Sharif University of Technology - Crockpot - Notebook

Contents

1 Geometry

1.1 Line intersection

```
point intersection(point a, point b, point c, point d)
{
        point ab = b - a;
        point cd = d - c;
        point ac = c - a;
        double alpha = cross(ac, cd) / cross(ab, cd);
        return a + alpha * ab;
}
```

1.2 Convex hull 3D

```
/*
  GETS:
  n->number of vertices
  you should use add_edge(u,v) and
  add pair of vertices as edges (vertices are 0..n-1)

  GIVES:
  output of edmonds() is the maximum matching in general graph
  match[i] is matched pair of i (-1 if there isn't a matched pair)

  O(nh)
*/

#include<bits/stdc++.h>
using namespace std;
typedef pair<int,int> pii;

struct point{
  int X,Y,Z;
  point(int x=0,int y=0,int z=0){
    X=x;
    Y=y;
    Z=z;
  }
  bool operator==(const point& rhs) const {
    return (rhs.X==this->X && rhs.Y==this->Y && rhs.Z==this->Z);
  }
  bool operator<(const point& rhs) const {
    return rhs.X > this->X || (rhs.X == this->X && rhs.Y > this->Y) ||
          (rhs.X==this->X && rhs.Y==this->Y && rhs.Z>this->Z);
  }
};

const int maxn=1000;
int n;
point P[maxn];
vector<point>ans;
queue<pii>Q;
set<pii>mark;

int cross2d(point p,point q){ return p.X*q.Y-p.Y*q.X;}
point operator -(point p,point q){ return point(p.X-q.X,p.Y-q.Y,p.Z-q.
    Z); }
int dot(point v,point u){ return u.X*v.X+u.Y*v.Y+u.Z*v.Z; }
point _cross(point u,point v){ return point(u.Y*v.Z-u.Z*v.Y,u.Z*v.X-u.
    X*v.Z,u.X*v.Y-u.Y*v.X); }
point cross(point o,point p,point q){ return _cross(p-o,q-o);}

point shift(point p) { return point(p.Y,p.Z,p.X);}
point norm(point p)
{
  if(p.Y<p.X || p.Z<p.X) p=shift(p);
  if(p.Y<p.X) p=shift(p);
  return p;
}

int main()
{
  cin>>n;
  int mn=0;
  for(int i=0;i<n;i++){
    cin>>P[i].X>>P[i].Y>>P[i].Z;
    if(P[i]<P[mn]) mn=i;
  }
  int nx=(mn==0);
  for(int i=0;i<n;i++)
    if(i!=mn && i!=nx && cross2d(P[nx]-P[mn],P[i]-P[mn])>0)
      nx=i;
```

```
  Q.push(pii(mn,nx));
  while(!Q.empty())
    {
      int v=Q.front().first,u=Q.front().second;
      Q.pop();
      if(mark.find(pii(v,u))!=mark.end()) continue;
      mark.insert(pii(v,u));
      int p=-1;
      for(int q=0;q<n;q++)
        if(q!=v && q!=u)
          if(p==-1 || dot(cross(P[v],P[u],P[p]),P[q]-P[v])<0)
            p=q;
      ans.push_back(norm(point(v,u,p)));
      Q.push(pii(p,u));
      Q.push(pii(v,p));
    }
  sort(ans.begin(),ans.end());
  ans.resize(unique(ans.begin(),ans.end())-ans.begin());
  for(int i=0;i<ans.size();i++)
    cout<<ans[i].X<<" "<<ans[i].Y<<" "<<ans[i].Z<<endl;
}
```

1.3 Number of integer points inside polygon

```
S = I + B / 2 - 1
```

1.4 Half plane

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef int T;
typedef long long T2;
typedef long long T4; // maybe int128_t

const int MAXLINES = 100 * 1000 + 10;
const int INF = 20 * 1000 * 1000;

typedef pair<T, T> point;
typedef pair<point, point> line;

// REPLACE ZERO WITH EPS FOR DOUBLE

point operator - (const point &a, const point &b)
{
        return point(a.first - b.first, a.second - b.second);
}


T2 cross(point a, point b)
{
        return ((T2)a.first * b.second - (T2)a.second * b.first);
}

bool cmp(line a, line b)
{
        bool aa = a.first < a.second;
        bool bb = b.first < b.second;
        if (aa == bb)
        {
                point v1 = a.second - a.first;
                point v2 = b.second - b.first;
                if (cross(v1, v2) == 0)
                        return cross(b.second - b.first, a.first - b.
                            first) > 0;
                else
                        return cross(v1, v2) > 0;
        }
        else
                return aa;
}

bool parallel(line a, line b)
{
        return cross(a.second - a.first, b.second - b.first) == 0;
}

pair<T2, T2> alpha(line a, line b)
{
        return pair<T2, T2>(cross(b.first - a.first, b.second - b.
            first),
                                                cross(a.second - a.
                                                    first, b.second -
                                                    b.first));
}

bool fcmp(T4 f1t, T4 f1b, T4 f2t, T4 f2b)
{
        if (f1b < 0)
        {
                f1t *= -1;
                f1b *= -1;
        }
        if (f2b < 0)
        {
                f2t *= -1;
                f2b *= -1;
        }
        return f1t * f2b < f2t * f1b; // check with eps
}

bool check(line a, line b, line c)
{
        bool crs = cross(c.second - c.first, a.second - a.first) > 0;
        pair<T2, T2> a1 = alpha(a, b);
        pair<T2, T2> a2 = alpha(a, c);
        bool alp = fcmp(a1.first, a1.second, a2.first, a2.second);
        return (crs ^ alp);
}

bool notin(line a, line b, line c) // is intersection of a and b in
    ccw direction of c?
{
        if (parallel(a, b))
                return false;
        if (parallel(a, c))
```

```
                return cross(c.second - c.first, a.first - c.first) <
                    0;
        if (parallel(b, c))
                return cross(c.second - c.first, b.first - c.first) <
                    0;
        return !(check(a, b, c) && check(b, a, c));
}

void print(vector<line> lines)
{
        cerr << "    " << endl;for (int i = 0; i < lines.size();
            i++)cerr << lines[i].first.first << " " <<
            lines[i].first.second << " -> " << lines[i].second.first
            << " " << lines[i].second.second << endl;cerr << "    "


 << endl<< endl;
}

line dq[MAXLINES];

vector<line> half_plane(vector<line> lines)
{
        lines.push_back(line(point(INF, -INF), point(INF, INF)));
        lines.push_back(line(point(-INF, INF), point(-INF, -INF)));
        lines.push_back(line(point(-INF, -INF), point(INF, -INF)));
        lines.push_back(line(point(INF, INF), point(-INF, INF)));
        sort(lines.begin(), lines.end(), cmp);
        int ptr = 0;
        for (int i = 0; i < lines.size(); i++)
                if (i > 0 &&
                        (lines[i - 1].first < lines[i - 1].second) ==
                            (lines[i].first < lines[i].second) &&
                        parallel(lines[i - 1], lines[i]))
                        continue;
                else
                        lines[ptr++] = lines[i];
        lines.resize(ptr);
        if (lines.size() < 2)
                return lines;
        //print(lines);
        int f = 0, e = 0;
        dq[e++] = lines[0];
        dq[e++] = lines[1];
        for (int i = 2; i < lines.size(); i++)
        {
                while (f < e - 1 && notin(dq[e - 2], dq[e - 1], lines[
                    i]))
                        e--;
                //print(vector<line>(dq + f, dq + e));
                if (e == f + 1)
                {
                        T2 crs = cross(dq[f].second - dq[f].first,
                            lines[i].second - lines[i].first) ;
                        if (crs < 0)
                                return vector<line>();
                        else if (crs == 0 && cross(lines[i].second -
                            lines[i].first, dq[f].second - lines[i].
                            first) < 0)
                                return vector<line>();
                }
                while (f < e - 1 && notin(dq[f], dq[f + 1], lines[i]))
                        f++;
                dq[e++] = lines[i];
        }
        while (f < e - 1 && notin(dq[e - 2], dq[e - 1], dq[f]))
                e--;
        while (f < e - 1 && notin(dq[f], dq[f + 1], dq[e - 1]))
                f++;
        vector<line> res;
        res.resize(e - f);
        for (int i = f; i < e; i++)
                res[i - f] = dq[i];
        return res;
}


int main()
{
        int n;
        cin >> n;
        vector<line> lines;
        for (int i = 0; i < n; i++)
        {
                int x1, y1, x2, y2;
                cin >> x1 >> y1 >> x2 >> y2;
                lines.push_back(line(point(x1, y1), point(x2, y2)));
        }
        lines = half_plane(lines);
        cout << lines.size() << endl;
        for (int i = 0; i < lines.size(); i++)
                cout << lines[i].first.first << " " << lines[i].first.
                    second << " " << lines[i].second.first << " " <<
                    lines[i].second.second << endl;
}
```

1.5 Is this point in circle of other 3 points?

```
#include <iostream>
#include <algorithm>

using namespace std;

typedef pair<int, int> point;

// returns positive if d is outside circle abc,
// positive if d is inside it and 0 if it's on border
int inCircle (point a, point b, point c, point d)
{
        int x[4][4] = {
                1, a.first, a.second, a.first * a.first + a.second * a
                    .second,
                1, b.first, b.second, b.first * b.first + b.second * b
                    .second,
                1, c.first, c.second, c.first * c.first + c.second * c
                    .second,
                1, d.first, d.second, d.first * d.first + d.second * d
                    .second
        };
        // you can replace the following with any faster way
        // of calculating determinant.
```

```
        int y[] = {0, 1, 2, 3};
        int ans = 0;
        do {
                int mul = 1;
                for (int i = 0; i < 4; i++)
                        for (int j = i + 1; j < 4; j++)
                                if (y[i] > y[j])
                                        mul *= -1;
                for (int i = 0; i < 4; i++)
                        mul *= x[i][y[i]];
                ans += mul;
        } while (next_permutation(y, y + 4));
        return ans;
}
```

1.6 Rotating Caliper

```
#include <iostream>
#include <algorithm>
#include <complex>
#include <vector>

using namespace std;

typedef pair<int, int> Point;
typedef pair<vector<Point>, vector<Point> > pvv;

int cross(Point a, Point b)
{
        return a.first * b.second - a.second * b.first;
}

int norm(Point a)
{
        return a.first * a.first + a.second * a.second;
}

Point operator - (Point a, Point b)
{
        return Point(a.first - b.first, a.second - b.second);
}

Point org;

bool cmp(Point a, Point b)
{
        a = a - org;
        b = b - org;
        if (cross(a, b) == 0)
                return norm(a) < norm(b);
        else
                return cross(a, b) > 0;
}

pvv convex_hull(vector<Point> v)
{
        org = v[0];
        for (int i = 0; i < v.size(); i++)
                org = min(org, v[i]);
        sort(v.begin(), v.end(), cmp);
        /*
        for (int i = 0; i < v.size(); i++)
                cout << v[i].first << ", " << v[i].second << endl;
        cout << endl;*/
        vector<Point> cv;
        cv.push_back(v[0]);
        cv.push_back(v[1]);
        for (int i = 2; i < v.size(); i++)
        {
                while (cv.size() >= 2 && cross(v[i] - cv[cv.size() -
                    2], cv[cv.size() - 1] - cv[cv.size() - 2]) > 0)
                        cv.pop_back();
                cv.push_back(v[i]);
        }
        vector<Point> uh, lh;
        int mn = 0, mx = 0;
        for (int i = 0; i < cv.size(); i++)
        {
                if (cv[i] < cv[mn])
                        mn = i;
                if (cv[i] > cv[mx])
                        mx = i;
        }
        for (int i = mn; i != mx; i = (i + 1) % cv.size())
                lh.push_back(cv[i]);
        lh.push_back(cv[mx]);
        for (int i = mx; i != mn; i = (i + 1) % cv.size())
                uh.push_back(cv[i]);
        uh.push_back(cv[mn]);
        reverse(uh.begin(), uh.end());
        reverse(lh.begin(), lh.end());
        return pvv(uh, lh);

}

int findMax(vector<Point> a, vector<Point> b)
{
        int p1 = 0, p2 = 0;
        int res = 0;
        while (p1 < a.size() && p2 < b.size())
        {
                //cerr << a[p1].first << " " << a[p1].second << "
                    ----- " << b[p2].first << " " << b[p2].second <<
                    endl;
                res = max(res, norm(b[p2] - a[p1]));
                if (p1 + 1 == a.size())
                        p2++;
                else if (p2 + 1 == b.size())
                        p1++;
                else
                {
                        Point v1, v2;
                        if (a[p1] < a[p1 + 1])
                                v1 = a[p1 + 1] - a[p1];
                        else
                                v1 = a[p1] - a[p1 + 1];

                        if (b[p2] <  b[p2 + 1])
                                v2 = b[p2 + 1] - b[p2];
                        else
                                v2 = b[p2] - b[p2 + 1];
```

```
                        //cerr << v1.first << " " << v1.second << "
                            ### " << v2.first << " " << v2.second <<
                            endl;
                        if (cross(v1, v2) > 0)
                                p2++;
                        else
                                p1++;
                }
        }
        return res;
}

vector<Point> v1, v2;

int main()
{
        int n;
        cin >> n;
        for (int i = 0; i < n; i++)
        {
                int x, y;
                cin >> x >> y;
                v1.push_back(Point(x, y));
        }
        sort(v1.begin(), v1.end());
        v1.resize(unique(v1.begin(), v1.end()) - v1.begin());
        int m;
        cin >> m;
        for (int i = 0; i < m; i++)
        {
                int x, y;
                cin >> x >> y;
                v2.push_back(Point(x, y));
        }
        sort(v2.begin(), v2.end());
        v2.resize(unique(v2.begin(), v2.end()) - v2.begin());


        pvv h1 = convex_hull(v1);
        pvv h2 = convex_hull(v2);
        cout << max(max(findMax(h1.first, h2.second), findMax(h1.
            second, h2.first)), max(findMax(h1.first, h2.first),
            findMax(h1.second, h2.second))) << endl;


}
```

2 Graph

2.1 Maximum matching - Edmond's blossom

```
/*
  GETS:
  n->number of vertices
  you should use add_edge(u,v) and
  add pair of vertices as edges (vertices are 0..n-1)
  (note: please don't add multiple edge)
  GIVES:
  output of edmonds() is the maximum matching in general graph
  match[i] is matched pair of i (-1 if there isn't a matched pair)

  O(mn^2)
*/

#include <bits/stdc++.h>
using namespace std;

struct struct_edge{int v;struct_edge* nxt;};
typedef struct_edge* edge;
const int MAXN=500;

struct Edmonds
{
  struct_edge pool[MAXN*MAXN*2];
  edge top=pool,adj[MAXN];
  int n,match[MAXN],qh,qt,q[MAXN],father[MAXN],base[MAXN];
  bool inq[MAXN],inb[MAXN];

  void add_edge(int u,int v)
  {
    top->v=v,top->nxt=adj[u],adj[u]=top++;
    top->v=u,top->nxt=adj[v],adj[v]=top++;
  }
  int LCA(int root,int u,int v)
  {
    static bool inp[MAXN];
    memset(inp,0,sizeof(inp));
    while(1)
      {
        inp[u=base[u]]=true;
        if (u==root) break;
        u=father[match[u]];
      }
    while(1)
      {
        if (inp[v=base[v]]) return v;
        else v=father[match[v]];
      }
  }

  void mark_blossom(int lca,int u)
  {
    while (base[u]!=lca)
      {
        int v=match[u];
        inb[base[u]]=inb[base[v]]=true;
        u=father[v];
        if (base[u]!=lca) father[u]=v;
      }
  }

  void blossom_contraction(int s,int u,int v)
  {
    int lca=LCA(s,u,v);
    memset(inb,0,sizeof(inb));
    mark_blossom(lca,u);
    mark_blossom(lca,v);
```

```
  if (base[u]!=lca)
    father[u]=v;
  if (base[v]!=lca)
    father[v]=u;
  for (int u=0;u<n;u++)
    if (inb[base[u]])
      {
        base[u]=lca;
        if (!inq[u])
          inq[q[++qt]=u]=true;
      }
}

int find_augmenting_path(int s)
{
  memset(inq,0,sizeof(inq));
  memset(father,-1,sizeof(father));
  for (int i=0;i<n;i++) base[i]=i;
  inq[q[qh=qt=0]=s]=true;
  while (qh<=qt)
    {
      int u=q[qh++];
      for (edge e=adj[u];e;e=e->nxt)
        {
          int v=e->v;
          if (base[u]!=base[v] && match[u]!=v)
            {
              if (v==s || (match[v]!=-1 && father[match[v]]!=-1))
                blossom_contraction(s,u,v);
              else if (father[v]==-1)
                {
                  father[v]=u;
                  if (match[v]==-1)
                    return v;
                  else if (!inq[match[v]])
                    inq[q[++qt]=match[v]]=true;
                }
            }
        }
    }
  return -1;
}

int augment_path(int s,int t)
{
  int u=t,v,w;
  while (u!=-1)
    {
      v=father[u];
      w=match[v];
      match[v]=u;
      match[u]=v;
      u=w;
    }
  return t!=-1;
}

int edmonds()
{
  int matchc=0;
  memset(match,-1,sizeof(match));
  for (int u=0;u<n;u++)
    if (match[u]==-1)
      matchc+=augment_path(u,find_augmenting_path(u));
  return matchc;
 }
};
```

2.2 Biconnected components

```
vector<int> adj[maxn];
bool vis[maxn];
int dep[maxn], par[maxn], lowlink[maxn];
vector<vector<int> > comp;
stack<int> st;
void dfs(int u, int depth = 0, int parent = -1)
{
        vis[u] = true;
        dep[u] = depth;
        par[u] = parent;
        lowlink[u] = depth;
        st.push(u);
        for (int i = 0; i < adj[u].size(); i++)
        {
                int v = adj[u][i];
                if (!vis[v])
                {
                        dfs(v, depth + 1, u);
                        lowlink[u] = min(lowlink[u], lowlink[v]);
                }
                else
                        lowlink[u] = min(lowlink[u], dep[v]);
        }
        if (lowlink[u] == dep[u] - 1)
        {
                comp.push_back(vector<int>());
                while (st.top() != u)
                {
                        comp.back().push_back(st.top());
                        st.pop();
                }
                comp.back().push_back(u);
                st.pop();
                comp.back().push_back(par[u]);
        }
}
void bicon(int n)
{
        for (int i = 0; i < n; i++)
                if (!vis[i])
                        dfs(i);
}
```

2.3 Gomory-hu

```
struct GomoryHu
{
        int par[MAXN], ans[MAXN][MAXN]; // SET MAXIMUM NUMBER OF NODES
        int edges[4 * MAXE]; // SET MAXIMUM NUMBER OF EDGES
```

```
        int ecnt;
        void clear()
        {
                ecnt = 0;
        }
        void add_edge(int u, int v, int uv, int vu = 0)
        {
                edges[ecnt++] = u;
                edges[ecnt++] = v;
                edges[ecnt++] = uv;
                edges[ecnt++] = vu;
        }
        Flow graph; // USE flow.cpp
        void build(int n)
        {
                for (int i = 0; i < n; i++)
                {
                        par[i] = 0;
                        for (int j = 0; j < n; j++)
                                ans[i][j] = 1e9; // SET YOUR INFINITY
                }
                for (int v = 1; v < n; v++)
                {
                        graph.clear();
                        for (int i = 0; i < ecnt; i += 4)
                                graph.add_edge(edges[i], edges[i + 1],
                                     edges[i + 2], edges[i + 3]);
                        int f = graph.max_flow(v, par[v]);
                        for (int u = v + 1; u < n; u++)
                                if (graph.d[u] != -1 && par[u] == par[
                                    v])
                                        par[u] = v;
                        ans[v][par[v]] = ans[par[v]][v] = f;
                        for (int u = 0; u < v; u++)
                                ans[u][v] = ans[v][u] = min(f, ans[par
                                    [v]][u]);
                }
        }
        GomoryHu()
        {
                clear();
        }
};
```

2.4 Directed minimum spanning tree (mlogn)

```
/*
  GETS:
  call make_graph(n) at first
  you should use add_edge(u,v,w) and
  add pair of vertices as edges (vertices are 0..n-1)

  GIVES:
  output of dmst(v) is the minimum arborescence with root v in
      directed graph
  (INF if it hasn't a spanning arborescence with root v)

  O(mlogn)
*/
#include <bits/stdc++.h>
using namespace std;

const int INF = 2e7;

struct MinimumAborescense
{
  struct edge {
    int src, dst, weight;
  };

  struct union_find {
    vector<int> p;
    union_find(int n) : p(n, -1) { };
    bool unite(int u, int v) {
      if ((u = root(u)) == (v = root(v))) return false;
      if (p[u] > p[v]) swap(u, v);
      p[u] += p[v]; p[v] = u;
      return true;
    }
    bool find(int u, int v) { return root(u) == root(v); }
    int root(int u) { return p[u] < 0 ? u : p[u] = root(p[u]); }
    int size(int u) { return -p[root(u)]; }
  };

  struct skew_heap {
    struct node {
      node *ch[2];
      edge key;
      int delta;
    } *root;
    skew_heap() : root(0) { }
    void propagate(node *a) {
      a->key.weight += a->delta;
      if (a->ch[0]) a->ch[0]->delta += a->delta;
      if (a->ch[1]) a->ch[1]->delta += a->delta;
      a->delta = 0;
    }
    node *merge(node *a, node *b) {
      if (!a || !b) return a ? a : b;
      propagate(a); propagate(b);
      if (a->key.weight > b->key.weight) swap(a, b);
      a->ch[1] = merge(b, a->ch[1]);
      swap(a->ch[0], a->ch[1]);
      return a;
    }
    void push(edge key) {
      node *n = new node();
      n->ch[0] = n->ch[1] = 0;
      n->key = key; n->delta = 0;
      root = merge(root, n);
    }
    void pop() {
      propagate(root);
      node *temp = root;
      root = merge(root->ch[0], root->ch[1]);
    }
    edge top() {
      propagate(root);
      return root->key;
    }
```

```
    bool empty() {
      return !root;
    }
    void add(int delta) {
      root->delta += delta;
    }
    void merge(skew_heap x) {
      root = merge(root, x.root);
    }
  };

  vector<edge> edges;
  void add_edge(int src, int dst, int weight) {
    edges.push_back({src, dst, weight});
  }
  int n;
  void make_graph(int _n) {
    n = _n;
    edges.clear();
  }

  int dmst(int r) {
    union_find uf(n);
    vector<skew_heap> heap(n);
    for (auto e: edges)
      heap[e.dst].push(e);

    double score = 0;
    vector<int> seen(n, -1);
    seen[r] = r;
    for (int s = 0; s < n; ++s) {
      vector<int> path;
      for (int u = s; seen[u] < 0;) {
        path.push_back(u);
        seen[u] = s;
        if (heap[u].empty()) return INF;

        edge min_e = heap[u].top();
        score += min_e.weight;
        heap[u].add(-min_e.weight);
        heap[u].pop();

        int v = uf.root(min_e.src);
        if (seen[v] == s) {
          skew_heap new_heap;
          while (1) {
            int w = path.back();
            path.pop_back();
            new_heap.merge(heap[w]);
            if (!uf.unite(v, w)) break;
          }
          heap[uf.root(v)] = new_heap;
          seen[uf.root(v)] = -1;
        }
        u = uf.root(v);
      }
    }
    return score;
  }
};
```

2.5 Directed minimum spanning tree (nm)

```
/*
  GETS:
  call make_graph(n) at first
  you should use add_edge(u,v,w) and
  add pair of vertices as edges (vertices are 0..n-1)

  GIVES:
  output of dmst(v) is the minimum arborescence with root v in
      directed graph
  (-1 if it hasn't a spanning arborescence with root v)

  O(mn)
*/

#include <bits/stdc++.h>
using namespace std;

const int INF = 2e7;

struct MinimumAborescense
{
  int n;
  struct edge {
    int src, dst;
    int weight;
  };
  vector<edge> edges;

  void make_graph(int _n) {
    n=_n;
    edges.clear();
  }

  void add_edge(int u, int v, int w) {
    edges.push_back({u, v, w});
  }

  int dmst(int r) {
    int N = n;
    for (int res = 0; ;) {
      vector<edge> in(N, {-1,-1,(int)INF});
      vector<int> C(N, -1);
      for (auto e: edges)
        if (in[e.dst].weight > e.weight)
          in[e.dst] = e;
      in[r] = {r, r, 0};

      for (int u = 0; u < N; ++u) { // no comming edge ==> no
          aborescense
        if (in[u].src < 0) return -1;
        res += in[u].weight;
      }
      vector<int> mark(N, -1); // contract cycles
      int index = 0;
      for (int i = 0; i < N; ++i) {
        if (mark[i] != -1) continue;
        int u = i;
```

```
      while (mark[u] == -1) {
        mark[u] = i;
        u = in[u].src;
      }
      if (mark[u] != i || u == r) continue;
      for (int v = in[u].src; u != v; v = in[v].src) C[v] = index;
      C[u] = index++;
    }
    if (index == 0) return res; // found arborescence
    for (int i = 0; i < N; ++i) // contract
      if (C[i] == -1) C[i] = index++;

    vector<edge> next;
    for (auto &e: edges)
      if (C[e.src] != C[e.dst] && C[e.dst] != C[r])
        next.push_back({C[e.src], C[e.dst], e.weight - in[e.dst].
            weight});
    edges.swap(next);
    N = index; r = C[r];
  }
 }
};
```

2.6 Dominator tree

```
struct DominatorTree
{
        vector<int> adj[MAXN], radj[MAXN], tree[MAXN], bucket[MAXN];
            // SET MAXIMUM NUMBER OF NODES
        int sdom[MAXN], par[MAXN], idom[MAXN], dsu[MAXN], label[MAXN];
        int arr[MAXN], rev[MAXN], cnt;
        void clear()
        {
                for (int i = 0; i < MAXN; i++)
                {
                        adj[i].clear();
                        radj[i].clear();
                        tree[i].clear();
                        sdom[i] = idom[i] = dsu[i] = label[i] = i;
                        arr[i] = -1;
                }
                cnt = 0;
        }
        void add_edge(int u, int v)
        {
                adj[u].push_back(v);
        }
        void dfs(int v)
        {
                arr[v] = cnt;
                rev[cnt] = v;
                cnt++;
                for (int i = 0; i < adj[v].size(); i++)
                {
                        int u = adj[v][i];
                        if (arr[u] == -1)
                        {
                                dfs(u);
                                par[arr[u]] = arr[v];
                        }
                        radj[arr[u]].push_back(arr[v]);
                }
        }
        int find(int v, int x = 0)
        {
                if (dsu[v] == v)
                        return (x ? -1 : v);
                int u = find(dsu[v], x + 1);
                if (u < 0)
                        return v;
                if (sdom[label[dsu[v]]] < sdom[label[v]])
                        label[v] = label[dsu[v]];
                dsu[v] = u;
                return (x ? u : label[v]);
        }
        void merge(int u, int v)
        {
                dsu[v] = u;
        }
        void build(int root)
        {
                dfs(root);
                int n = cnt;
                for (int v = n - 1; v >= 0; v--)
                {
                        for (int i = 0; i < radj[v].size(); i++)
                        {
                                int u = radj[v][i];
                                sdom[v] = min(sdom[v], sdom[find(u)]);
                        }
                        if (v > 0)
                                bucket[sdom[v]].push_back(v);
                        for (int i = 0; i < bucket[v].size(); i++)
                        {
                                int u = bucket[v][i];
                                int w = find(u);
                                if (sdom[u] == sdom[w])
                                        idom[u] = sdom[u];
                                else
                                        idom[u] = w;
                        }
                        if (v > 0)
                                merge(par[v], v);
                }
                for (int v = 1; v < n; v++)
                {
                        if (idom[v] != sdom[v])
                                idom[v] = idom[idom[v]];
                        tree[rev[v]].push_back(rev[idom[v]]);
                        tree[rev[idom[v]]].push_back(rev[v]);
                }
        }
        DominatorTree()
        {
                clear();
        }
};
```

2.7 Flow - Dinic

```
struct Flow
{
        int head[MAXN], q[MAXN], d[MAXN], ptr[MAXN]; // SET MAXIMUM
            NUMBER OF NODES
        int from[2 * MAXE], to[2 * MAXE], cap[2 * MAXE], prv[2 * MAXE
            ]; // SET MAXIMUM NUMBER OF EDGES
        int ecnt;
        void clear()
        {
                memset(head, -1, sizeof(head));
                ecnt = 0;
        }
        void add_edge(int u, int v, int uv, int vu = 0)
        {
                from[ecnt] = u, to[ecnt] = v, cap[ecnt] = uv, prv[ecnt
                    ] = head[u]; head[u] = ecnt++;
                from[ecnt] = v, to[ecnt] = u, cap[ecnt] = vu, prv[ecnt
                    ] = head[v]; head[v] = ecnt++;
        }
        bool bfs(int source, int sink)
        {
                int h = 0, t = 0;
                memset(d, -1, sizeof(d));
                d[source] = 0;
                q[t++] = source;
                while (h < t)
                {
                        int v = q[h++];
                        for (int i = head[v]; i != -1; i = prv[i])
                                if (cap[i] && d[to[i]] == -1)
                                {
                                        d[to[i]] = d[v] + 1;
                                        q[t++] = to[i];
                                }
                }
                return (d[sink] != -1);
        }
        int dfs(int v, int sink, int f = 1e9) // SET YOUR INFINITY
        {
                if (!f || v == sink)
                        return f;
                int ans = 0;
                for (int &i = ptr[v]; i != -1; i = prv[i])
                        if (d[to[i]] == d[v] + 1)
                        {
                                int x = dfs(to[i], sink, min(f, cap[i
                                    ]));
                                cap[i] -= x;
                                cap[i ^ 1] += x;
                                f -= x;
                                ans += x;
                                if (!f)
                                        break;
                        }
                return ans;
        }
        int max_flow(int source, int sink)
        {
                int f = 0;
                while (bfs(source, sink))
                {
                        int x;
                        memcpy(ptr, head, sizeof(head));
                        while (x = dfs(source, sink))
                                f += x;
                }
                return f;
        }
        Flow()
        {
                clear();
        }
};
```

2.8 Maximum weighted matching - Hungarian

```
/*
  GETS:
  n->number of vertices in each part
  cost[i][j]->weight of edge between i,j
  (vertices in each part are 0..n-1)

  GIVES:
  output of hungarian() is the maximum weighted matching
  xy[v] is matched pair of v if v is in X
  and yx[v] is matched pair of v if v is in Y
  (-1 if there isn't a matched pair)

  O(n^3)
*/

#include<bits/stdc++.h>
using namespace std;

const int MAXN = 505;
const int inf = 1e8;

struct Hungarian
{
  int cost[MAXN][MAXN];
  int n, max_match;
  int lx[MAXN], ly[MAXN];
  int xy[MAXN];
  int yx[MAXN];
  bool S[MAXN], T[MAXN];
  int slack[MAXN];
  int slackx[MAXN];
  int prev[MAXN];

  void init_labels()
  {
    memset(lx, 0, sizeof(lx));
    memset(ly, 0, sizeof(ly));
    for (int x = 0; x < n; x++)
      for (int y = 0; y < n; y++)
        lx[x] = max(lx[x], cost[x][y]);
  }

  void add_to_tree(int x, int prevx)
  {
    S[x] = true;
```

```
  prev[x] = prevx;
  for (int y = 0; y < n; y++)
    if (lx[x] + ly[y] - cost[x][y] < slack[y])
      {
        slack[y] = lx[x] + ly[y] - cost[x][y];
        slackx[y] = x;
      }
}

void update_labels()
{
  int x, y, delta = inf;
  for (y = 0; y < n; y++)
    if (!T[y])
      delta = min(delta, slack[y]);
  for (x = 0; x < n; x++)
    if (S[x]) lx[x] -= delta;
  for (y = 0; y < n; y++)
    if (T[y]) ly[y] += delta;
  for (y = 0; y < n; y++)
    if (!T[y])
      slack[y] -= delta;
}

void augment()
{
  if (max_match == n) return;
  int x, y, root;
  int q[MAXN], wr = 0, rd = 0;
  memset(S, false, sizeof(S));
  memset(T, false, sizeof(T));
  memset(prev, -1, sizeof(prev));
  for (x = 0; x < n; x++)
    if (xy[x] == -1)
      {
        q[wr++] = root = x;
        prev[x] = -2;
        S[x] = true;
        break;
      }
  for (y = 0; y < n; y++)
    {
      slack[y] = lx[root] + ly[y] - cost[root][y];
      slackx[y] = root;
    }


  while (true)
    {
      while (rd < wr)
        {
          x = q[rd++];
          for (y = 0; y < n; y++)
            if (cost[x][y] == lx[x] + ly[y] && !T[y])
              {
                if (yx[y] == -1) break;
                T[y] = true;
                q[wr++] = yx[y];
                add_to_tree(yx[y], x);
              }
          if (y < n) break;
        }
      if (y < n) break;

      update_labels();
      wr = rd = 0;
      for (y = 0; y < n; y++)
        if (!T[y] && slack[y] == 0)
          {
            if (yx[y] == -1)
              {
                x = slackx[y];
                break;
              }
            else
              {
                T[y] = true;
                if (!S[yx[y]])
                  {
                    q[wr++] = yx[y];
                    add_to_tree(yx[y], slackx[y]);
                  }
              }
          }
      if (y < n) break;
    }


  if (y < n)
    {
      max_match++;
      for (int cx = x, cy = y, ty; cx != -2; cx = prev[cx], cy = ty)
        {
          ty = xy[cx];
          yx[cy] = cx;
          xy[cx] = cy;
        }
      augment();
    }
}

int hungarian()
{
  int ret = 0;
  max_match = 0;
  memset(xy, -1, sizeof(xy));
  memset(yx, -1, sizeof(yx));
  init_labels();
  augment();
  for (int x = 0; x < n; x++)
    ret += cost[x][xy[x]];
  return ret;
}
};
```

2.9 Ear decomposition

```
1- Find a spanning tree of the given graph and choose a root for the
    tree.
2- Determine, for each edge uv that is not part of the tree, the
    distance between the root and the lowest common ancestor of u and
```

```
    v.
3- For each edge uv that is part of the tree, find the corresponding "
    master edge", a non-tree edge wx such that the cycle formed by
    adding wx to the tree passes through uv and such that, among such
    edges, w and x have a lowest common ancestor that is as close to
    the root as possible (with ties broken by edge identifiers).
4- Form an ear for each non-tree edge, consisting of it and the tree
    edges for which it is the master, and order the ears by their
    master edges' distance from the root (with the same tie-breaking
    rule).
```

3 Combinatorics

3.1 LP simplex

```cpp
#include <cmath>
#include <cstdio>
#include <memory.h>

const int MAXEQ = 310;
const int MAXVAR = 310;
const long double eps = 1e-8; // 1e-6?

struct Simplex {
        long double A[MAXEQ][MAXVAR+1];
        long double obj[MAXVAR+1];
        int ones[MAXEQ];

        long double saved[MAXVAR+1];
        long double sol[MAXVAR+1];

        int n_r, n_col;

        void mult_row(long double* row, long double k) {
                for (int j = 0; j <= n_col; j++)
                        row[j] *= k;
        }
        void add_row_mult(long double* row_b, long double* row_a, long
             double mult) {
                for (int j = 0; j <= n_col; j++)
                        row_b[j] += row_a[j]*mult;
        }
        void pivot(int r, int c) {
                mult_row(A[r], 1.0 / A[r][c]);
                ones[r] = c;
                for (int i = 0; i <= n_r; i++) {
                        if (i != r && A[i][c] != 0)
                                add_row_mult(A[i], A[r], -A[i][c]);
                }
                add_row_mult(obj, A[r], -obj[c]);
                add_row_mult(saved, A[r], -saved[c]);
        }
        void move_col(int c2, int c1) {
                for (int i = 0; i < n_r; i++)
                {
                        A[i][c2] = A[i][c1];
                        A[i][c1] = 0;
                }
                obj[c2] = obj[c1];
                obj[c1] = 0;
                saved[c2] = saved[c1];
                saved[c1] = 0;
        }

        long double solve_feasible() {
                while (true) {
                        int new_one = -1;
                        for (int j = 0; j < n_col; j++)
                                if (obj[j] < -eps)
                                {
                                        new_one = j;
                                        break;
                                }
                        if (new_one == -1)
                                break;

                        int row = -1;
                        long double lim = 1e100;
                        for (int i = 0; i < n_r; i++) {
                                if (A[i][new_one] > eps) {
                                        long double val = A[i][n_col]
                                            / A[i][new_one];
                                        if (val < lim) {
                                                lim = val;
                                                row = i;
                                        }
                                }
                        }
                        if (row == -1)
                                return -1e100; // unbounded // !!
                                    promjena
                        pivot(row, new_one);
                }
                memset(sol, 0, sizeof sol);
                for (int i = 0; i < n_r; i++) sol[ones[i]] = A[i][
                    n_col];
                return obj[n_col];
        }

        bool get_feasibile() {
                int min_row = -1;
                for (int i = 0; i < n_r; i++)
                        if (min_row == -1 || A[i][n_col] < A[min_row][
                            n_col])
                                min_row = i;
                if (A[min_row][n_col] > eps)
                        return true; // basic feasible

                ++n_col;
                for (int i = 0; i < n_col; i++)
                        saved[i] = obj[i];
                move_col(n_col, n_col-1);

                memset(obj, 0, sizeof obj);
                obj[n_col - 1] = 1;
                for (int i = 0; i < n_r; i++)
                        A[i][n_col - 1] = -1;

                pivot(min_row, n_col - 1);
```

```
                long double val = solve_feasible();
                if (val < -eps)
                        return false; // infeasible // !!!! promjena
                for (int i = 0; i < n_r; i++) {
                        if (ones[i] == n_col - 1) {
                                int maxj = -1;
                                for (int j = 0; j < n_col; j++)
                                        if (maxj == -1 || fabs(A[i][j
                                            ]) > fabs(A[i][maxj]))
                                                maxj = j;
                                pivot(i, maxj);
                        }
                }

                move_col(n_col-1, n_col);
                for (int i = 0; i < n_col; i++)
                        obj[i] = saved[i];
                --n_col;
                return true;
        }

        long double solve_all() {
                if (!get_feasibile()) return 1e100; // impossible
                return - solve_feasible(); // !!! promjena
        }
};
```

3.2 FFT

```
const int LG = 20; // IF YOU WANT TO CONVOLVE TWO ARRAYS OF LENGTH N
    AND M CHOOSE LG IN SUCH A WAY THAT 2^LG > n + m
const int MAX = 1 << LG;

struct point
{
        double real, imag;
        point(double _real = 0.0, double _imag = 0.0)
        {
                real = _real;
                imag = _imag;
        }
};
point operator + (point a, point b)
{
        return point(a.real + b.real, a.imag + b.imag);
}
point operator - (point a, point b)
{
        return point(a.real - b.real, a.imag - b.imag);
}
point operator * (point a, point b)
{
        return point(a.real * b.real - a.imag * b.imag, a.real * b.
            imag + a.imag * b.real);
}

void fft(point *a, bool inv)
{
        for (int mask = 0; mask < MAX; mask++)
        {
                int rev = 0;
                for (int i = 0; i < LG; i++)
                        if ((1 << i) & mask)
                                rev |= (1 << (LG - 1 - i));
                if (mask < rev)
                        swap(a[mask], a[rev]);
        }
        for (int len = 2; len <= MAX; len *= 2)
        {
                double ang = 2.0 * M_PI / len;
                if (inv)
                        ang *= -1.0;
                point wn(cos(ang), sin(ang));
                for (int i = 0; i < MAX; i += len)
                {
                        point w(1.0, 0.0);
                        for (int j = 0; j < len / 2; j++)
                        {
                                point t1 = a[i + j] + w * a[i + j +
                                    len / 2];
                                point t2 = a[i + j] - w * a[i + j +
                                    len / 2];
                                a[i + j] = t1;
                                a[i + j + len / 2] = t2;
                                w = w * wn;
                        }
                }
        }
        if (inv)
                for (int i = 0; i < MAX; i++)
                {
                        a[i].real /= MAX;
                        a[i].imag /= MAX;
                }
}
```

3.3 NTT

```
const int MOD = 998244353;
const int LG = 16; // IF YOU WANT TO CONVOLVE TWO ARRAYS OF LENGTH N
    AND M CHOOSE LG IN SUCH A WAY THAT 2^LG > n + m
const int MAX = (1 << LG);
const int ROOT = 44759; // ENSURE THAT ROOT^2^(LG - 1) = MOD - 1
int bpow(int a, int b)
{
        int ans = 1;
        while (b)
        {
                if (b & 1)
                        ans = 1LL * ans * a % MOD;
                b >>= 1;
                a = 1LL * a * a % MOD;
        }
        return ans;
}
void ntt(int *a, bool inv)
{
        for (int mask = 0; mask < MAX; mask++)
        {
                int rev = 0;
```

```
		for (int i = 0; i < LG; i++)
			if ((1 << i) & mask)
				rev |= (1 << (LG - 1 - i));
		if (mask < rev)
			swap(a[mask], a[rev]);
	}
	for (int len = 2; len <= MAX; len *= 2)
	{
		int wn = bpow(ROOT, MAX / len);
		if (inv)
			wn = bpow(wn, MOD - 2);
		for (int i = 0; i < MAX; i += len)
		{
			int w = 1;
			for (int j = 0; j < len / 2; j++)
			{
				int l = a[i + j];
				int r = 1LL * w * a[i + j + len / 2] %
				    MOD;
				a[i + j] = (l + r);
				a[i + j + len / 2] = l - r + MOD;
				if (a[i + j] >= MOD)
					a[i + j] -= MOD;
				if (a[i + j + len / 2] >= MOD)
					a[i + j + len / 2] -= MOD;
				w = 1LL * w * wn % MOD;
			}
		}
	}
	if (inv)
	{
		int x = bpow(MAX, MOD - 2);
		for (int i = 0; i < MAX; i++)
			a[i] = 1LL * a[i] * x % MOD;
	}
}
```

4 String

4.1 Manacher

```
int m[MAXN]; // SET MAXIMUM LENGTH OF STRING
void build(string s)
{
	int n = s.size();
	int l = 0, r = 0;
	for (int i = 0; i < n; i++)
		if (r <= i)
		{
			l = i, r = i + 1;
			while (2 * l - r >= 0 && s[r] == s[2 * l - r])
				r++;
			m[i] = r - l;
		}
		else if (m[2 * l - i] < r - i)
			m[i] = m[2 * l - i];
		else
		{
			l = i;
			while (2 * l - r >= 0 && s[r] == s[2 * l - r])
				r++;
			m[i] = r - l;
		}
}
```

4.2 Palindromic tree

```
struct PalindromicTree
{
	struct node
	{
		int to[SIGMA]; // SET MAXIMUM NUMBER OF CHARACTERS IN
		    ALPHABET
		int link, len;
		node()
		{
			for (int i = 0; i < SIGMA; i++)
				to[i] = -1;
			link = len = 0;
		}
	} tree[MAXN]; // SET MAXIMUM LENGTH OF STRING
	int sz, suf;
	string s;
	void clear()
	{
		sz = 0;
		tree[sz++] = node();
		tree[sz++] = node();
		tree[0].len = -1;
		suf = 1;
		s = "";
	}
	bool add_letter(int c)
	{
		int pos = s.size();
		s += char(c);
		while (pos - tree[suf].len - 1 < 0 || s[pos] != s[pos
		    - tree[suf].len - 1])
			suf = tree[suf].link;
		if (tree[suf].to[c] != -1)
		{
			suf = tree[suf].to[c];
			return false;
		}
		tree[sz] = node();
		tree[sz].len = tree[suf].len + 2;
		tree[suf].to[c] = sz++;
		int cur = suf;
		suf = sz - 1;
		if (tree[suf].len == 1)
		{
			tree[suf].link = 1;
			return true;
		}
		do
		{
			cur = tree[cur].link;
```

```
                } while (pos - tree[cur].len - 1 < 0 || s[pos] != s[
                    pos - tree[cur].len - 1]);
                tree[suf].link = tree[cur].to[c];
                return true;
        }
        PalindromicTree()
        {
                clear();
        }
};
```

4.3 Z function

```
int z[MAXN]; // SET MAXIMUM LENGTH OF STRING
void build(string s)
{
        int n = s.size();
        z[0] = n;
        int l = 0, r = 0;
        for (int i = 1; i < n; i++)
                if (r < i)
                {
                        l = r = i;
                        while (r < n && s[r - l] == s[r])
                                r++;
                        z[i] = r - l;
                }
                else if (z[i - l] < r - i)
                        z[i] = z[i - l];
                else
                {
                        l = i;
                        while (r < n && s[r - l] == s[r])
                                r++;
                        z[i] = r - l;
                }
}
```

5 Data structure

5.1 Treap

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef pair<int, int> pii;

struct Treap {
        typedef pii T;
        typedef struct _Node {
                T x;
                int y, cnt;
                _Node *l, *r;
                _Node(T _x) {
                        x = _x;
                        y = ((rand() & ((1 << 16) - 1)) << 16) ^ rand
                            ();
                        l = r = NULL;
                        cnt = 1;
                }
                ~_Node() { delete l; delete r; }
                void recalc() {
                        cnt = 0;
                        if (l)
                                cnt += l->cnt;
                        cnt++;
                        if (r)
                                cnt += r->cnt;
                }
                void debug() {
                        if (l)
                                l->debug();
                        if (r)
                                r->debug();
                        cerr << x.first << " " << x.second << " " << (
                            l ? l->x.first : -1) << " " << (r ? r->x.
                            first : -1) << " " << cnt << endl;
                }
        } *Node;

Node merge(Node l, Node r) {
        if (!l || !r) return l ? l : r;
        if (l->y < r->y) {
                l->r = merge(l->r, r);
                l->recalc();
                return l;
        } else {
                r->l = merge(l, r->l);
                r->recalc();
                return r;
        }
}

void split(Node v, T x, Node &l, Node &r, bool eq=false) {
        l = r = NULL;
        if (!v) return;
        if (v->x < x || (eq && v->x == x)) {
                split(v->r, x, v->r, r);
                l = v;
        } else {
                split(v->l, x, l, v->l);
                r = v;
        }
        v->recalc();
}

Node root;
Treap() : root(NULL) {}
~Treap() { delete root; }
void insert(T x) {
        Node l, r;
        split(root, x, l, r);
        root = merge(merge(l, new _Node(x)), r);
}
```

```
void erase(T x) {
        Node l, m, r;
        split(root, x, l, m);
        split(m, x, m, r, true);
        // assert(m && m->cnt == 1 && m->x == x);
        delete m;
        root = merge(l, r);
        }
        int size() const { return root ? root->cnt : 0; }
};

Treap t;
```