if(num1[edge->to] == -1)
      dfs(g, edge->to);
    if(visited[edge->to])
      low1[cv] = min(low1[cv], low1[edge->to]);
  }
  if(low1[cv] == num1[cv]) {
    int index = SCCindex++;
    while(true) {
      int v = S.top(); S.pop(); visited[v] = 0;
      components[v] = index;
      if (cv == v)
        break;
    }
  }
}

vi stronglyConnectedComponents(Graph &g) {
  counter1 = 0, SCCindex = 0;
  visited = vector<bool>(g.V, 0);
```

```
  num1 = vi(g.V, -1), low1 = vi(g.V, 0), components = vi(g.V, 0);
  S = stack<int>();
  FOR(i, 0, g.V)
    if(num1[i] == -1)
      dfs(g, i);
  return components;
}
```

2.4 Max Bipartite Matching

$$O(VE)$$

Nodes in the left set must be nodes [0, left).
g must be unweighted directed bipartite graph.
match[r] = l, where r belongs to R and l belongs to L.

```
int augment(Graph &g, int cv, vi &match, vi &visited) {
  if(visited[cv]) return 0;
  visited[cv] = 1;
  FORC(g.edges[cv], edge)
    if(match[edge->to] == -1 || augment(g, match[edge->to], match, visited))
      return match[edge->to] = cv, 1;
  return 0;
}

int maxBipartiteMatching(Graph &g, int left) {
  int MCBM = 0;
  vi match(g.V, -1);
  FOR(cv, 0, left) {
    vi visited(left, 0);
    MCBM += augment(g, cv, match, visited);
  }
  return MCBM;
}
```

2.5 Edmonds-Karp

$$O(VE^2)$$

Finds a the maxflow from source to sink of a directed graph.
The weight of an edge denotes the capacity of the edge.
The negative weight edges are the edges with flow.

```
int augment(MatrixGraph &g, int flow, vi &parent, int source, int cv, int
    minEdge) {
  if(cv == source)
    return minEdge;
  if(parent[cv] != -1) {
    flow = augment(g, flow, parent, source, parent[cv], min(minEdge, g.edges[
        parent[cv]][cv].weight));
    g.edges[parent[cv]][cv].weight -= flow;
    g.edges[cv][parent[cv]].weight += flow;
  }
  return flow;
}

int maxFlow(MatrixGraph &g, int source, int sink) {
  int mf = 0, flow = -1;
  while(flow) {
    vi distanceTo(g.V, INF);
    distanceTo[source] = 0;
    queue<int> q; q.push(source);
    vi parent(g.V, -1);
    while(!q.empty()) {
      int cv = q.front(); q.pop();
      if(cv == sink) break;
      FOR(i, 0, g.V)
        if(g.edges[cv][i].weight > 0 && distanceTo[i] == INF)
          distanceTo[i] = distanceTo[cv] + 1, q.push(i), parent[i] = cv;
    }
    mf += flow = augment(g, 0, parent, source, sink, INF);
  }
  return mf;
}
```

2.6 Dinic

$$O(EV^2)$$

Adjacency list implementation of Dinic's blocking flow algorithm.
This is very fast in practice, and only loses to push-relabel flow.
OUTPUT: - maximum flow value - To obtain actual flow values, look at edges with capacity ¿ 0 (zero capacity edges are residual edges).

```
typedef long long LL;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
  LL rcap() { return cap - flow; }
```

```
};

struct Dinic {
  int N;
  vector<vector<Edge> > G;
  vector<vector<Edge *> > Lf;
  vector<int> layer;
  vector<int> Q;

  Dinic(int N) : N(N), G(N), Q(N) {}

  void AddEdge(int from, int to, int cap) {
    if (from == to) return;
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }

  LL BlockingFlow(int s, int t) {
    layer.clear(); layer.resize(N, -1);
    layer[s] = 0;
    Lf.clear(); Lf.resize(N);

    int head = 0, tail = 0;
    Q[tail++] = s;
    while (head < tail) {
      int x = Q[head++];
      for (int i = 0; i < G[x].size(); i++) {
        Edge &e = G[x][i]; if (e.rcap() <= 0) continue;
        if (layer[e.to] == -1) {
          layer[e.to] = layer[e.from] + 1;
          Q[tail++] = e.to;
        }
        if (layer[e.to] > layer[e.from]) {
          Lf[e.from].push_back(&e);
        }
      }
    }
    if (layer[t] == -1) return 0;

    LL totflow = 0;
    vector<Edge *> P;
    while (!Lf[s].empty()) {
      int curr = P.empty() ? s : P.back()->to;
      if (curr == t) { // Augment
        LL amt = P.front()->rcap();
        for (int i = 0; i < P.size(); ++i) {
          amt = min(amt, P[i]->rcap());
        }
        totflow += amt;
        for (int i = P.size() - 1; i >= 0; --i) {
```

```
          P[i]->flow += amt;
          G[P[i]->to][P[i]->index].flow -= amt;
          if (P[i]->rcap() <= 0) {
            Lf[P[i]->from].pop_back();
            P.resize(i);
          }
        }
      } else if (Lf[curr].empty()) { // Retreat
        P.pop_back();
        for (int i = 0; i < N; ++i)
          for (int j = 0; j < Lf[i].size(); ++j)
            if (Lf[i][j]->to == curr)
              Lf[i].erase(Lf[i].begin() + j);
      } else { // Advance
        P.push_back(Lf[curr].back());
      }
    }
    return totflow;
  }

  LL GetMaxFlow(int s, int t) {
    LL totflow = 0;
    while (LL flow = BlockingFlow(s, t))
      totflow += flow;
    return totflow;
  }
};
```

2.7 Min Cost Max Flow

Implementation of min cost max flow algorithm using adjacency matrix (Edmonds and Karp 1972).
This implementation keeps track of forward and reverse edges separately (so you can set cap[i][j] != cap[j][i]).
For a regular max flow, set all edge costs to 0.
Running time, $O(|V|^2)$ cost per augmentation
max flow: $O(|V|^3)$ augmentations
min cost max flow: $O(|V|^4 * MAX_EDGE_COST)$ augmentations
INPUT:
- graph, constructed using AddEdge()
- source
- sink
OUTPUT:
- (maximum flow value, minimum cost value)
- To obtain the actual flow, look at positive values only.

```
typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
  int N;
  VVL cap, flow, cost;
  VI found;
  VL dist, pi, width;
  VPII dad;

  MinCostMaxFlow(int N) :
    N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
    found(N), dist(N), pi(N), width(N), dad(N) {}

  void AddEdge(int from, int to, L cap, L cost) {
    this->cap[from][to] = cap;
    this->cost[from][to] = cost;
  }

  void Relax(int s, int k, L cap, L cost, int dir) {
    L val = dist[s] + pi[s] - pi[k] + cost;
    if (cap && val < dist[k]) {
      dist[k] = val;
      dad[k] = make_pair(s, dir);
      width[k] = min(cap, width[s]);
    }
  }

  L Dijkstra(int s, int t) {
    fill(found.begin(), found.end(), false);
    fill(dist.begin(), dist.end(), INF);
    fill(width.begin(), width.end(), 0);
    dist[s] = 0;
    width[s] = INF;

    while (s != -1) {
      int best = -1;
      found[s] = true;
      for (int k = 0; k < N; k++) {
        if (found[k]) continue;
        Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
        Relax(s, k, flow[k][s], -cost[k][s], -1);
```

```
        if (best == -1 || dist[k] < dist[best]) best = k;
      }
      s = best;
    }

    for (int k = 0; k < N; k++)
      pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
  }

  pair<L, L> GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
    while (L amt = Dijkstra(s, t)) {
      totflow += amt;
      for (int x = t; x != s; x = dad[x].first) {
        if (dad[x].second == 1) {
          flow[dad[x].first][x] += amt;
          totcost += amt * cost[dad[x].first][x];
        } else {
          flow[x][dad[x].first] -= amt;
          totcost -= amt * cost[x][dad[x].first];
        }
      }
    }
    return make_pair(totflow, totcost);
  }
};
```

2.8 PushRelabel

$$O(V^3)$$

Adjacency list implementation of FIFO push relabel maximum flow with the gap relabeling heuristic.
This implementation is significantly faster than straight Ford-Fulkerson.
It solves random problems with 10000 vertices and 1000000 edges in a few seconds, though it is possible to construct test cases that achieve the worst-case.
OUTPUT:
- maximum flow value
- To obtain the actual flow values, look at all edges with capacity ¿ 0 (zero capacity edges are residual edges).

```
typedef long long LL;

struct Edge {
```

```
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct PushRelabel {
  int N;
  vector<vector<Edge> > G;
  vector<LL> excess;
  vector<int> dist, active, count;
  queue<int> Q;

  PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N), count(2*N) {}

  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }

  void Enqueue(int v) {
    if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v); }
  }

  void Push(Edge &e) {
    int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
    if (dist[e.from] <= dist[e.to] || amt == 0) return;
    e.flow += amt;
    G[e.to][e.index].flow -= amt;
    excess[e.to] += amt;
    excess[e.from] -= amt;
    Enqueue(e.to);
  }

  void Gap(int k) {
    for (int v = 0; v < N; v++) {
      if (dist[v] < k) continue;
      count[dist[v]]--;
      dist[v] = max(dist[v], N+1);
      count[dist[v]]++;
      Enqueue(v);
    }
  }

  void Relabel(int v) {
    count[dist[v]]--;
    dist[v] = 2*N;
    for (int i = 0; i < G[v].size(); i++)
      if (G[v][i].cap - G[v][i].flow > 0)
  dist[v] = min(dist[v], dist[G[v][i].to] + 1);
```

```
      count[dist[v]]++;
      Enqueue(v);
    }

    void Discharge(int v) {
      for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);
      if (excess[v] > 0) {
        if (count[dist[v]] == 1)
    Gap(dist[v]);
        else
    Relabel(v);
      }
    }

    LL GetMaxFlow(int s, int t) {
      count[0] = N-1;
      count[N] = 1;
      dist[s] = N;
      active[s] = active[t] = true;
      for (int i = 0; i < G[s].size(); i++) {
        excess[s] += G[s][i].cap;
        Push(G[s][i]);
      }

      while (!Q.empty()) {
        int v = Q.front();
        Q.pop();
        active[v] = false;
        Discharge(v);
      }

      LL totflow = 0;
      for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
      return totflow;
    }
};
```

2.9 Min Cost Matching

$$O(V^3)$$

Min cost bipartite matching via shortest augmenting paths.
In practice, it solves 1000x1000 problems in around 1 second. cost[i][j] = cost for pairing left node i with right node j
Lmate[i] = index of right node that left node i pairs with
Rmate[j] = index of left node that right node j pairs with
The values in cost[i][j] may be positive or negative.

To perform maximization, simply negate the cost[][] matrix.

```
typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
  int n = int(cost.size());

  // construct dual feasible solution
  VD u(n);
  VD v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
  }

  // construct primal solution satisfying complementary slackness
  Lmate = VI(n, -1);
  Rmate = VI(n, -1);
  int mated = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (Rmate[j] != -1) continue;
      if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
  Lmate[i] = j;
  Rmate[j] = i;
  mated++;
  break;
      }
    }
  }

  VD dist(n);
  VI dad(n);
  VI seen(n);

  // repeat until primal solution is feasible
  while (mated < n) {

    // find an unmatched left node
    int s = 0;
    while (Lmate[s] != -1) s++;

    // initialize Dijkstra
```

```
  fill(dad.begin(), dad.end(), -1);
  fill(seen.begin(), seen.end(), 0);
  for (int k = 0; k < n; k++)
    dist[k] = cost[s][k] - u[s] - v[k];

  int j = 0;
  while (true) {

    // find closest
    j = -1;
    for (int k = 0; k < n; k++) {
if (seen[k]) continue;
if (j == -1 || dist[k] < dist[j]) j = k;
    }
    seen[j] = 1;

    // termination condition
    if (Rmate[j] == -1) break;

    // relax neighbors
    const int i = Rmate[j];
    for (int k = 0; k < n; k++) {
if (seen[k]) continue;
const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
if (dist[k] > new_dist) {
  dist[k] = new_dist;
  dad[k] = j;
}
    }
  }

  // update dual variables
  for (int k = 0; k < n; k++) {
    if (k == j || !seen[k]) continue;
    const int i = Rmate[k];
    v[k] += dist[k] - dist[j];
    u[i] -= dist[k] - dist[j];
  }
  u[s] += dist[j];

  // augment along path
  while (dad[j] >= 0) {
    const int d = dad[j];
    Rmate[j] = Rmate[d];
    Lmate[Rmate[j]] = j;
    j = d;
  }
  Rmate[j] = s;
  Lmate[s] = j;

```

```
    mated++;
  }

  double value = 0;
  for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];

  return value;
}
```

2.10 Min Cut

$$O(|V|^3)$$

Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
OUTPUT:
- (min cut value, nodes in half of min cut)

```
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);
}
```

2.11 Edmonds Graph Matching

```
struct edge {
   int v, nx;
};
const int MAXN = 1000, MAXE = 2000;
edge graph[MAXE];
int last[MAXN], match[MAXN], px[MAXN], base[MAXN], N, edges;
bool used[MAXN], blossom[MAXN], lused[MAXN];
inline void add_edge(int u, int v) {
   graph[edges] = (edge) {v, last[u]};
   last[u] = edges++;
   graph[edges] = (edge) {u, last[v]};
   last[v] = edges++;
}
void mark_path(int v, int b, int children) {
   while (base[v] != b) {
      blossom[base[v]] = blossom[base[match[v]]] = true;
      px[v] = children;
      children = match[v];
      v = px[match[v]];
   }
}
int lca(int a, int b) {
   memset(lused, 0, N);
   while (1) {
      lused[a = base[a]] = true;
      if (match[a] == -1)
         break;
      a = px[match[a]];
   }
   while (1) {
      b = base[b];
      if (lused[b])
         return b;
      b = px[match[b]];
   }
}
int find_path(int root) {
   memset(used, 0, N);
```

```
    memset(px, -1, sizeof(int) * N);
    for (int i = 0; i < N; ++i)
        base[i] = i;
    used[root] = true;
    queue<int> q;
    q.push(root);
    int v, e, to, i;
    while (!q.empty()) {
        v = q.front(); q.pop();
        for (e = last[v]; e >= 0; e = graph[e].nx) {
            to = graph[e].v;
            if (base[v] == base[to] || match[v] == to)
                continue;
            if (to == root || (match[to] != -1 && px[match[to]] != -1)) {
                int curbase = lca(v, to);
                memset(blossom, 0, N);
                mark_path(v, curbase, to);
                mark_path(to, curbase, v);
                for (i = 0; i < N; ++i)
                    if (blossom[base[i]]) {
                        base[i] = curbase;
                        if (!used[i]) {
                            used[i] = true;
                            q.push(i);
                        }
                    }
            } else if (px[to] == -1) {
                px[to] = v;
                if (match[to] == -1)
                    return to;
                to = match[to];
                used[to] = true;
                q.push(to);
            }
        }
    }
    return -1;
}
void build_pre_matching() {
    int u, e, v;
    for (u = 0; u < N; ++u)
        if (match[u] == -1)
            for (e = last[u]; e >= 0; e = graph[e].nx) {
                v = graph[e].v;
                if (match[v] == -1) {
                    match[u] = v;
                    match[v] = u;
                    break;
                }
            }
```

```
}
void edmonds() {
   memset(match, 0xff, sizeof(int) * N);
   build_pre_matching();
   int i, v, pv, ppv;
   for (i = 0; i < N; ++i)
      if (match[i] == -1) {
         v = find_path(i);
         while (v != -1) {
            pv = px[v], ppv = match[pv];
            match[v] = pv, match[pv] = v;
            v = ppv;
         }
      }
}
```

2.12 Dijkstra

$$O((V + E)logV)$$

Finds the shortest path from source to every other vertex.

```
vi dijkstra(Graph &g, int src) {
  vi dist(g.V, INF);
  dist[src] = 0;
  priority_queue<ii, vii, greater<ii> > pq;
  pq.push(ii(0, src));
  while(!pq.empty()) {
    int cv = pq.top().second;
    int d = pq.top().first;
    pq.pop();
    if(d > dist[cv]) continue;
    FORC(g.edges[cv], edge)
      if(dist[edge->to] > dist[cv] + edge->weight) {
        dist[edge->to] = dist[cv] + edge->weight;
        pq.push(ii(dist[edge->to], edge->to));
      }
  }
  return dist;
}
```

2.13 Kruskal

$$O(ELogV)$$

Finds a minimum spanning tree of a undirected graph.
Returns the indices of the edges that are int the MST.

```cpp
int *comparator1;
bool compare(int a, int b) { return comparator1[a] < comparator1[b]; }
vi kruskal(vii &edges, int weight[], int V) {
  vi order(edges.size()), minTree;
  UnionFindDS ds(V);
  comparator1 = weight;
  FOR(i, 0, order.size()) order[i] = i;
  sort(order.begin(), order.end(), compare);
  for(int i=0; i<int(edges.size()) && int(minTree.size()) < V - 1; i++)
    if(!ds.connected(edges[order[i]].first, edges[order[i]].second)) {
      ds.connect(edges[order[i]].first, edges[order[i]].second);
      minTree.pb(order[i]);
    }
  return minTree;
}
```

2.14 Prim

$$O(ELogV)$$

Finds a minimum spanning tree of a undirected graph.
Returns a list of edges (node, indexOfEdge).

```cpp
Graph* comparator2;
struct Compare { bool operator()(ii a, ii b) { return comparator2->edges[a.
    first][a.second].weight > comparator2->edges[b.first][b.second].weight;} };
vii prim(Graph &g) {
  vi visited(g.V, 0);
  visited[0] = 1;
  vii tree; //list of edges in the MST
  int visitedNodes = 1;
  comparator2 = &g;
  priority_queue<ii, vector<ii>, Compare> pq;
  int cv = 0;
  while(visitedNodes != g.V) {
    FORC(g.edges[cv], edge)
      if(!visited[edge->to])
        pq.push(ii(cv, edge - g.edges[cv].begin()));
    ii nextEdge;
    do {
      nextEdge = pq.top();
      pq.pop();
    } while(visited[g.edges[nextEdge.first][nextEdge.second].to] && !pq.empty()
      );
    tree.pb(nextEdge);
    cv = g.edges[nextEdge.first][nextEdge.second].to;
    visitedNodes++;
```

```
    visited[cv] = 1;
  }
  return tree;
}
```

2.15 All Nodes Longest Path to Leaf in Tree

$$O(V + E)$$

Returns V where V[i].second contains the height of the tree if node i is the root. V[i].first contains the index of the next node in the longest path towards a leaf.

```
int getLongestPathDown(Graph &g, int cv, vii &longestPathDown, vii &
    secondLongestPathDown, vi &parent) {
  FORC(g.edges[cv], edge) {
    if(edge->to != parent[cv]) {
      parent[edge->to] = cv;
      int pathDownLength = 1 + getLongestPathDown(g, edge->to, longestPathDown,
           secondLongestPathDown, parent);
      if(pathDownLength > longestPathDown[cv].second) {
        secondLongestPathDown[cv] = longestPathDown[cv];
        longestPathDown[cv] = ii(edge->to, pathDownLength);
      } else if(pathDownLength > secondLongestPathDown[cv].second) {
        secondLongestPathDown[cv] = ii(edge->to, pathDownLength);
      }
    }
  }
  return longestPathDown[cv].second;
}

void getLongestPath(Graph &g, vii &longestPath) {
  longestPath.assign(g.V, ii(-1, 0));
  vii longestPathDown(g.V, ii(-1, 1)), secondLongestPathDown(g.V, ii(-1, 1)),
      secondLongestPath(g.V, ii(-1, 0));
  vi parent(g.V, -1);
  getLongestPathDown(g, 0, longestPathDown, secondLongestPathDown, parent);
  queue<int> q;
  q.push(0);
  while(!q.empty()) {
    int cv = q.front(); q.pop();
    FORC(g.edges[cv], edge)
      if(edge->to != parent[cv])
        q.push(edge->to);
    if(parent[cv] == -1) {
      longestPath[cv] = longestPathDown[cv];
      secondLongestPath[cv] = secondLongestPathDown[cv];
    } else {
```

```
      ii longestPathThroughParent = ii(parent[cv], (longestPath[parent[cv]].
          first != cv ? longestPath[parent[cv]].second : secondLongestPath[
          parent[cv]].second)+1);
      if(longestPathThroughParent.second >= longestPathDown[cv].second) {
        longestPath[cv] = longestPathThroughParent;
        secondLongestPath[cv] = longestPathDown[cv];
      } else if(longestPathThroughParent.second >= secondLongestPathDown[cv].
          second) {
        longestPath[cv] = longestPathDown[cv];
        secondLongestPath[cv] = longestPathThroughParent;
      } else {
        longestPath[cv] = longestPathDown[cv];
        secondLongestPath[cv] = secondLongestPathDown[cv];
      }
    }
  }
}
```

3 Number Theory

3.1 Sieve of Atkin

Obtains primes in the range [1, n]

```
typedef vector<ll> vll;
vll primes;
void sieve_atkins(ll n) {
   vector<bool> isPrime(n + 1);
   isPrime[2] = isPrime[3] = true;
   for (ll i = 5; i <= n; i++)
      isPrime[i] = false;

   ll lim = ceil(sqrt(n));
   for (ll x = 1; x <= lim; x++) {
      for (ll y = 1; y <= lim; y++) {
         ll num = (4 * x * x + y * y);
         if (num <= n && (num % 12 == 1 || num % 12 == 5))
            isPrime[num] = true;
         num = (3 * x * x + y * y);
         if (num <= n && (num % 12 == 7))
            isPrime[num] = true;
         if (x > y) {
            num = (3 * x * x - y * y);
            if (num <= n && (num % 12 == 11))
               isPrime[num] = true;
         }
      }
```

```
    }

    for (ll i = 5; i <= lim; i++)
        if (isPrime[i])
            for (ll j = i * i; j <= n; j += i)
                isPrime[j] = false;

    for (ll i = 2; i <= n; i++)
        if (isPrime[i])
            primes.pb(i);
}
```

3.2 Sieve of Eratosthenes

Obtains primes in the range [1, n]

```
#define SIZE 1000000
bitset<SIZE> sieve;
void buildSieve() {
  sieve.set();
  sieve[0] = sieve[1] = 0;
  int root = sqrt(SIZE);
  FOR(i, 2, root+1)
    if (sieve[i])
      for(int j = i*i; j < SIZE; j+=i)
        sieve[j] = 0;
}

vi primesList;
void buildPrimesList() {
  if(!sieve[2])
    buildSieve();
  primesList.reserve(SIZE/log(SIZE));
  FOR(i, 2, SIZE+1)
    if(sieve[i])
      primesList.pb(i);
}
```

3.3 Extended Euclid

Finds x,y such that d = ax + by.
Returns d = gcd(a,b).

```
int extended_euclid(int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
```

```
  while (b) {
    int q = a/b;
    int t = b; b = a%b; a = t;
    t = xx; xx = x-q*xx; x = t;
    t = yy; yy = y-q*yy; y = t;
  }
  return a;
}
```

3.4 Modular Linear Equation Solver

Finds all solutions to ax = b (mod n)

```
vi modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  vi solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    FOR(i, 0, d)
      solutions.pb(mod(x + i*(n/d), n));
  }
  return solutions;
}
```

3.5 Modular Inverse

Computes b such that ab = 1 (mod n), returns -1 on failure

```
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}
```

3.6 Chinese Remainder Theorem

Returns

$$x = a_i (mod n_i)$$

n's must be pairwise coprimes

```
int chinese_remainder(int *n, int *a, int len) {
  int p, i, prod = 1, sum = 0;
  for (i = 0; i < len; i++) prod *= n[i];
  for (i = 0; i < len; i++) {
    p = prod / n[i];
    sum += a[i] * mod_inverse(p, n[i]) * p;
  }
  return sum % prod;
}
```

3.7 Miller-Rabin Primality Test

$$O(log(N)^3)$$

```
ll mulmod(ll a, ll b, ll c) {
  ll x = 0, y = a % c;
  while (b) {
    if (b & 1) x = (x + y) % c;
    y = (y << 1) % c;
    b >>= 1;
  }
  return x % c;
}

ll fastPow(ll x, ll n, ll MOD) {
  ll ret = 1;
  while (n) {
    if (n & 1) ret = mulmod(ret, x, MOD);
    x = mulmod(x, x, MOD);
    n >>= 1;
  }
  return ret;
}

bool isPrime(ll n) {
  ll d = n - 1;
  int s = 0;
  while (d % 2 == 0) {
    s++;
    d >>= 1;
  }
  // It's garanteed that these values will work for any number smaller than
      3*10**18 (3 and 18 zeros)
  int a[9] = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
  FOR(i, 0, 9) {
    bool comp = fastPow(a[i], d, n) != 1;
    if(comp) FOR(j, 0, s) {
```

```
      ll fp = fastPow(a[i], (1LL << (ll)j)*d, n);
      if (fp == n - 1) {
        comp = false;
        break;
      }
    }
    if(comp) return false;
  }
  return true;
}
```

3.8 isPrime

$$O(sqrt(N))$$

```
bool isPrime(int n) {
  if(n < 2) return false;
  if(n == 2 || n == 3) return true;
  if(!(n&1 && n%3)) return false;
  long long sqrtN = sqrt(n)+1;
  for(long long i = 6LL; i <= sqrtN; i += 6)
    if(!(n%(i-1)) || !(n%(i+1))) return false;
  return true;
}
```

3.9 GCD

```
int gcd(int a, int b) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}
```

4 Strings

4.1 Suffix Array

$$O(Nlog(N))$$

Finds a permutation of the suffixes of S where suffix(i) ¡ suffix(j) for all i less than —S—-2
Example for finding the frequence and length of all distinct substrings:

```
int main() {
  char S[7] = "ababc$";
  int n = strlen(S);
  buildSA(S, n);
  buildLCP(S, n);

  FOR(i, 0, n)
    cout << i << "␣" << LCP[i] << "␣" << S+SA[i] << endl;

  FOR(i, 1, n) {
    if(LCP[i]) {
      int l = i-1;
      while(LCP[l] >= LCP[i]) l--;
      int j = l;
      while(j<=i || (j<n && LCP[j] >= LCP[i])) j++;
      int freq = j-l;
      int len = LCP[i];
      int startIndex = SA[i];
    }
  }
}
```

```
#define MAX_N 100010

int RA[MAX_N], SA[MAX_N], LCP[MAX_N];

void countingSort(int k, char S[], int n) {
  vi c(max(int(300), n), 0), tempSA(n);
  int sum = 0, maxi = max(int(300), n);
  FOR(i, 0, n) c[i+k<n ? RA[i+k]:0]++;
  FOR(i, 0, maxi) {
    sum += c[i];
    c[i] = sum - c[i];
  }
  FOR(i, 0, n)
    tempSA[c[SA[i]+k<n?RA[SA[i]+k]:0]++] = SA[i];
  FOR(i, 0, n)
    SA[i] = tempSA[i];
}

//S must end with a <=47 char
//FOR(i, 0, n)
//   cout << S+SA[i] << ": " << LCP[i] << endl;
void buildSA(char S[], int n) {
  vi tempRA(n);
  FOR(i, 0, n)
    RA[i] = S[i], SA[i] = i;
  for(int k=1, r=0; k<n; k<<=1) {
    countingSort(k, S, n);
    countingSort(0, S, n);
```

```
    tempRA[SA[0]] = r = 0;
    FOR(i, 1, n)
      tempRA[SA[i]] = (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k
          ]) ? r : ++r;
    FOR(i, 0, n)
      RA[i] = tempRA[i];
    if(RA[SA[n-1]] == n-1) break;
  }
}

ii findPattern(char S[], int n, char P[], int m) {
  int lo = 0, hi = n-1, mid;
  while(lo < hi) {
    mid = (lo + hi) / 2;
    if(strncmp(S+SA[mid], P, m) >= 0) hi = mid;
    else lo = mid+1;
  }
  if(strncmp(S+SA[lo], P, m) != 0) return ii(-1, -1);
  ii bounds; bounds.first = lo;
  lo = 0; hi = n-1; mid = lo;
  while(lo < hi) {
    mid = (lo + hi)/2;
    if(strncmp(S+SA[mid], P, m) > 0) hi = mid;
    else lo = mid+1;
  }
  if(strncmp(S+SA[hi], P, m) != 0) hi--;
  bounds.second = hi;
  return bounds;
}
```

4.2 Longest Common Prefix

$$O(n)$$

SA contains the suffix array for S
LCP[i] = longest common prefix between SA[i] and SA[i-1], LCP[0] = 0

```
void buildLCP(char S[], int n) {
  vi phi(n), plcp(n);
  int L = 0;
  phi[SA[0]] = -1;
  FOR(i, 1, n)
    phi[SA[i]] = SA[i-1];
  FOR(i, 0, n) {
    if(phi[i] == -1) { plcp[i] = 0; continue; }
    while(S[i+L] == S[phi[i]+L]) L++;
    plcp[i] = L;
```

```
    L = max(L-1, int(0));
  }
  FOR(i, 0, n) LCP[i] = plcp[SA[i]];
}
```

4.3 Linear Suffix Array

$$O(n)$$

Construct a suffic array in linear time
Example usage:

```
int main() { _
  int N = 6;
  int SA[6];
  char s[6] = "abcab";
  SA_IS((unsigned char*)s, SA, N, 256);
  FOR(i, 0, N)
    cout << s + SA[i] << endl;
}
```

```
#include <functional>
#include <stdlib.h>
#include <unistd.h>

unsigned char mask[] = { 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 };
#define tget(i) ( (t[(i)/8]&mask[(i)%8]) ? 1 : 0 )
#define tset(i, b) t[(i)/8]=(b) ? (mask[(i)%8]|t[(i)/8]) : ((~mask[(i)%8])&t[(i
    )/8])
#define chr(i) (cs==sizeof(int)?((int*)s)[i]:((unsigned char *)s)[i])
#define isLMS(i) (i>0 && tget(i) && !tget(i-1))

// find the start or end of each bucket
void getBuckets(unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
    int i, sum = 0;
    for (i = 0; i <= K; i++)
        bkt[i] = 0; // clear all buckets
    for (i = 0; i < n; i++)
        bkt[chr(i)]++; // compute the size of each bucket
    for (i = 0; i <= K; i++) {
        sum += bkt[i];
        bkt[i] = end ? sum : sum - bkt[i];
    }
}
// compute SAl
void induceSAl(unsigned char *t, int *SA, unsigned char *s, int *bkt, int n,
    int K, int cs, bool end) {
```

```
    int i, j;
    getBuckets(s, bkt, n, K, cs, end); // find starts of buckets
    for (i = 0; i < n; i++) {
        j = SA[i] - 1;
        if (j >= 0 && !tget(j))
            SA[bkt[chr(j)]++] = j;
    }
}
// compute SAs
void induceSAs(unsigned char *t, int *SA, unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
    int i, j;
    getBuckets(s, bkt, n, K, cs, end); // find ends of buckets
    for (i = n - 1; i >= 0; i--) {
        j = SA[i] - 1;
        if (j >= 0 && tget(j))
            SA[--bkt[chr(j)]] = j;
    }
}

// find the suffix array SA of s[0..n-1] in {1..K}^n
// require s[n-1]=0 (the sentinel!), n>=2
// use a working space (excluding s and SA) of at most 2.25n+O(1) for a constant alphabet
void SA_IS(unsigned char *s, int *SA, int n, int K, int cs = 1) {
    int i, j;
    unsigned char *t = (unsigned char *) malloc(n / 8 + 1); // LS-type array in bits
    // Classify the type of each character
    tset(n-2, 0);
    tset(n-1, 1); // the sentinel must be in s1, important!!!
    for (i = n - 3; i >= 0; i--)
        tset(i, (chr(i)<chr(i+1) || (chr(i)==chr(i+1) && tget(i+1)==1))?1:0);
    // stage 1: reduce the problem by at least 1/2
    // sort all the S-substrings
    int *bkt = (int *) malloc(sizeof(int) * (K + 1)); // bucket array
    getBuckets(s, bkt, n, K, cs, true); // find ends of buckets
    for (i = 0; i < n; i++)
        SA[i] = -1;
    for (i = 1; i < n; i++)
        if (isLMS(i))
            SA[--bkt[chr(i)]] = i;
    induceSAl(t, SA, s, bkt, n, K, cs, false);
    induceSAs(t, SA, s, bkt, n, K, cs, true);
    free(bkt);
    // compact all the sorted substrings into the first n1 items of SA
    // 2*n1 must be not larger than n (proveable)
    int n1 = 0;
    for (i = 0; i < n; i++)
        if (isLMS(SA[i]))
```

```
            SA[n1++] = SA[i];
    // find the lexicographic names of all substrings
    for (i = n1; i < n; i++)
        SA[i] = -1; // init the name array buffer
    int name = 0, prev = -1;
    for (i = 0; i < n1; i++) {
        int pos = SA[i];
        bool diff = false;
        for (int d = 0; d < n; d++)
            if (prev == -1 || chr(pos+d) != chr(prev+d) || tget(pos+d) != tget(
               prev+d)) {
                diff = true;
                break;
            } else if (d > 0 && (isLMS(pos+d) || isLMS(prev+d)))
                break;
        if (diff) {
            name++;
            prev = pos;
        }
        pos = (pos % 2 == 0) ? pos / 2 : (pos - 1) / 2;
        SA[n1 + pos] = name - 1;
    }
    for (i = n - 1, j = n - 1; i >= n1; i--)
        if (SA[i] >= 0)
            SA[j--] = SA[i];
    // stage 2: solve the reduced problem
    // recurse if names are not yet unique
    int *SA1 = SA, *s1 = SA + n - n1;
    if (name < n1)
        SA_IS((unsigned char*) s1, SA1, n1, name - 1, sizeof(int));
    else
        // generate the suffix array of s1 directly
        for (i = 0; i < n1; i++)
            SA1[s1[i]] = i;
    // stage 3: induce the result for the original problem
    bkt = (int *) malloc(sizeof(int) * (K + 1)); // bucket array
    // put all left-most S characters into their buckets
    getBuckets(s, bkt, n, K, cs, true); // find ends of buckets
    for (i = 1, j = 0; i < n; i++)
        if (isLMS(i))
            s1[j++] = i; // get p1
    for (i = 0; i < n1; i++)
        SA1[i] = s1[SA1[i]]; // get index in s
    for (i = n1; i < n; i++)
        SA[i] = -1; // init SA[n1..n-1]
    for (i = n1 - 1; i >= 0; i--) {
        j = SA[i];
        SA[i] = -1;
        SA[--bkt[chr(j)]] = j;
    }
```

```
    induceSAl(t, SA, s, bkt, n, K, cs, false);
    induceSAs(t, SA, s, bkt, n, K, cs, true);
    free(bkt);
    free(t);
}
```

4.4 Trie

Constructs a tree for storing strings

```
#define ALPHABET_SIZE 52
int getIndex(char c) {
  if(c >= 'A' && c <= 'Z')
    return c-'A';
  return c-'a'+26;
}

struct Trie {
  int words, prefixes;
  Trie *edges[ALPHABET_SIZE];
  Trie() : words(0), prefixes(0) { FOR(i, 0, ALPHABET_SIZE) edges[i] = 0; }
  ~Trie(){ FOR(i, 0, ALPHABET_SIZE) if(edges[i]) delete edges[i]; }
  void insert(char *word, int pos = 0) {
    if(word[pos] == 0) {
      words++;
      return;
    }
    prefixes++;
    int index = getIndex(word[pos]);
    if(edges[index] == 0)
      edges[index] = new Trie;
    edges[index]->insert(word, pos+1);
  }
  int countWords(char *word, int pos = 0) {
    if(word[pos] == 0)
      return words;
    int index = getIndex(word[pos]);
    if(edges[index]==0)
      return 0;
    return edges[index]->countWords(word, pos+1);
  }
  int countPrefix(char *word, int pos = 0) {
    if(word[pos] == 0)
      return prefixes;
    int index = getIndex(word[pos]);
    if(edges[index] == 0)
      return 0;
    return edges[index]->countPrefix(word, pos+1);
  }
```

```
};
```

4.5 KMP

$$O(N + M)$$

Searches for a pattern in a string

```
vi buildTable(string& pattern) {
  vi table(pattern.length()+1);
  int i = 0, j = -1, m = pattern.length();
  table[0] = -1;
  while(i < m) {
    while(j >= 0 && pattern[i] != pattern[j]) j = table[j];
    i++, j++;
    table[i] = j;
  }
  return table;
}

vi find(string& text, string& pattern) {
  vi matches;
  int i = 0, j = 0, n = text.length(), m = pattern.length();
  vi table = buildTable(pattern);
  while(i < n) {
    while(j >= 0 && text[i] != pattern[j]) j = table[j];
    i++, j++;
    if(j == m) {
      matches.pb(i-j);
      j = table[j];
    }
  }
  return matches;
}
```

5 Data Structures

5.1 HeavyLightDecomposition

Constructs the heavy light decomposition of a tree.
Allows querying in a path of a tree.

```
struct HeavyLightDecomposition {
  vector<vi> lists;
  vi values, listIndex, posIndex, parent, treeSizes;
  vector<SparseTable> sts;
  LCA *lca;
```

```
HeavyLightDecomposition(Graph &g, vi values) : values(values) {
  lca = new LCA(g, 0);
  listIndex = posIndex = parent = treeSizes = vi(g.V, -1);
  getTreeSizes(g, 0);
  makeLists(g, 0, -1);
  FORC(lists, list) {
    vi v;
    FORC(*list, it) v.pb(values[*it]);
    sts.pb(SparseTable(v));
  }
}
~HeavyLightDecomposition() { delete lca; }
int getTreeSizes(Graph &g, int cv) {
  treeSizes[cv] = 1;
  FORC(g.edges[cv], edge) if(edge->to != parent[cv])
      parent[edge->to] = cv, treeSizes[cv] += getTreeSizes(g, edge->to);
  return treeSizes[cv];
}
void makeLists(Graph &g, int cv, int listNum) {
  if(listNum == -1)
    listNum = lists.size(), lists.pb(vi());
  listIndex[cv] = listNum;
  posIndex[cv] = lists[listNum].size();
  lists[listNum].pb(cv);
  int MAX = -1;
  FORC(g.edges[cv], edge) if(edge->to != parent[cv])
    if(MAX == -1 || treeSizes[edge->to] > treeSizes[MAX]) MAX = edge->to;
  FORC(g.edges[cv], edge) if(edge->to != parent[cv])
    makeLists(g, edge->to, edge->to == MAX ? listNum : -1);
}
int query(int from, int to) {
  int anc = lca->query(from, to), posLeft, posRight;
  int result = min(queryToAncestor(from, anc, posLeft), queryToAncestor(to,
      anc, posRight));
  if(posLeft < posRight) swap(posLeft, posRight);
  result = min(result, values[lists[listIndex[anc]][sts[listIndex[anc]].query
      (posIndex[anc], posRight)]]);
  if(posRight != posLeft)
    result = min(result, values[lists[listIndex[anc]][sts[listIndex[anc]].
        query(posRight+1, posLeft)]]);
  return result;
}
int queryToAncestor(int from, int anc, int &posInAncestorList) {
  int result = INF, left = from;
  while(listIndex[left] != listIndex[anc]) {
    result = min(result, values[lists[listIndex[left]][sts[listIndex[left]].
        query(0, posIndex[left])]]);
    left = parent[lists[listIndex[left]][0]];
  }
  posInAncestorList = posIndex[left];
```

```
    return result;
  }
};
```

5.2 KD-Tree

A straightforward, but probably sub-optimal KD-tree implmentation that's probably good enough for most things (current it's a 2D-tree)
- constructs from n points in $O(nlg^2n)$ time
- handles nearest-neighbor query in $O(lgn)$ if points are well distributed
- worst case for nearest-neighbor may be linear in pathological case

```

#include <limits>
#include <cstdlib>

// 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 kdnode
{
    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

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

    kdnode() : leaf(false), first(0), second(0) {}
    ~kdnode() { 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
{
    kdnode *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(kdnode *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;
```

```
}
```

5.3 Lowest Common Ancestor

$$O(N) construction$$

$$O(Nlog(N)) queries$$

Answers lowest common ancestor queries in a tree.
Can be modified to use a sparse table for $O(1)$ queries.

```
struct LCA {
  vi order, height, index, st;
  int minIndex(int i, int j) {
    return height[i] < height[j] ? i : j;
  }
  LCA(Graph &g, ll root) {
    index.assign(g.V, -1);
    dfs(g, root, 0);
    st.assign(height.size()*2, 0);
    FOR(i, 0, height.size())
      st[height.size() + i] = i;
    for(int i = height.size()-1; i; i--)
      st[i] = minIndex(st[i<<1], st[i<<1|1]);
  }
  void dfs(Graph &g, ll cv, ll h) {
    index[cv] = order.size();
    order.pb(cv), height.pb(h);
    FORC(g.edges[cv], edge)
      if(index[edge->to] == -1) {
        dfs(g, edge->to, height.back() + 1);
        order.pb(cv), height.pb(h);
      }
  }
  ll query(ll i, ll j) {
    int from = index[i], to = index[j];
    if (from > to) swap(from, to);
    int idx = from;
    for(int l = from + height.size(), r = to + height.size() + 1; l < r; l >>=
        1, r >>= 1) {
      if (l&1) idx = minIndex(idx, st[l++]);
      if (r&1) idx = minIndex(idx, st[--r]);
    }
    return order[idx];
  }
};
```

5.4 Sparse Table

$$O(N * log(N))\ construction$$

$$O(1)\ queries$$

Answers RMQ

```
struct SparseTable {
  vi A; vvi M;
  int log2(int n) { int i=0; while(n >>= 1) i++; return i; }
  SparseTable(vi arr) {
    int N = arr.size();
    A.assign(N, 0);
    M.assign(N, vi(log2(N)+1));
    int i, j;
    for(i=0; i<N; i++)
      M[i][0] = i, A[i] = arr[i];

    for(j=1; 1<<j <= N; j++)
      for(i=0; i + (1<<j) - 1 < N; i++)
        if(A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])
          M[i][j] = M[i][j - 1];
        else
          M[i][j] = M[i + (1 << (j - 1))][j - 1];
  }
  //returns the index of the minimum value
  int query(int i, int j) {
    if(i > j) swap(i, j);
    int k = log2(j-i+1);
    if(A[M[i][k]] < A[M[j-(1 << k)+1][k]])
      return M[i][k];
    return M[j-(1 << k)+1][k];
  }
};
```

6 SegmentTree

```
struct SegmentTree {
  vi t; int N;
  SegmentTree(vi &values) {
    N = values.size();
    t.assign(N<<1, 0);
    FOR(i, 0, N) t[i+N] = values[i];
    for(int i = N-1; i; --i) t[i] = combine(t[i<<1], t[i<<1|1]);
  }
  int combine(int a, int b) { return a+b; }
```

```
  void set(int index, int value) {
    t[index+N] = value;
    for(int i = (index+N)>>1; i; i >>= 1) t[i] = combine(t[i<<1], t[i<<1|1]);
  }
  int query(int from, int to) {
    int ansL = 0, ansR = 0;
    for(int l = N+from, r = N+to; l<r; l >>= 1, r >>= 1) {
      if (l&1) ansL = combine(ansL, t[l++]);
      if (r&1) ansR = combine(ansR, t[--r]);
    }
    return combine(ansL, ansR);
  }
};

struct LazySegmentTree {
  vi t, d; int N, h;
  LazySegmentTree(vi &values) {
    N = values.size();
    h = sizeof(int) * 8 - __builtin_clz(n);
    t.assign(N<<1, 0), d.assign(N, 0);
    FOR(i, 0, N) t[i+N] = values[i];
    build(i+N, N<<1);
  }
  void calc(int p, int k) {
    if (d[p] == 0) t[p] = t[p<<1] + t[p<<1|1];
    else t[p] = d[p] * k;
  }
  void apply(int p, int value, int k) {
    t[p] = value * k;
    if (p < n) d[p] = value;
  }
  void push(int l, int r) {
    int s = h, k = 1 << (h-1);
    for (l += n, r += n-1; s > 0; --s, k >>= 1)
      for (int i = l >> s; i <= r >> s; ++i) if (d[i]) {
        apply(i<<1, d[i], k);
        apply(i<<1|1, d[i], k);
        d[i] = 0;
      }
  }
  void build(int l, int r) {
    int k = 2;
    for (l += n, r += n-1; l; k <<= 1) {
      l >>= 1, r >>= 1;
      for (int i = r; i >= l; --i) calc(i, k);
    }
  }
  void modify(int l, int r, int value) {
    if (value == 0) return;
    push(l, l + 1); push(r - 1, r);
```

```
    int l0 = l, r0 = r, k = 1;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1, k <<= 1) {
      if (l&1) apply(l++, value, k);
      if (r&1) apply(--r, value, k);
    }
    build(l0, l0 + 1);
    build(r0 - 1, r0);
  }
  int query(int l, int r) {
    push(l, l + 1); push(r - 1, r);
    int res = 0;
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (l&1) res += t[l++];
      if (r&1) res += t[--r];
    }
    return res;
  }
};
```

7 Geometry

7.1 Point

```
const double PI = 2*asin(1);

bool eq(double a, double b) { return fabs(a-b) < EPS; }
bool les(double a, double b) { return !eq(a, b) && a < b; }
struct Point {
  double x, y, z;
  Point() : x(0), y(0), z(0) {}
  Point(double x, double y) : x(x), y(y), z(0) {}
  Point(double x, double y, double z) : x(x), y(y), z(z) {}
  bool operator <(const Point &p) const {
        return les(x, p.x) || (eq(x, p.x) && les(y, p.y)) || (eq(x, p.x) && eq(
            y, p.y) && les(z, p.z));
  }
    bool operator==(const Point &p) {
        return eq(x, p.x) && eq(y, p.y) && eq(z, p.z);
    }
};

double DEG_to_RAD(double deg) {
    return deg/180*2*asin(1);
}

double dist(Point p1, Point p2) {
  return sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2) + pow(p1.z-p2.z, 2)); }

Point rotate(Point p, double theta) {
```

```
  double rad = DEG_to_RAD(theta);
  return Point(p.x*cos(rad) - p.y*sin(rad),
         p.x*sin(rad) + p.y*cos(rad));
}

double ANG(double rad) { return rad*180/PI; }
double angulo(Point p) {
  double d = atan(double(p.y)/p.x);
  if(p.x < 0)
    d += PI;
  else if(p.y < 0)
    d += 2*PI;
  return ANG(d);
}
```

7.2 Vector

```
struct Vec {
  double x, y, z;
  Vec(double x, double y, double z) : x(x), y(y), z(z) {}
  Vec() : x(0), y(0), z(0) {}
  Vec(double x, double y) : x(x), y(y), z(0) {}
  Vec(Point a, Point b) : x(b.x-a.x), y(b.y-a.y), z(b.z-a.z) {}
};

Vec toVec(Point a, Point b){
  return Vec(a, b); }

Vec scale(Vec v, double s) {
  return Vec(v.x*s, v.y*s, v.z*s); }

Point translate(Point p, Vec v) {
  return Point(p.x+v.x, p.y+v.y, p.z+v.z); }

double dot(Vec a, Vec b) {
  return (a.x*b.x + a.y*b.y + a.z*b.z); }

double norm_sq(Vec v) {
  return v.x*v.x + v.y*v.y + v.z*v.z; }

//angle in radians
Vec rotate(Vec v, double angle) {
  Matrix rotation = CREATE(2, 2);
  rotation[0][0] = rotation[1][1] = cos(angle);
  rotation[1][0] = sin(angle);
  rotation[0][1] = -rotation[1][0];

  Matrix vec = CREATE(2, 1);
  vec[0][0] = v.x, vec[0][1] = v.y;

```

```
  Matrix res = multiply(rotation, vec);
  Vec result(res[0][0], res[0][1]);
  return result;
}

double cross (Vec a, Vec b) { return a.x*b.y - a.y*b.x; }

// returns true if r is on the left side of line pq
bool ccw(Point p, Point q, Point r){
  return cross(toVec(p, q), toVec(p, r)) > 0; }

bool collinear(Point p, Point q, Point r) {
  return abs(cross(toVec(p, q), toVec(p, r))) < EPS; }

double angle(Point a, Point o, Point b) { // returns angle aob in rad
  Vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
}
```

7.3 Triangle

```
struct Triangle {
  Point A, B, C;
  Triangle() {}
  Triangle(Point A, Point B, Point C) : A(A), B(B), C(C) {}
};

double perimeter(double a, double b, double c) { return a+b+c; }

// Heron's formula
double area(double a, double b, double c){
  double s = perimeter(a, b, c)*0.5;
  return sqrt(s*(s-a)*(s-b)*(s-c));
}

double area(const Triangle &T) {
    double ab = dist(T.A, T.B);
    double bc = dist(T.B, T.C);
    double ca = dist(T.C, T.A);
    return area(ab, bc, ca);
}

double rInCircle(double ab, double bc, double ca){
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }

double rInCircle(Point a, Point b, Point c) {
  return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

bool inCircle(Point p1, Point p2, Point p3, Point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
```

```
  if(abs(r) < EPS) return false;
  Line l1, l2;
  double ratio = dist(p1, p2) / dist(p1, p3);
  Point p = translate(p2, scale(toVec(p2, p3), ratio/(1+ratio)));
  l1 = Line(p1, p);
  ratio = dist(p2, p1) / dist(p2, p3);
  l2 = Line(p2, p);
  areIntersect(l1, l2, ctr);
  return true;
}

double rCircumCircle(double ab, double bc, double ca) { return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

Point circumcenter(const Triangle &T) {
    Point A = T.A, B = T.B, C = T.C;
  double D = 2*(A.x*(B.y - C.y) + B.x*(C.y - A.y) + C.x*(A.y - B.y));
  double AA = A.x*A.x + A.y*A.y, BB = B.x*B.x + B.y*B.y, CC = C.x*C.x + C.y*C.y;
  return Point((AA*(B.y - C.y) + BB*(C.y - A.y) + CC*(A.y - B.y)) / D, (AA*(C.x - B.x) + BB*(A.x - C.x) + CC*(B.x - A.x)) / D);
}
```

7.4 Lines

```
struct Line {
  double a, b, c;
  Line() : a(0), b(0), c(0) {}
  Line(Point p1, Point p2) {
    if(abs(p1.x-p2.x) < EPS) {
      a = 1.0; b = 0.0; c = -p1.x;
    } else {
      a = -(double)(p1.y-p2.y)/(p1.x-p2.x);
      b = 1.0;
      c = -(double)(a*p1.x)-p1.y;
    }
  }
};

bool areParallel(Line l1, Line l2) {
  return (abs(l1.a-l2.a) < EPS) && (abs(l1.b-l2.b) < EPS); }

bool areSame(Line l1, Line l2) {
  return areParallel(l1, l2) && (abs(l1.c-l2.c) < EPS); }

bool areIntersect(Line l1, Line l2, Point &p) {
  if (areParallel(l1, l2)) return false;
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  if (abs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                 p.y = -(l2.a * p.x + l2.c);
```

```
  return true;
}

// Interseccion de AB con CD
// * WARNING: Does not work for collinear line segments!
bool lineSegIntersect(Point a, Point b, Point c, Point d) {
  double ucrossv1 = cross(toVec(a, b), toVec(a, c));
  double ucrossv2 = cross(toVec(a, b), toVec(a, d));
  if (ucrossv1 * ucrossv2 > 0) return false;
  double vcrossu1 = cross(toVec(c, d), toVec(c, a));
  double vcrossu2 = cross(toVec(c, d), toVec(c, b));
  return (vcrossu1 * vcrossu2 <= 0);
}

// Calcula la distancia de un punto P a una recta AB, y guarda en C la inters
double distToLine(Point p, Point a, Point b, Point &c) {
  Vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));
  return dist(p, c);
}

// Distancia a de P a segmento AB
double distToLineSegment(Point p, Point a, Point b, Point &c) {
  Vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = a; return dist(p, a); }
  if (u > 1.0) { c = b; return dist(p, b); }
  return distToLine(p, a, b, c);
}
```

7.5 Circles

```
bool circle2PtsRad(Point p1, Point p2, double r, Point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
  return true;
} // to get the other center, reverse p1 and p2
```

7.6 Polygons

```
typedef vector<Point> Polygon;

ll cross(const Point &O, const Point &A, const Point &B) {
  return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}
```

```
Polygon convexHull(Polygon &P) {
  int n = P.size(), k = 0;
  Polygon H(2*n);
  sort(P.begin(), P.end());
  FOR(i, 0, n) {
    while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
    H[k++] = P[i];
  }
  for (int i = n-2, t = k+1; i >= 0; i--) {
    while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
    H[k++] = P[i];
  }
  H.resize(k);
  return H;
}

// return area when Points are in cw or ccw, p[0]  = p[n-1]
double area(const Polygon &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1*y2-x2*y1);
  }
  return abs(result) / 2.0;
}

bool isConvex(const Polygon &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false;
  bool isLeft = ccw(P[0], P[1], P[2]);
  for (int i = 1; i < sz-1; i++)
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;
  return true;
}

// works for convex and concave
bool inPolygon (Point pt, const Polygon &P) {
  if((int)P.size() == 0) return false;
  double sum = 0;
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
      sum += angle(P[i], pt, P[i+1]);
    else sum -= angle(P[i], pt, P[i+1]); }
  return abs(abs(sum) - 2*PI) < EPS;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
```

```
bool isSimple(const Polygon &p) {
  for (int i = 0; i < p.size(); i++) {
  for (int k = i+1; k < p.size(); k++) {
    int j = (i+1) % p.size();
    int l = (k+1) % p.size();
    if (i == l || j == k) continue;
    if (lineSegIntersect(p[i], p[j], p[k], p[l]))
    return false;
  }
  }
  return true;
}

Point lineIntersectSeg(Point p, Point q, Point A, Point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x*A.y - A.x*B.y;
  double u = abs(a*p.x + b*p.y + c);
  double v = abs(a*q.x + b*q.y + c);
  return Point((p.x*v + q.x*u) / (u+v), (p.y*v + q.y*u) / (u+v));
}

// cuts polygon Q along line AB
Polygon cutPolygon(Point a, Point b, const Polygon &Q) {
  Polygon P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i+1])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.pb(Q[i]);
    if (left1 * left2 < -EPS)
      P.pb(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.pb(P.front());
  return P;
}

// only works for convex
bool pointInPolygon(Polygon &p1, Point p) {
  FOR(i, 0, p1.size() - 1)
    if (cross(p1[i], p1[i+1], p) >= 0)
      return false;
  return true;
}

// polygons must be convex
// returns polygon with size < 3 if there is no intersection
Polygon intersection(Polygon &p1, Polygon &p2) {
  set<Point> result;
  FOR(i, 0, p1.size() - 1) {
```

```
    if (pointInPolygon(p2, p1[i]))
      result.insert(p1[i]);
    FOR(j, 0, p2.size() - 1) {
      Line l1 = Line(p1[i], p1[i+1]);
      Line l2 = Line(p2[j], p2[j+1]);
      vector<Point> ps1, ps2;
      ps1.pb(p1[i]); ps1.pb(p1[i+1]);
      ps2.pb(p2[j]); ps2.pb(p2[j+1]);
      sort(ps1.begin(), ps1.end());
      sort(ps2.begin(), ps2.end());
      if (!areParallel(l1, l2)) {
        Point intersect;
        bool b = areIntersect(l1, l2, intersect);
        if (b && checkPointInSegm(intersect, ps1[0], ps1[1]) &&
            checkPointInSegm(intersect, ps2[0], ps2[1]))
          result.insert(intersect);
      } else if (areSame(l1, l2)) {
        if (ps1[1] >= ps2[0] && ps2[1] >= ps1[0]) {
          vector<Point> ps3;
          ps3.pb(ps1[0]); ps3.pb(ps1[1]); ps3.pb(ps2[0]); ps3.pb(ps2[1]);
          sort(all(ps3));
          result.insert(ps3[1]);
          result.insert(ps3[2]);
        }
      }
    }
  }

  FOR(i, 0, p2.size() - 1) {
    if (pointInPolygon(p1, p2[i]))
      result.insert(p2[i]);
  }

  if (result.size() <= 2) {
    return Polygon(result.begin(), result.end());
  }

  Polygon p(result.begin(), result.end());
  return convexHull(p);
}
```

7.7 Delaunay Triangulation

$$O(N^4)$$

Delaunay triangulation for a set P of points in a plane is a triangulation DT(P) such that no point in P is inside the circumcircle of any triangle in DT(P).
Slow but simple Delaunay triangulation.
INPUT: x[] = x-coordinates

y[] = y-coordinates
OUTPUT: triples = a vector containing m triples of indices corresponding to triangle vertices

```
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;
}
```

8 Miscellaneous

8.1 Fast Fourier Transform

in: input array
out: output array
step: SET TO 1 (used internally)
size: length of the input/output MUST BE A POWER OF 2
dir: either plus or minus one (direction of the FFT)
RESULT: out[k] =

$$\sum_{j=0}^{size-1} in[j] * exp(dir * 2pi * i * j * k/size)$$

Usage:
f[0...N-1] and g[0..N-1] are numbers
Want to compute the convolution h, defined by
h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
Let F[0...N-1] be FFT(f), and similarly, define G and H.
The convolution theorem says H[n] = F[n]G[n] (element-wise product).
To compute h[] in $O(NlogN)$ time, do the following:
1. Compute F and G (pass dir = 1 as the argument).
2. Get H by element-wise multiplying F and G.
3. Get h by taking the inverse FFT (use dir = -1 as the argument)
and *dividing by N*. DO NOT FORGET THIS SCALING FACTOR.

```
#include <cassert>

struct cpx
{
  cpx(){}
  cpx(double aa):a(aa),b(0){}
  cpx(double aa, double bb):a(aa),b(bb){}
  double a;
  double b;
  double modsq(void) const
  {
    return a * a + b * b;
```

```
  }
  cpx bar(void) const
  {
    return cpx(a, -b);
  }
};

cpx operator +(cpx a, cpx b)
{
  return cpx(a.a + b.a, a.b + b.b);
}

cpx operator *(cpx a, cpx b)
{
  return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}

cpx operator /(cpx a, cpx b)
{
  cpx r = a * b.bar();
  return cpx(r.a / b.modsq(), r.b / b.modsq());
}

cpx EXP(double theta)
{
  return cpx(cos(theta),sin(theta));
}

const double two_pi = 4 * acos(0);

void FFT(cpx *in, cpx *out, int step, int size, int dir)
{
  if(size < 1) return;
  if(size == 1)
  {
    out[0] = in[0];
    return;
  }
  FFT(in, out, step * 2, size / 2, dir);
  FFT(in + step, out + size / 2, step * 2, size / 2, dir);
  for(int i = 0 ; i < size / 2 ; i++)
  {
    cpx even = out[i];
    cpx odd = out[i + size / 2];
    out[i] = even + EXP(dir * two_pi * i / size) * odd;
    out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) / size) * odd;
  }
}
```

8.2 LatLong

Converts from rectangular coordinates to latitude/longitude and vice versa. Uses degrees (not radians).

```
struct ll
{
  double r, lat, lon;
};

struct rect
{
  double x, y, z;
};

ll convert(rect& P)
{
  ll Q;
  Q.r = sqrt(P.x*P.x+P.y*P.y+P.z*P.z);
  Q.lat = 180/M_PI*asin(P.z/Q.r);
  Q.lon = 180/M_PI*acos(P.x/sqrt(P.x*P.x+P.y*P.y));

  return Q;
}

rect convert(ll& Q)
{
  rect P;
  P.x = Q.r*cos(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.y = Q.r*sin(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
  P.z = Q.r*sin(Q.lat*M_PI/180);

  return P;
}

int main()
{
  rect A;
  ll B;

  A.x = -1.0; A.y = 2.0; A.z = -3.0;

  B = convert(A);
  cout << B.r << "␣" << B.lat << "␣" << B.lon << endl;

  A = convert(B);
  cout << A.x << "␣" << A.y << "␣" << A.z << endl;
}
```

8.3 Matrices

```
typedef vector<vector<double> > Matrix;
#define EPS 1E-7
#define CREATE(R, C) Matrix(R, vector<double>(C));

Matrix identity(int n) {
  Matrix m = CREATE(n, n);
  FOR(i, 0, n)
    m[i][i] = 1;
  return m;
}

Matrix multiply(Matrix m, double k) {
  FOR(i, 0, m.size())
    FOR(j, 0, m[0].size())
      m[i][j] *= k;
  return m;
}

Matrix multiply(Matrix m1, Matrix m2) {
  Matrix result = CREATE(m1.size(), m2[0].size());
  if(m1[0].size() != m2.size())
    return result;
  FOR(i, 0, result.size())
    FOR(j, 0, result[0].size())
      FOR(k, 0, m1[0].size())
        result[i][j] += m1[i][k]*m2[k][j];
  return result;
}

Matrix pow(Matrix m, int exp) {
  if(!exp) return identity(m.size());
  if(exp == 1) return m;
  Matrix result = identity(m.size());
  while(exp) {
    if(exp & 1) result = multiply(result, m);
    m = multiply(m, m);
    exp >>= 1;
  }
  return result;
}

//solves AX=B, output: A^-1 in A, X in B, returns det(A)
double gaussJordan(Matrix &a, Matrix &b) {
  int n = a.size(), m = b[0].size();
  vi irow(n), icol(n), ipiv(n);
  double det = 1;
  FOR(i, 0, n) {
    int pj = -1, pk = -1;
```

```
    FOR(j, 0, n) if (!ipiv[j])
      FOR(k, 0, n) if (!ipiv[k])
        if (pj == -1 || abs(a[j][k]) > abs(a[pj][pk])) { pj = j; pk = k; }
    if (abs(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;

    double c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    FOR(p, 0, n) a[pk][p] *= c;
    FOR(p, 0, m) b[pk][p] *= c;
    FOR(p, 0, n) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      FOR(q, 0, n) a[p][q] -= a[pk][q] * c;
      FOR(q, 0, m) b[p][q] -= b[pk][q] * c;
    }
  }
  for(int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    FOR(k, 0, n) swap(a[k][irow[p]], a[k][icol[p]]);
  }
  return det;
}

//returns the rank of a
int rref(Matrix &a) {
  int n = a.size(), m = a[0].size();
  int r = 0;
  FOR(c, 0, m) {
    int j = r;
    FOR(i, r+1, n)
      if (abs(a[i][c]) > abs(a[j][c])) j = i;
    if (abs(a[j][c]) < EPS) continue;
    swap(a[j], a[r]);
    double s = 1.0 / a[r][c];
    FOR(j, 0, m) a[r][j] *= s;
    FOR(i, 0, n) if (i != r) {
      double t = a[i][c];
      FOR(j, 0, m) a[i][j] -= t * a[r][j];
    }
    r++;
  }
  return r;
}
```

8.4 Dates

```
int toJulian(int day, int month, int year) {
  return 1461 * (year + 4800 + (month - 14) / 12) / 4 + 367 * (month - 2 -
    (month - 14) / 12 * 12) / 12 - 3 * ((year + 4900 + (month - 14) / 12)
    / 100) / 4 + day - 32075;
}

void toGregorian(int julian, int &day, int &month, int &year) {
  int x, n, i, j;
  x = julian + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  day = x - 2447 * j / 80;
  x = j / 11;
  month = j + 2 - 12 * x;
  year = 100 * (n - 49) + i + x;
}

bool isLeap(int year) { return (year%4 == 0 && year%100 != 0) || year%400 == 0; }
```

8.5 Nth Permutation

seq must be sorted

```
string nthPermutation(string seq, int permNum) {
  if(!seq.length()) return "";
  int f = fact(seq.length() - 1);
  int q = permNum/f, r = permNum%f;
  return seq[q] + nthPermutation(seq.substr(0, q) + seq.substr(q+1), r);
}
```

8.6 Shunting Yard

For parsing mathematical expressions specified in infix notation

```
void output(ostream &out, string x) {
    out << x << "␣";
}
string readToken(istream &in) {
    string t; int c;
    while((c = in.peek()) != EOF) {
        if(isalpha(c) || isdigit(c)) t.pb((char)c), in.get();
        else if(t != "") return t;
        else {in.get(); if(!isspace(c)) {t.pb((char)c); return t;}}
    } return t;
```

```
}

#define LEFT 0
#define RIGHT 1
#define isOp(x) (prec.find(x) != prec.end())
void shunting(istream &in, ostream &out) {
    string token;
    stack<string> ops;
    map<string, int> prec;
    prec["^"] = 6;
    prec["*"] = prec["/"] = prec["%"] = 5;
    prec["+"] = prec["-"] = 4;
    map<string, int> assoc; // default 0
    assoc["^"] = RIGHT;
    while((token = readToken(in)) != "") {
        if(isOp(token)) {
            while(!ops.empty() && isOp(ops.top())
            && ((assoc[token] == LEFT && prec[token] <= prec[ops.top()])
                 || (assoc[token] == RIGHT && prec[token] < prec[ops.top()])))
                output(out, ops.top()), ops.pop();
            ops.push(token);
        } else if(token == "(") {
            ops.push(token);
        } else if(token == ")") {
            while(!ops.empty() && ops.top() != "(")
                output(out, ops.top()), ops.pop();
            // ops.empty() || ops.top() != "(" ====> MISMATCH
            ops.pop();
        } else // numbers vars
            output(out, token);
    }
    while(!ops.empty()) { // if ops.top() == ")"  || ops.top() == "(" =======> MISMATCH
        output(out, ops.top()), ops.pop();
    }
}
```

9 Formulas

9.1 Catalan Numbers

$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^{n} \frac{n+k}{k}, n \geq 0$

$C_0 = 1, C_{n+1} = \frac{2(2n+1)}{n+2} C_n$

9.2 Law of Cosines

$c^2 = a^2 + b^2 - 2ab * cos(C)$

9.3 Law of Sines

$\frac{a}{sin(A)} = \frac{b}{sin(B)}$

9.4 Newton Raphson

$x_{n+1} = x_n - \frac{f(x_0)}{f'(x_0)}$

9.5 Arithmetic Series

$\sum_{k=1}^{n}(a_1 + (k-1)d) = na_1 + \frac{1}{2}nd(n-1)$

9.6 Geometric Series

$\sum_{k=1}^{n} r^k = \frac{r(1-r^n)}{1-r}$

$\sum_{k=1}^{\infty} r^k = \frac{r}{1-r}, |r| < 1$

9.7 Simpson's Rule

$\int_a^b f(x)dx \approx \frac{b-a}{6}(f(a) + 4f(\frac{a+b}{2}) + f(b))$

9.8 Stirling's Approximation

$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$

$ln(n!) = n * ln(n) - n + \frac{ln(2n)}{2}$

9.9 Sum of Powers

$\sum_{k=1}^{n} k^2 = \frac{1}{6}n(n+1)(2n+1)$

$\sum_{k=1}^{n} k^3 = (\sum_{k=1}^{n} k)^2 = (\frac{1}{2}n(n+1))^2$

9.10 Fermat's little Theorem

$a^p \equiv a\ (mod\ p)$ where p is prime

$a^{p-1} \equiv 1\ (mod\ p)$ where p is prime and a is not divisible by p

9.11 Euler's Totient Function

$\phi(n) = n\Pi_{p|n}(1 - \frac{1}{p})$ where p is prime

9.12 Euler's Theorem

$a^{\phi(n)} \equiv 1 \ (mod \ n)$ where gcd(a, n) = 1

9.13 Convex Polygon Centroid

Given the polygon $P = A_1, A_2, ..., A_n$
let $a = (A_{k+1} - A_1)$, $k = 1, 2, ..., n-1$ (the edge vectors)
let $C = A_1 + \frac{1}{3}(a_k + a_{k+1})$, $k = 1, 2, ..., n-2$ (the centroids of the triangles)
let $w = \frac{1}{2}(a_k \times a_{k+1})$, $k = 1, 2, ..., n-2$ (the areas of the triangles)
$centroid = \frac{\sum_{k=1}^{n-2} w_k C_k}{\sum_{k=1}^{n-2} w_k} = A_1 + \frac{1}{3}\frac{\sum_{k=1}^{n-2}(a_k + a_{k+1})(a_k \times a_{k+1})}{\sum_{k=1}^{n-2}(a_k \times a_{k+1})}$

9.14 Regular Polyhedron Volume

$volume = L^3$

9.15 Kirchoff Theorem

Let D be the degree matrix of G
Let A be the adjaceny matrix of G
Let $Q = D - A$
Let Q' be the matrix resulting from deleting any row and any column from Q
The number of spanning trees in a graph is equal to the determinant of Q'

There are n^{n-2} spanning trees in a complete graph

There are $m^{n-1} * n^{m-1}$ spanning trees in complete a bipartite graph

9.16 Derangements

A derangement is a permutation of a set where all elements are in a different position than their original position

$der(n) = (n-1) * (der(n-1) + der(n-2)), der(0) = 1, der(1) = 0$

9.17 Planar Graph Faces

$F = E - V - 2$