Contents

1 Combinatorial optimization

1.1 Dinic's

```cpp
struct Dinic {
  struct Edge {
    int u, v;
    long long cap, flow;
    Edge() {}
    Edge(int u, int v, long long cap): u(u), v(v), cap(
        cap), flow(0) {}
  };
  int N;
  vector<Edge> E;
  vector<vector<int>> g;
  vector<int> d, pt;

  Dinic(int N): N(N), E(0), g(N), d(N), pt(N) {}

  void AddEdge(int u, int v, long long cap) {
    if (u != v) {
      E.emplace_back(Edge(u, v, cap));
      g[u].emplace_back(E.size() - 1);
      E.emplace_back(Edge(v, u, 0));
      g[v].emplace_back(E.size() - 1);
    }
  }

  bool BFS(int S, int T) {
    queue<int> q({S});
    fill(d.begin(), d.end(), N + 1);
    d[S] = 0;
    while(!q.empty()) {
      int u = q.front(); q.pop();
      if (u == T) break;
      for (int k: g[u]) {
        Edge &e = E[k];
        if (e.flow < e.cap && d[e.v] > d[e.u] + 1) {
          d[e.v] = d[e.u] + 1;
          q.emplace(e.v);
        }
      }
    }
    return d[T] != N + 1;
  }
  long long DFS(int u, int T, long long flow = -1) {
    if (u == T || flow == 0) return flow;
```

```
    for (int &i = pt[u]; i < g[u].size(); ++i) {
      Edge &e = E[g[u][i]];
      Edge &oe = E[g[u][i]^1];
      if (d[e.v] == d[e.u] + 1) {
        long long amt = e.cap - e.flow;
        if (flow != -1 && amt > flow) amt = flow;
        if (long long pushed = DFS(e.v, T, amt)) {
          e.flow += pushed;
          oe.flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }
  long long MaxFlow(int S, int T) {
    long long total = 0;
    while (BFS(S, T)) {
      fill(pt.begin(), pt.end(), 0);
      while (long long flow = DFS(S, T))
        total += flow;
    }
    return total;
  }
};
```

1.2 Min-cost Circulation

```
// Runs in O(<max_flow > * log(V * max_edge_cost)) = O(
    V^3 * // log(V * C)) Really fast in // practice , 3e4
     edges are fine. Operates on integers , // costs are
    multiplied by N!!
template<typename flow_t = int, typename cost_t = int>
struct mcSFlow{
  struct Edge{
    cost_t c;
    flow_t f;
    int to, rev;
    Edge(int _to, cost_t _c, flow_t _f, int _rev):c(_c),
        f(_f), to(_to), rev(_rev){}
  };
  static constexpr cost_t INFCOST = numeric_limits<
      cost_t>::max()/2;
  cost_t eps;
  int N, S, T;
  vector<vector<Edge> > G;
  vector<unsigned int> isq, cur;
  vector<flow_t> ex;
  vector<cost_t> h;
  mcSFlow(int _N, int _S, int _T):eps(0), N(_N), S(_S),
      T(_T), G(_N){}
  void add_edge(int a, int b, cost_t cost, flow_t cap){
    assert(cap>=0);
    assert(a>=0&&a<N&&b>=0&&b<N);
    if(a==b){assert(cost>=0); return;}
    cost*=N;
    eps = max(eps, abs(cost));
    G[a].emplace_back(b, cost, cap, G[b].size());
    G[b].emplace_back(a, -cost, 0, G[a].size()-1);
  }
  void add_flow(Edge& e, flow_t f) {
    Edge &back = G[e.to][e.rev];
    if (!ex[e.to] && f)
      hs[h[e.to]].push_back(e.to);
    e.f -= f; ex[e.to] += f;
    back.f += f; ex[back.to] -= f;
  }
  vector<vector<int> > hs;
  vector<int> co;
  flow_t max_flow() {
    ex.assign(N, 0);
    h.assign(N, 0); hs.resize(2*N);
    co.assign(2*N, 0); cur.assign(N, 0);
    h[S] = N;
    ex[T] = 1;
    co[0] = N-1;
    for(auto &e:G[S]) add_flow(e, e.f);
    if(hs[0].size())
    for (int hi = 0;hi>=0;) {
      int u = hs[hi].back();
      hs[hi].pop_back();
      while (ex[u] > 0) { // discharge u
        if (cur[u] == G[u].size()) {
          h[u] = 1e9;
          for(unsigned int i=0;i<G[u].size();++i){
            auto &e = G[u][i];
            if (e.f && h[u] > h[e.to]+1){
              h[u] = h[e.to]+1, cur[u] = i;
            }
          }
          if (++co[h[u]], !--co[hi] && hi < N)
            for(int i=0;i<N;++i)
              if (hi < h[i] && h[i] < N){
                --co[h[i]];
                h[i] = N + 1;
              }
          hi = h[u];
        } else if (G[u][cur[u]].f && h[u] == h[G[u][cur[
            u]].to]+1)
          add_flow(G[u][cur[u]], min(ex[u], G[u][cur[u
              ]].f));
        else ++cur[u];
      }
      while (hi>=0 && hs[hi].empty()) --hi;
    }
    return -ex[S];
  }
  void push(Edge &e, flow_t amt){
    if(e.f < amt) amt=e.f;
    e.f-=amt; ex[e.to]+=amt;
    G[e.to][e.rev].f+=amt; ex[G[e.to][e.rev].to]-=amt;
  }
  void relabel(int vertex){
    cost_t newHeight = -INFCOST;
    for(unsigned int i=0;i<G[vertex].size();++i){
      Edge const&e = G[vertex][i];
      if(e.f && newHeight < h[e.to]-e.c){
        newHeight = h[e.to] - e.c;
        cur[vertex] = i;
      }
    }
    h[vertex] = newHeight - eps;
```

```
  }
  static constexpr int scale=2;
  pair<flow_t, cost_t> minCostMaxFlow(){
    cost_t retCost = 0;
    for(int i=0;i<N;++i)
      for(Edge &e:G[i])
        retCost += e.c*(e.f);
    //find max-flow
    flow_t retFlow = max_flow();
    h.assign(N, 0); ex.assign(N, 0);
    isq.assign(N, 0); cur.assign(N,0);
    queue<int> q;
    for(;eps;eps>>=scale){
      //refine
      fill(cur.begin(), cur.end(), 0);
      for(int i=0;i<N;++i)
        for(auto &e:G[i])
          if(h[i] + e.c - h[e.to] < 0 && e.f) push(e, e.
              f);
      for(int i=0;i<N;++i){
        if(ex[i]>0){
          q.push(i);
          isq[i]=1;
        }
      }
      // make flow feasible
      while(!q.empty()){
        int u=q.front();q.pop();
        isq[u]=0;
        while(ex[u]>0){
          if(cur[u] == G[u].size())
            relabel(u);
          for(unsigned int &i=cur[u], max_i = G[u].size
              ();i<max_i;++i){
            Edge &e=G[u][i];
            if(h[u] + e.c - h[e.to] < 0){
              push(e, ex[u]);
              if(ex[e.to]>0 && isq[e.to]==0){
                q.push(e.to);
                isq[e.to]=1;
              }
              if(ex[u]==0) break;
            }
          }
        }
      }
      if(eps>1 && eps>>scale==0){
        eps = 1<<scale;
      }
    }
    for(int i=0;i<N;++i){
      for(Edge &e:G[i]){
        retCost -= e.c*(e.f);
      }
    }
    return make_pair(retFlow, retCost/2/N);
  }
  flow_t getFlow(Edge const &e){
    return G[e.to][e.rev].f;
  }
};
```

1.3 Edmonds Max Matching

```
#define MAXN 505
vector<int>g[MAXN];
int pa[MAXN],match[MAXN],st[MAXN],S[MAXN],v[MAXN];
int t,n;
inline int lca(int x,int y){
  for(++t;;swap(x,y)) if(x){
    if(v[x]==t)return x;
    v[x]=t, x=st[pa[match[x]]];
  }
}
#define qpush(x) q.push(x),S[x]=0
void flower(int x,int y,int l,queue<int> &q){
  while(st[x]!=l){
    pa[x]=y, y=match[x];
    if(S[y]==1) qpush(y);
    st[x]=st[y]=l, x=pa[y];
  }
}
bool bfs(int x){
  for(int i=1;i<=n;++i)st[i]=i;
  memset(S+1,-1,sizeof(int)*n);
  queue<int>q;
  qpush(x);
  while(q.size()){
    x=q.front(),q.pop();
    for(size_t i=0;i<g[x].size();++i){
      int y=g[x][i];
      if(S[y]==-1){
        pa[y]=x, S[y]=1;
        if(!match[y]){
          for(int lst;x;y=lst,x=pa[y]){
            lst=match[x], match[x]=y, match[y]=x;
          }
          return 1;
        }
        qpush(match[y]);
      } else if(!S[y]&&st[y]!=st[x]){
        int l=lca(y,x);
        flower(y,x,l,q); flower(x,y,l,q);
      }
    }
  }
  return 0;
}
int blossom(){
  int ans=0;
  for(int i=1;i<=n;++i)
    if(!match[i]&&bfs(i))++ans;
  return ans;
}
int main(){
  while(m--){
    g[x].push_back(y);
    g[y].push_back(x);
  }
  printf("%d\n",blossom());
  for(int i=1;i<=n;i++)printf("%d ",match[i]);
}
```

1.4 Hungarian Min-cost

```
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;
}
```

1.5 Konig's Theorem (Text)

In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover. To exhibit the vertex cover:

1. Find a maximum matching
2. Change each edge **used** in the matching into a directed edge from **right to left**
3. Change each edge **not used** in the matching into a directed edge from **left to right**
4. Compute the set T of all vertices reachable from unmatched vertices on the left (including themselves)
5. The vertex cover consists of all vertices on the right that are **in** T, and all vertices on the left that are **not in** T

1.6 Minimum Edge Cover (Text)

If a minimum edge cover contains C edges, and a maximum matching contains M edges, then $C + M = |V|$. To obtain the edge cover, start with a maximum matching, and then, for every vertex not matched, just select some edge incident upon it and add it to the edge set.

2 Geometry

2.1 Miscellaneous geometry

```
double INF = 1e100,EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x,
      y+p.y); }
  PT operator - (const PT &p)  const { return PT(x-p.x,
      y-p.y); }
  PT operator * (double c)     const { return PT(x*c,
      y*c  ); }
  PT operator / (double c)     const { return PT(x/c,
      y/c  ); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  os << "(" << p.x << "," << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t)
      );
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+
    by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c,
                          double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are
    parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return
          true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-
        b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return
      false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return
      false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that
    unique
// intersection exists; for segment intersection, check
    if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c
      , c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex
    polygon (by William
// Randolph Franklin); returns 1 for strictly interior
    points, 0 for
// strictly exterior points, and 0 or 1 for the
    remaining points.
// Note that it is possible to convert this into an *
    exact* test using
// integer arithmetic by taking care of the division
    appropriately
// (making sure to deal with signs properly) and then by
     writing exact
```

```
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y)
         / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()
       ], q), q) < EPS)
      return true;
    return false;
}

// compute intersection of line through points a and b
   with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c,
    double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with
   radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r
   , double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (
   possibly nonconvex)
// polygon, assuming that the coordinates are listed in
   a clockwise or
// counterclockwise fashion.  Note that the centroid is
   often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW
   order) is simple
bool IsSimple(const vector<PT> &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 (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```

2.2 3D geometry

```
public class Geom3D {
  // distance from point (x, y, z) to plane aX + bY + cZ
     + d = 0
  public static double ptPlaneDist(double x, double y,
      double z,
       double a, double b, double c, double d) {
    return Math.abs(a*x + b*y + c*z + d) / Math.sqrt(a*a
        + b*b + c*c);
  }

  // distance between parallel planes aX + bY + cZ + d1
      = 0 and
  // aX + bY + cZ + d2 = 0
  public static double planePlaneDist(double a, double b
      , double c,
       double d1, double d2) {
    return Math.abs(d1 - d2) / Math.sqrt(a*a + b*b + c*c
        );
  }
```

```
  // distance from point (px, py, pz) to line (x1, y1,
      z1)-(x2, y2, z2)
  // (or ray, or segment; in the case of the ray, the
      endpoint is the
  // first point)
  public static final int LINE = 0;
  public static final int SEGMENT = 1;
  public static final int RAY = 2;
  public static double ptLineDistSq(double x1, double y1
      , double z1,
      double x2, double y2, double z2, double px, double
          py, double pz,
      int type) {
    double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1
        -z2)*(z1-z2);

    double x, y, z;
    if (pd2 == 0) {
      x = x1;
      y = y1;
      z = z1;
    } else {
      double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (
          pz-z1)*(z2-z1)) / pd2;
      x = x1 + u * (x2 - x1);
      y = y1 + u * (y2 - y1);
      z = z1 + u * (z2 - z1);
      if (type != LINE && u < 0) {
        x = x1;
        y = y1;
        z = z1;
      }
      if (type == SEGMENT && u > 1.0) {
        x = x2;
        y = y2;
        z = z2;
      }
    }

    return (x-px)*(x-px) + (y-py)*(y-py) + (z-pz)*(z-pz)
        ;
  }

  public static double ptLineDist(double x1, double y1,
      double z1,
      double x2, double y2, double z2, double px, double
          py, double pz,
      int type) {
    return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2
        , px, py, pz, type));
  }
}
```

2.3 Convex hull

```
typedef pair<long long, long long> PT;
inline long long cross(PT o, PT a, PT b){
  PT OA = {a.first-o.first,a.second-o.second};
  PT OB = {b.first-o.first,b.second-o.second};
  return OA.first*OB.second - OA.second*OB.first;
}
inline long double dist(PT a, PT b){
  return sqrt(pow(a.first-b.first,2)+pow(a.second-b.
      second,2));
}
vector<PT> convexhull(vector<PT> a){
  vector<PT> hull;
  sort(a.begin(),a.end(),[](PT i, PT j){
    if(i.second!=j.second)
      return i.second < j.second;
    return i.first < j.first;
  });
  for(int i=0;i<a.size();++i){
    while(hull.size()>1 && cross(hull[hull.size()-2],
        hull.back(),a[i])<=0)
      hull.pop_back();
    hull.push_back(a[i]);
  }
  for(int i=a.size()-1, siz = hull.size();i--;){
    while(hull.size()>siz && cross(hull[hull.size()-2],
        hull.back(),a[i])<=0)
      hull.pop_back();
    hull.push_back(a[i]);
  }
  return hull;
}
```

2.4 Min Enclosing Circle

```
// Minimum enclosing circle, Welzl's algorithm
// Expected linear time.
// If there are any duplicate points in the input, be
    sure to remove them first.
struct point {
        double x;
        double y;
};
struct circle {
        double x;
        double y;
        double r;
        circle() {}
        circle(double x, double y, double r): x(x), y(y)
            , r(r) {}
};
circle b_md(vector<point> R) {
        if (R.size() == 0) {
                return circle(0, 0, -1);
        } else if (R.size() == 1) {
                return circle(R[0].x, R[0].y, 0);
        } else if (R.size() == 2) {
                return circle((R[0].x+R[1].x)/2.0,
                              (R[0].y+R[1].y)/2.0,
                                      hypot(R[0].x-R
                                          [1].x, R
                                          [0].y-R[1].
                                          y)/2.0);
        } else {
                double D = (R[0].x - R[2].x)*(R[1].y - R
                    [2].y) - (R[1].x - R[2].x)*(R[0].y -
                    R[2].y);
                double p0 = (((R[0].x - R[2].x)*(R[0].x
                    + R[2].x) + (R[0].y - R[2].y)*(R[0].y
```

```
                + R[2].y)) / 2 * (R[1].y - R[2].y) -
                ((R[1].x - R[2].x)*(R[1].x + R[2].x)
                + (R[1].y - R[2].y)*(R[1].y + R[2].y
                )) / 2 * (R[0].y - R[2].y))/D;
            double p1 = (((R[1].x - R[2].x)*(R[1].x
                + R[2].x) + (R[1].y - R[2].y)*(R[1].y
                + R[2].y)) / 2 * (R[0].x - R[2].x) -
                ((R[0].x - R[2].x)*(R[0].x + R[2].x)
                + (R[0].y - R[2].y)*(R[0].y + R[2].y
                )) / 2 * (R[1].x - R[2].x))/D;
            return circle(p0, p1, hypot(R[0].x - p0,
                R[0].y - p1));
        }
}
circle b_minidisk(vector<point>& P, int i, vector<point>
    R) {
        if (i == P.size() || R.size() == 3) {
            return b_md(R);
        } else {
            circle D = b_minidisk(P, i+1, R);
            if (hypot(P[i].x-D.x, P[i].y-D.y) > D.r)
                {
                    R.push_back(P[i]);
                    D = b_minidisk(P, i+1, R);
                }
            return D;
        }
}

// Call this function.
circle minidisk(vector<point> P) {
        random_shuffle(P.begin(), P.end());
        return b_minidisk(P, 0, vector<point>());
}
```

2.5 Pick's Theorem (Text)

For a polygon with all vertices on lattice points, $A = i + b/2 - 1$, where Akon is the area, i is the number of lattice points strictly within the polygon, and b is the number of lattice points on the boundary of the polygon. (Note, there is no generalization to higher dimensions)

2.6 Slow Delaunay triangulation

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

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

3 Numerical algorithms

3.1 Pollard Rho

```
typedef long long unsigned int llui;
typedef long long int lli;
typedef long double float64;

llui mul_mod(llui a, llui b, llui m){
  llui y = (llui)((float64)a*(float64)b/m+(float64)1/2);
  y = y * m;
  llui x = a * b;
  llui r = x - y;
  if ( (lli)r < 0 ){
    r = r + m; y = y - 1;
  }
  return r;
}
llui C,a,b;
llui gcd(){
  llui c;
  if(a>b){
    c = a; a = b; b = c;
```

```
    }
    while(1){
      if(a == 1LL) return 1LL;
      if(a == 0 || a == b) return b;
      c = a; a = b%a;
      b = c;
    }
  }
  llui f(llui a, llui b){
    llui tmp;
    tmp = mul_mod(a,a,b);
    tmp+=C; tmp%=b;
    return tmp;
  }
  llui pollard(llui n){
    if(!(n&1)) return 2;
    C=0;
    llui iteracoes = 0;
    while(iteracoes <= 1000){
      llui x,y,d;
      x = y = 2; d = 1;
      while(d == 1){
         x = f(x,n);
         y = f(f(y,n),n);
         llui m = (x>y)?(x-y):(y-x);
         a = m; b = n; d = gcd();
      }
      if(d != n)
         return d;
      iteracoes++; C = rand();
    }
    return 1;
  }
  llui pot(llui a, llui b, llui c){
    if(b == 0) return 1;
    if(b == 1) return a%c;
    llui resp = pot(a,b>>1,c);
    resp = mul_mod(resp,resp,c);
    if(b&1)
      resp = mul_mod(resp,a,c);
    return resp;
  }
  // Rabin-Miller primality testing algorithm
  bool isPrime(llui n){
    llui d = n-1;
    llui s = 0;
    if(n <=3 || n == 5) return true;
    if(!(n&1)) return false;
    while(!(d&1)){ s++; d>>=1; }
    for(llui i = 0;i<32;i++){
      llui a = rand();
      a <<=32;
      a+=rand();
      a%=(n-3); a+=2;
      llui x = pot(a,d,n);
      if(x == 1 || x == n-1) continue;
      for(llui j = 1;j<= s-1;j++){
        x = mul_mod(x,x,n);
        if(x == 1) return false;
        if(x == n-1)break;
      }
      if(x != n-1) return false;
    }
    return true;
  }
  map<llui,int> factors;
  // Precondition: factors is an empty map, n is a
     positive integer
  // Postcondition: factors[p] is the exponent of p in
     prime factorization of n
  void fact(llui n){
    if(!isPrime(n)){
      llui fac = pollard(n);
      fact(n/fac); fact(fac);
    }else{
      map<llui,int>::iterator it;
      it = factors.find(n);
      if(it != factors.end()){
        (*it).second++;
      }else{
        factors[n] = 1;
      }
    }
  }
```

3.2 Simplex algorithm

```
  // Two-phase simplex algorithm for solving linear
     programs of the form
  //     maximize     c^T x
  //     subject to   Ax <= b ; x >= 0
  // INPUT: A -- an m x n matrix
  //        b -- an m-dimensional vector
  //        c -- an n-dimensional vector
  //        x -- a vector where the optimal solution will
     be stored
  // OUTPUT: value of the optimal solution (infinity if
  //         unbounded above, nan if infeasible)
  // To use this code, create an LPSolver object with
  // A, b, and c as arguments. Then, call Solve(x).
  typedef long double DOUBLE;
  typedef vector<DOUBLE> VD;
  typedef vector<VD> VVD;
  typedef vector<int> VI;
  const DOUBLE EPS = 1e-9;
  struct LPSolver {
    int m, n;
    VI B, N;
    VVD D;
    LPSolver(const VVD &A, const VD &b, const VD &c) :
      m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2,
         VD(n + 2)) {
      for (int i = 0; i < m; i++) for (int j = 0; j < n; j
         ++) D[i][j] = A[i][j];
      for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n]
         = -1; D[i][n + 1] = b[i]; }
      for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c
         [j]; }
      N[n] = -1; D[m + 1][n] = 1;
    }
    void Pivot(int r, int s) {
      double inv = 1.0 / D[r][s];
```

```
      for (int i = 0; i < m + 2; i++) if (i != r)
        for (int j = 0; j < n + 2; j++) if (j != s)
          D[i][j] -= D[r][j] * D[i][s] * inv;
      for (int j = 0; j < n + 2; j++) if (j != s) D[r][j]
        *= inv;
      for (int i = 0; i < m + 2; i++) if (i != r) D[i][s]
        *= -inv;
      D[r][s] = inv;
      swap(B[r], N[s]);
    }
    bool Simplex(int phase) {
      int x = phase == 1 ? m + 1 : m;
      while (true) {
        int s = -1;
        for (int j = 0; j <= n; j++) {
          if (phase == 2 && N[j] == -1) continue;
          if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D
              [x][s] && N[j] < N[s]) s = j;
        }
        if (D[x][s] > -EPS) return true;
        int r = -1;
        for (int i = 0; i < m; i++) {
          if (D[i][s] < EPS) continue;
          if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n +
              1] / D[r][s] ||
              (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r
                ][s]) && B[i] < B[r]) r = i;
        }
        if (r == -1) return false;
        Pivot(r, s);
      }
    }
    DOUBLE Solve(VD &x) {
      int r = 0;
      for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n
          + 1]) r = i;
      if (D[r][n + 1] < -EPS) {
        Pivot(r, n);
        if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return
            -numeric_limits<DOUBLE>::infinity();
        for (int i = 0; i < m; i++) if (B[i] == -1) {
          int s = -1;
          for (int j = 0; j <= n; j++)
            if (s == -1 || D[i][j] < D[i][s] || D[i][j] ==
                 D[i][s] && N[j] < N[s]) s = j;
          Pivot(i, s);
        }
      }
      if (!Simplex(2)) return numeric_limits<DOUBLE>::
          infinity();
      x = VD(n);
      for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] =
          D[i][n + 1];
      return D[m][n + 1];
    }
  };
  int main() {
    const int m = 4, n = 3;
    DOUBLE _A[m][n] = {
      { 6, -1, 0 },
      { -1, -5, 0 },
      { 1, 5, 1 },
      { -1, -5, -1 }
    };
    DOUBLE _b[m] = { 10, -4, 5, -5 }, _c[n] = { 1, -1, 0
        };
    VVD A(m);
    VD b(_b, _b + m), c(_c, _c + n);
    for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n
        );
    LPSolver solver(A, b, c);
    VD x;
    DOUBLE value = solver.Solve(x);
    cerr << "VALUE: " << value << endl; // VALUE: 1.29032
    cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
    for (size_t i = 0; i < x.size(); i++) cerr << " " << x
        [i];
  }
```

3.3 Reduced row echelon form

```
// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting. This can be used for
// computing the rank of a matrix.
// Running time: O(n^3)
// INPUT:    a[][] = an nxm matrix
// OUTPUT:   rref[][] = an nxm matrix (stored in a[][])
//           returns rank of a[][]
const double EPSILON = 1e-10;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;
int rref(VVT &a) {
  int n = a.size(), m = a[0].size(), r = 0;
  for (int c = 0; c < m && r < n; c++) {
    int j = r;
    for (int i = r + 1; i < n; i++)
      if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
    if (fabs(a[j][c]) < EPSILON) continue;
    swap(a[j], a[r]);

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

3.4 Fast Fourier transform

```
template<typename fpt>
struct fft_wrap {
  using cpx_t = complex<fpt>;
  const fpt two_pi = 4 * acosl(0);

  vector<cpx_t> roots; //stores the N-th roots of unity.
  int N;
```

```
  fft_wrap(int N) : roots(N), N(N) {
    for (int i = 0; i < N; ++i) {
      roots[i] = EXP(two_pi * i / fpt(N));
    }
  }

  cpx_t EXP(fpt theta) {
    return {cos(theta), sin(theta)};
  }

  void fft(cpx_t *in, cpx_t *out, int size, int dir){
    bit_reverse(in, out, size);
    for (int s = 0; (1 << s) < size; ++s) {
      int s_ = s + 1;
      for (int k = 0; k < size; k += (1 << s_)) {
        for (int j = 0; j < (1 << s); ++j) {
          int id = (N + dir * (N >> s_) * j) & (N - 1);
          cpx_t w = roots[id];
          cpx_t t = w * out[k + j + (1 << s)];
          cpx_t u = out[k + j];
          out[k + j] = u + t;
          out[k + j + (1 << s)] = u - t;
        }
      }
    }
  }

  void bit_reverse(cpx_t *in, cpx_t *out, int size){
    for (int i = 0; i < size; ++i) {
      int rev = 0, i_copy = i;
      for (int j = 0; (1 << j) < size; ++j) {
        rev = (rev << 1) + (i_copy & 1);
        i_copy >>= 1;
      }
      out[rev] = in[i];
    }
  }
};

int main(){
  typedef complex<double> cpx_t;
  fft_wrap<double> fft_wrapper(2048);

  vector<cpx_t> in = {1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0,
      0.0}, out(8);

  fft_wrapper.fft(&in[0], &out[0], 8, 1);
  fft_wrapper.fft(&out[0], &in[0], 8, -1);
  for (int i = 0; i < 8; ++i) {
    cout << in[i].real() << ' ' << in[i].imag() << endl;
  }
}
```

3.5 Number Theoretic transform

```
// gen should be 5^((p - 1) / fft_wrapper_size)

template<int P>
struct field_t { ... };

template<typename fpt, int gen>
struct fft_wrap {
  vector<fpt> roots; //stores the N-th roots of unity.
  int N;

  fft_wrap(int _N) : roots(_N), N(_N) {
    roots[0] = 1;
    for (int i = 1; i < N; ++i) {
      roots[i] = roots[i - 1] * gen;
      if (i != N - 1) {
        assert(roots[i].v != 1);
      }
    }
  }
  void fft(fpt *in, fpt *out, int size, int dir) {
    bit_reverse(in, out, size);
    for (int s = 0; (1 << s) < size; ++s) {
      int s_ = s + 1;
      for (int k = 0; k < size; k += (1 << s_)) {
        for (int j = 0; j < (1 << s); ++j) {
          int id = (N + dir * (N >> s_) * j) & (N - 1);
          fpt w = roots[id];
          fpt t = w * out[k + j + (1 << s)];
          fpt u = out[k + j];
          out[k + j] = u + t;
          out[k + j + (1 << s)] = u - t;
        }
      }
    }
  }
  void bit_reverse(fpt *in, fpt *out, int size) {
    for (int i = 0; i < size; ++i) {
      int rev = 0, i_copy = i;
      for (int j = 0; (1 << j) < size; ++j) {
        rev = (rev << 1) + (i_copy & 1);
        i_copy >>= 1;
      }
      out[rev] = in[i];
    }
  }
};

using fpt = field_t<mod>;
fft_wrap<fpt, gen> fft_wrapper(1 << 18);

vector<fpt> polymul(vector<fpt> P1, vector<fpt> P2) {
  int fsize = P1.size() + P2.size() - 1;
  int N = max(P1.size(), P2.size());
  while(N & (N - 1)) {
    ++N;
  }
  N *= 2;
  P1.resize(N); P2.resize(N);
  vector<fpt> temp(N);
  fft_wrapper.fft(&P1[0], &temp[0], N, 1);
  fft_wrapper.fft(&P2[0], &P1[0], N, 1);
  for (int i = 0; i < N; ++i) {
    P1[i] *= temp[i];
  }
  fft_wrapper.fft(&P1[0], &temp[0], N, -1);
  field_t<mod> inv(N);
  inv = inv.modexp(mod - 2);
  for (int i = 0; i < N; ++i) {
    temp[i] *= inv;
  }
  temp.resize(fsize);
  return temp;
```

```
}
```

3.6 Discrete Logarithm

```
// Calculates x such that g^x % md == h
int baby_giant(int g, int h, int md){
  unordered_map<int,int> mp;
  int sq = ceil(sqrtl(md));
  for(int i=0,now=1;i<sq;++i)
    mp[now] = i, now = (long long) now*g % md;
  for(int i=0,jmp=power(g,md-1-sq);i<sq;++i){
    if(mp.find(h)!=mp.end()) return i*sq+mp[h];
    h = (long long) h*jmp % md;
  }
  return -1;
}
```

3.7 Mobius Inversion (Text)

$$\mu(n) = \begin{cases} 0 & n \text{ not squarefree} \\ 1 & n \text{ squarefree w/ even no. of prime factors} \\ 1 & n \text{ squarefree w/ odd no. of prime factors} \end{cases}$$

Note that $\mu(a)\mu(b) = \mu(ab)$ for a, b relatively prime

Also $\sum_{d|n} \mu(d) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{otherwise} \end{cases}$

Möbius Inversion If $g(n) = \sum_{d|n} f(d)$ for all $n \geq 1$, then $f(n) = \sum_{d|n} \mu(d) g(n/d)$ for all $n \geq 1$.

3.8 Burnside Lemma (Text)

The number of orbits of a set X under the group action G equals the average number of elements of X fixed by the elements of G.

Here's an example. Consider a square of $2n$ times $2n$ cells. How many ways are there to color it into X colors, up to rotations and/or reflections? Here, the group has only 8 elements (rotations by 0, 90, 180 and 270 degrees, reflections over two diagonals, over a vertical line and over a horizontal line). Every coloring stays itself after rotating by 0 degrees, so that rotation has X^{4n^2} fixed points. Rotation by 180 degrees and reflections over a horizonal/vertical line split all cells in pairs that must be of the same color for a coloring to be unaffected by such rotation/reflection, thus there exist X^{2n^2} such colorings for each of them. Rotations by 90 and 270 degrees split cells in groups of four, thus yielding X^{n^2} fixed colorings. Reflections over diagonals split cells into $2n$ groups of 1 (the diagonal itself) and $2n^2 - n$ groups of 2 (all remaining cells), thus yielding $X^{2n^2-n+2n} = X^{2n^2+n}$ unaffected colorings. So, the answer is $(X^{4n^2} + 3X^{2n^2} + 2X^{n^2} + 2X^{2n^2+n})/8$.

3.9 Number Theory (Modular, CRT, Linear Diophantine)

```
// All algorithms described here work on nonnegative
   integers.
int mod(int a, int b) {
  return ((a%b) + b) % b;
}
int lcm(int a, int b) {
  return a / __gcd(a, b)*b;
}
int powermod(int a, int b, int m){
  return b?powermod(a*a%m,b/2,m)*(b%2?a:1)%m:1;
}
// returns g = gcd(a, b); finds x, y such that d = ax +
   by
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;
}
// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
  int x, y;
  VI ret;
  int g = extended_euclid(a, n, x, y);
  if (!(b%g)) {
    x = mod(x*(b / g), n);
    for (int i = 0; i < g; i++)
      ret.push_back(mod(x + i*(n / g), n));
  }
  return ret;
}
// computes b such that ab = 1 (mod n), returns -1 on
   failure
int mod_inverse(int a, int n) {
  int x, y;
  int g = extended_euclid(a, n, x, y);
  if (g > 1) return -1;
  return mod(x, n);
}
// Chinese remainder theorem (special case): find z such
// that z % m1 = r1, z % m2 = r2.
// Here, z is unique modulo M = lcm(m1, m2).
// Return (z, M). On failure, M = -1.
PII chinese_remainder_theorem(int m1, int r1, int m2,
   int r2) {
  int s, t;
  int g = extended_euclid(m1, m2, s, t);
  if (r1%g != r2%g) return make_pair(0, -1);
  return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1
     *m2 / g);
}
// Find z such that z % m[i] = r[i] for all i.
// The solution is unique modulo M = lcm_i (m[i]).
// Return (z, M). On failure, M = -1.
// We don't require a[i]'s to be relatively prime.
PII chinese_remainder_theorem(const VI &m,const VI &r){
```

```
  PII ret = make_pair(r[0], m[0]);
  for (int i = 1; i < m.size(); i++) {
    ret = chinese_remainder_theorem(ret.second, ret.
        first, m[i], r[i]);
    if (ret.second == -1) break;
  }
  return ret;
}
// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int
     &y) {
  if (!a && !b) {
    if (c) return false;
    x = 0; y = 0;
    return true;
  }
  if (!a) {
    if (c % b) return false;
    x = 0; y = c / b;
    return true;
  }
  if (!b) {
    if (c % a) return false;
    x = c / a; y = 0;
    return true;
  }
  int g = __gcd(a, b);
  if (c % g) return false;
  x = c / g * mod_inverse(a / g, b / g);
  y = (c - a*x) / b;
  return true;
}
```

4 Graph algorithms

4.1 Dynamic Connectivity

```
struct UnionFind {
  int n,comp;
  vector<int> uf,si,c;
  UnionFind(int n=0):n(n),comp(n),uf(n),si(n,1){
    for(int i=0;i<n;++i)
      uf[i]=i;
  }
  int find(int x){return x==uf[x]?x:find(uf[x]);}
  bool join(int x, int y){
    if((x=find(x))==(y=find(y))) return false;
    if(si[x]<si[y]) swap(x,y);
    si[x]+=si[y];uf[y]=x;comp--;
    c.push_back(y);
    return true;
  }
  int snap(){return c.size();}
  void rollback(int snap){
    while(c.size()>snap){
      int x=c.back(); c.pop_back();
      si[uf[x]]-=si[x];uf[x]=x;comp++;
    }
  }
};
enum {ADD,DEL,QUERY};
struct Query {int type,x,y;};
struct DynCon {
  vector<Query> q;
  UnionFind dsu;
  vector<int> mt;
  map<pair<int,int>,int> last;
  DynCon(int n):dsu(n){}
  void add(int x, int y){
    if(x>y) swap(x,y);
    q.push_back((Query){ADD,x,y}),mt.push_back(-1);
    last[make_pair(x,y)]=q.size()-1;
  }
  void remove(int x, int y){
    if(x>y)swap(x,y);
    q.push_back((Query){DEL,x,y});
    int pr=last[make_pair(x,y)];
    mt[pr]=q.size()-1;
    mt.push_back(pr);
  }
  void query(int x, int y){
    q.push_back((Query){QUERY,x,y});
    mt.push_back(-1);
  }
  void process(){ // answers all queries in order
    if(!q.size()) return;
    for(int i=0;i<q.size();++i)
      if(q[i].type==ADD&&mt[i]<0)
        mt[i]=q.size();
    go(0,q.size());
  }
  void go(int l, int r){
    if(l+1==r){
      if(q[l].type==QUERY) // answer query using DSU
        puts(dsu.find(q[l].x)==dsu.find(q[l].y)? "YES":"
            NO");
      return;
    }
    int s=dsu.snap(),m=(l+r)/2;
    for(int i=r-1;i>=m;--i)
      if(mt[i]>=0&&mt[i]<l)
        dsu.join(q[i].x,q[i].y);
    go(l,m);
    dsu.rollback(s);
    for(int i=m-1;i>=l;--i)if(mt[i]>=r)dsu.join(q[i].x,q
        [i].y);
    go(m,r);
    dsu.rollback(s);
  }
};
```

4.2 Bridges

```
// Finds bridges and cut vertices
//
// Receives:
// N: number of vertices
// l: adjacency list
//
// Gives:
// vis, seen, par (used to find cut vertices)
```

```
// ap - 1 if it is a cut vertex, 0 otherwise
// brid - vector of pairs containing the bridges

typedef pair<int, int> PII;

int N;
vector <int> l[MAX];
vector <PII> brid;
int vis[MAX], seen[MAX], par[MAX], ap[MAX];
int cnt, root;

void dfs(int x){
  if(vis[x] != -1)
    return;
  vis[x] = seen[x] = cnt++;

  int adj = 0;
  for(int i = 0; i < (int)l[x].size(); i++){
    int v = l[x][i];
    if(par[x] == v)
      continue;
    if(vis[v] == -1){
      adj++;
      par[v] = x;
      dfs(v);
      seen[x] = min(seen[x], seen[v]);
      if(seen[v] >= vis[x] && x != root)
        ap[x] = 1;
      if(seen[v] == vis[v])
        brid.push_back(make_pair(v, x));
    }
    else{
      seen[x] = min(seen[x], vis[v]);
      seen[v] = min(seen[x], seen[v]);
    }
  }
  if(x == root) ap[x] = (adj>1);
}

void bridges(){
  brid.clear();
  for(int i = 0; i < N; i++){
    vis[i] = seen[i] = par[i] = -1;
    ap[i] = 0;
  }
  cnt = 0;
  for(int i = 0; i < N; i++)
    if(vis[i] == -1){
      root = i;
      dfs(i);
    }
}
```

4.3 Strongly connected components

```
struct SCC {
  int V, group_cnt;
  vector<vector<int> > adj, radj;
  vector<int> group_num, vis;
  stack<int> stk;
  // V = number of vertices
  SCC(int V): V(V), group_cnt(0), group_num(V), vis(V),
      adj(V), radj(V) {}

  // Call this to add an edge (0-based)
  void add_edge(int v1, int v2) {
    adj[v1].push_back(v2);
    radj[v2].push_back(v1);
  }
  void fill_forward(int x) {
    vis[x] = true;
    for (int i = 0; i < adj[x].size(); i++) {
      if (!vis[adj[x][i]]) {
        fill_forward(adj[x][i]);
      }
    }
    stk.push(x);
  }
  void fill_backward(int x) {
    vis[x] = false;
    group_num[x] = group_cnt;
    for (int i = 0; i < radj[x].size(); i++) {
      if (vis[radj[x][i]]) {
        fill_backward(radj[x][i]);
      }
    }
  }
  // Returns number of strongly connected components.
  // After this is called, group_num contains component
      assignments (0-based)
  int get_scc() {
    for (int i = 0; i < V; i++) {
      if (!vis[i]) fill_forward(i);
    }
    group_cnt = 0;
    while (!stk.empty()) {
      if (vis[stk.top()]) {
        fill_backward(stk.top());
        group_cnt++;
      }
      stk.pop();
    }
    return group_cnt;
  }
};
```

5 String Stuff

5.1 Suffix Automaton

```
struct SuffixAutomaton {
  vector<map<char,int>> edges; // edges[i]: the labeled
      edges from node i
  vector<int> link;   // link[i]  : the parent of i
  vector<int> length; // length[i]: length of longest
      string in ith class
  vector<int> cnt;    // No. of times substring occurs
  int last;  // index of equivalence class of whole
      string
  SuffixAutomaton(string const& s) {
    // add the initial node
    edges.push_back(map<char,int>());
    link.push_back(-1);
    length.push_back(0);
```

```
    cnt.push_back(0);
    last = 0;

    for(int i=0;i<s.size();i++) {
      // construct r
      edges.push_back(map<char,int>());
      length.push_back(i+1);
      link.push_back(0);
      cnt.push_back(1);
      int r = edges.size() - 1;
      // add edges to r and find p with link to q
      int p = last;
      while(p >= 0 && !edges[p].count(s[i])){
        edges[p][s[i]] = r;
        p = link[p];
      }
      if(p != -1) {
        int q = edges[p][s[i]];
        if(length[p] + 1 == length[q]) {
          // we do not have to split q, just set the
              correct suffix link
          link[r] = q;
        } else {
          // we have to split, add q'
          edges.push_back(edges[q]); // copy edges of q
          length.push_back(length[p] + 1);
          link.push_back(link[q]); // copy parent of q
          cnt.push_back(0);
          int qq = edges.size()-1;
          // add qq as the new parent of q and r
          link[q] = link[r] = qq; cnt[r] = 1;
          // move short classes pointing to q to point
              to q'
          while(p >= 0 && edges[p][s[i]] == q) {
            edges[p][s[i]] = qq;
            p = link[p];
          }
        }
      }
      last = r;
    }

    vector<int> ind(length.size());
    iota(ind.begin(), ind.end(), 0);
    sort(ind.begin(), ind.end(), [&](int i, int j){
      return length[i] > length[j];
    });
    for(auto i:ind) if(link[i] >= 0)
      cnt[link[i]] += cnt[i];
  }
};
```

5.2 Suffix array

```
vector<int> suffix_array(string &A){
  int n=A.size(),i=n, *M=new int[5*n];
  int *B=M,*C=M+n,*F=M+2*n,*G=M+3*n,*S=M+4*n;
  for(;i--;S[i]=n-i-1) B[i]=A[i];
  stable_sort(S,S+n,[&](int i,int j){return A[i]<A[j];})
      ;
  for(int L=1,p;L<n;L*=2){
    for(;++i<n;F[i]=B[S[i]],G[i]=B[S[i]+L/2]);
    for(;--i;F[i]=F[i]==F[i-1]&&G[i]==G[i-1]&&S[i-1]<n-L
        );
    for(p=B[*S]=0;++i<n;B[S[i]]=p=F[i]?p:i);
    for(fill_n(G,n,0);i--;F[i]=S[i]<L?-1:B[S[i]-L]);
    for(iota(C,C+n,0);++i<n;~F[i]?G[i]=C[F[i]]++:0);
    for(copy_n(S,n,F);i--;F[i]<L?0:S[G[i]]=F[i]-L);
  }
  vector<int>res(S,S+n);
  delete[] M;
  return res;
}
vector<int> kasai(string &s, vector<int> &sa){
  int n = s.size();
  vector<int> lcp(n),inv(n);
  for(int i=0;i<n;++i) inv[sa[i]] = i;
  for(int i=0,k=0;i<n;++i){
    if(k<0) k = 0;
    if(inv[i]==n-1){ k=0; continue; }
    for(int j=sa[inv[i]+1];max(i,j)+k<n&&s[i+k]==s[j+k
        ];++k);
    lcp[inv[i]] = k--;
  }
  return lcp;
}
```

5.3 Z Algorithm

```
vector<int> compute_Z(string s) {
  int n = s.length();
  vector<int> z(n, 0);
  z[0] = n;
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (r >= i) {
      z[i] = min(z[i - l], r - i + 1);
    }
    while (i+z[i] < n and s[i+z[i]] == s[z[i]]) {
      ++z[i];
    }
    if (i + z[i] - 1 > r) {
      r = i + z[i] - 1;
      l = i;
    }
  }
  return z;
}
```

5.4 KMP

```
vector<int> find_prefix(const vector<int> &P){
  int M = P.size();
  vector<int> pi(M);
  /* pi[i] <- largest prefix P[0..pi[i]] which is a
      suffix of P[0..i]
   * (but not equal to it) */
  pi[0] = -1;
  for (int i = 1, k = -1; i < M; ++i) {
    while(k > -1 && P[k + 1] != P[i])
      k = pi[k];
    if (P[k + 1] == P[i]) ++k;
    pi[i] = k;
  }
```

```
  return pi;
}
int kmp_matcher(const vector<int> &T, const vector<int>
    &P){
  int M = P.size(), N = T.size();
  vector<int> pi = find_prefix(P);
  int q = -1, matches = 0;
  for (int i = 0; i < N; ++i) {
    while(q > -1 && P[q + 1] != T[i])
      q = pi[q];
    if (P[q + 1] == T[i]) ++q;
    if (q == M - 1)
      ++matches, q = pi[q];
  }
  return matches;
}
```

5.5 String Hashing

```
struct hasher{
  int hashes[MAXN+5];
  int *pow, *inv;
  int mod;
  int n;
  void init(string &str, int *p, int *i, int m){
    pow = p;
    inv = i;
    mod = m;
    n = str.size();
    int last = 0;
    for (int i = 0; i < n; i++){
      int c = str[i] - 'a' + 1;
      last = (last + 1ll*c*pow[i]) % mod;
      hashes[i] = last;
    }
  }
  int getHash(int l, int r){
    if (r >= n || l < 0)
      return -1;
    int curr = hashes[r] - (l-1 >= 0 ? hashes[l-1] : 0);
    curr = ((curr % mod) + mod) % mod;
    curr = (1ll*curr*inv[l]) % mod;
    return curr;
  }
} A, B, C;
```

5.6 Palindrome DSU

```
// given an unknown string s and Q ranges that
// are know to be palindromes, this computes the
    characters
// that have to be equal in O(Q + n log n)
struct Palindrome{
  Palindrome(){}
  Palindrome(int n_):n(n_), m(3+__lg(n)), qs(m), p(2*n){
    iota(p.begin(), p.end(), 0);
  }
  int f(int i){
    return p[i] == i ? i : p[i] = f(p[i]);
  }
  void u(int a, int b){
    assert(0 <= a && a < 2*n);
    assert(0 <= b && b < 2*n);
    // union with splicing is a bit faster than
    // just path compression also guarantees p[i] <= i
    while(p[a] != p[b]){
      if(p[a] < p[b]) swap(a, b);
      if(p[a] == a){
        p[a] = b;
        return;
      }
      int tmp = p[a];
      p[a] = p[b];
      a = p[tmp];
    }
  }
  int components(){
    int ret = 0;
    for(int i=0;i<(int)p.size();++i)
      if(p[i] == i) ++ret;
    return ret;
  }
  // call this after adding all queries
  void compute(){
    vector<int> p2(2*n);
    for(int l=m-1; l>=0; --l){
      const int s = 1<<l;
      for(int i=0; i<2*n; ++i) p2[i] = f(i);
      for(int i=0; i+s<2*n; ++i){
        const int j = p2[i];
        if(j+s < 2*n) u(i+s, j+s);
      }
      for(auto const&e:qs[l])
        u(e.first, e.second);
    }
    // link point with mirror-image
    for(int i=0;i<n;++i)
      u(i, 2*n-1 - i);
  }
  // force [l, r] to be a palindrome
  void add_q(int l, int r){
    assert(0 <= l && l <= r && r < n);
    if(l==r) return;
    const int range = r-l+1;
    const int k = __lg(range);
    qs[k].emplace_back(l, 2*n-1 - r);
  }
  int n, m;
  vector<vector<pair<int, int> > > qs;
  vector<int> p;
};
```

5.7 Eertree

```
/*
Palindrome tree. Useful structure to deal with
    palindromes in strings. O(N)
This code counts no. of palindrome substrings of string.
     Based on problem 1750 from informatics.mccme.ru:
http://informatics.mccme.ru/moodle/mod/statements/view.
    php?chapterid =1750
*/
```

```
const int MAXN = 105000;

struct node {
    int next[26];
    int len;
    int sufflink;
    int num;
};
int len;
char s[MAXN];
node tree[MAXN];
int num;            // node 1 - root with len -1, node 2
    - root with len 0
int suff;           // max suffix palindrome
long long ans;

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

6 Data structures

6.1 BIT Range Queries

```
struct BIT {
  int n;
  vector<int> slope;
  vector<int> intercept;

  // BIT can be thought of as having entries f[1], ...,
      f[n] which are 0-initialized
  BIT(int n): n(n), slope(n+1), intercept(n+1) {}
  // returns f[1] + ... + f[idx-1]
  // precondition idx <= n+1

  int query(int idx) {
    int m = 0, b = 0;
    for (int i = idx-1; i > 0; i -= i&-i) {
      m += slope[i];
      b += intercept[i];
    }
    return m*idx + b;
  }

  // adds amt to f[i] for i in [idx1, idx2)
  // precondition 1 <= idx1 <= idx2 <= n+1 (you can't
      update element 0)
  void update(int idx1, int idx2, int amt) {
    for (int i = idx1; i <= n; i += i&-i) {
      slope[i] += amt;
      intercept[i] -= idx1*amt;
    }
    for (int i = idx2; i <= n; i += i&-i) {
      slope[i] -= amt;
      intercept[i] += idx2*amt;
    }
  }
};
```

6.2 Treaps

```
typedef struct node{
  int prior,size;
  int val;//value stored in the array
  int sum;//whatever info you want to maintain in
      segtree for each node
  int lazy;//whatever lazy update you want to do
  struct node *l,*r;
} node;

struct Treap {
  typedef node* pnode;
  int sz(pnode t){
    return t?t->size:0;
  }
```

```
void upd_sz(pnode t){
  if(t)t->size=sz(t->l)+1+sz(t->r);
}
void lazy(pnode t){
  if(!t || !t->lazy)return;
  t->val+=t->lazy;//operation of lazy
  t->sum+=t->lazy*sz(t);
  if(t->l)t->l->lazy+=t->lazy;//propagate lazy
  if(t->r)t->r->lazy+=t->lazy;
  t->lazy=0;
}
void reset(pnode t){
  if(t)t->sum = t->val;//no need to reset lazy coz
      when we call this lazy would itself be propagated
}
void combine(pnode& t,pnode l,pnode r){//combining two
     ranges of segtree
  if(!l || !r)return void(t = l?l:r);
  t->sum = l->sum + r->sum;
}
void operation(pnode t){//operation of segtree
  if(!t)return;
  reset(t);//reset the value of current node assuming
      it now represents a single element of the array
  lazy(t->l);lazy(t->r);//imp:propagate lazy before
      combining t->l,t->r;
  combine(t,t->l,t);
  combine(t,t,t->r);
}
void split(pnode t,pnode &l,pnode &r,int pos,int add
    =0){
  if(!t)return void(l=r=NULL);
  lazy(t);
  int curr_pos = add + sz(t->l);
  if(curr_pos<=pos)//element at pos goes to left
      subtree(l)
    split(t->r,t->r,r,pos,curr_pos+1),l=t;
  else
    split(t->l,l,t->l,pos,add),r=t;
  upd_sz(t);
  operation(t);
}
void merge(pnode &t,pnode l,pnode r){ //l->leftarray,r
    ->rightarray,t->resulting array
  lazy(l);lazy(r);
  if(!l || !r) t = l?l:r;
  else if(l->prior>r->prior)merge(l->r,l->r,r),t=l;
  else    merge(r->l,l,r->l),t=r;
  upd_sz(t);
  operation(t);
}
pnode init(int val){
  pnode ret = (pnode)malloc(sizeof(node));
  ret->prior=rand();ret->size=1;
  ret->val=val;
  ret->sum=val;ret->lazy=0;
  return ret;
}
int range_query(pnode t,int l,int r){//[l,r]
  pnode L,mid,R;
    split(t,L,mid,l-1);
    split(mid,t,R,r-l);//note: r-l!!
    int ans = t->sum;
    merge(mid,L,t);
    merge(t,mid,R);
    return ans;
  }
  void range_update(pnode t,int l,int r,int val){//[l,r]
    pnode L,mid,R;
    split(t,L,mid,l-1);
    split(mid,t,R,r-l);//note: r-l!!
    t->lazy+=val; //lazy_update
    merge(mid,L,t);
    merge(t,mid,R);
  }
};
```

6.3 Link-Cut Tree

```
const int MXN = 100005, MEM = 100005;
struct Splay {
  static Splay nil, mem[MEM], *pmem;
  Splay *ch[2], *f;
  int val, rev, size;
  Splay (int _val=-1) : val(_val), rev(0), size(1)
  { f = ch[0] = ch[1] = &nil; }
  bool isr()
  { return f->ch[0] != this && f->ch[1] != this; }
  int dir()
  { return f->ch[0] == this ? 0 : 1; }
  void setCh(Splay *c, int d){
    ch[d] = c;
    if (c != &nil) c->f = this;
    pull();
  }
  void push(){
    if( !rev ) return;
    swap(ch[0], ch[1]);
    if (ch[0] != &nil) ch[0]->rev ^= 1;
    if (ch[1] != &nil) ch[1]->rev ^= 1;
    rev=0;
  }
  void pull(){
    size = ch[0]->size + ch[1]->size + 1;
    if (ch[0] != &nil) ch[0]->f = this;
    if (ch[1] != &nil) ch[1]->f = this;
  }
} Splay::nil, Splay::mem[MEM], *Splay::pmem = Splay::mem
    ;
Splay *nil = &Splay::nil;
void rotate(Splay *x){
  Splay *p = x->f;
  int d = x->dir();
  if (!p->isr()) p->f->setCh(x, p->dir());
  else x->f = p->f;
        p->setCh(x->ch[!d], d); x->setCh(p, !d); p->pull
            (); x->pull();
}
vector<Splay*> splayVec;
void splay(Splay *x){
  splayVec.clear();
```

```
    for(Splay *q=x;; q=q->f){
      splayVec.push_back(q);
      if (q->isr()) break;
    }
    reverse(begin(splayVec), end(splayVec));
    for (auto it : splayVec) it->push();
    while (!x->isr())
      if (x->f->isr()) rotate(x);
      else if (x->dir()==x->f->dir()) rotate(x->f),rotate(
          x);
      else rotate(x),rotate(x);
  }
  int id(Splay *x) { return x - Splay::mem + 1; }
  Splay* access(Splay *x){
    Splay *q = nil;
    for (;x!=nil;x=x->f) {splay(x); x->setCh(q, 1); q = x;
        }
    return q;
  }
  void chroot(Splay *x){
    access(x); splay(x); x->rev ^= 1; x->push(); x->pull()
        ;
  }
  void link(Splay *x, Splay *y){
    access(x); splay(x);
    chroot(y); x->setCh(y, 1);
  }
  void cut_p(Splay *y) {
    access(y); splay(y); y->push(); y->ch[0] = y->ch[0]->f
        = nil;
  }
  void cut(Splay *x, Splay *y){
    chroot(x); cut_p(y);
  }
  Splay* get_root(Splay *x) {
    access(x); splay(x);
    for(; x->ch[0] != nil; x = x->ch[0]) x->push();
    splay(x); return x;
  }
  bool conn(Splay *x, Splay *y) {
    return (x = get_root(x)) == (y = get_root(y));
  }
  Splay* lca(Splay *x, Splay *y) {
    access(x); access(y); splay(x);
    if (x->f == nil) return x;
    else return x->f;
  }
```

7 Miscellaneous

7.1 2-SAT

```
  class Two_Sat {
    int N; // number of variables
    vector<int> val; // assignment of x is at val[2x] and
        -x at val[2x+1]
    vector<char> valid; // changes made at time i are kept
        iff valid[i]
    vector<vector<int> > G; // graph of implications G[x][
        i] = y means (x -> y)
    Two_Sat(int N_) : N(N_) { // create a formula over N
        variables (numbered 1 to N)
      G.resize(2*N);
    }
    int add_variable() {
      G.emplace_back(); G.emplace_back();
      return N++;
    }
  private:
    // converts a signed variable index to its position in
         val[] and G[]
    int to_ind(int x) {
      return 2*(abs(x)-1) + (x<0);
    }
    // Add a directed edge to the graph.
    // You most likely do not want to call this yourself!
    void add_edge(int a, int b) {
      G[to_ind(a)].push_back(to_ind(b));
    }
    int time() {
      return valid.size()-1;
    }
    bool dfs(int x) {
      if(valid[abs(val[x])]) return val[x]>0;
      val[x] = time();
      val[x^1] = -time();
      for(int e:G[x])
        if(!dfs(e))
          return false;
      return true;
    }
  public:
    // Add the or-clause: (a or b)
    void add_or(int a, int b) {
      add_edge(-a,b); add_edge(-b,a);
    }
    // Add the implication: a -> b
    void add_implication(int a, int b) {
      add_or(-a, b);
    }
    // Add condition: x is true
    void add_true(int x) {
      add_or(x,x);
    }
    // At most one with linear number of clauses
    template<typename T>
    void add_at_most_one(T vars) {
      if(vars.begin() == vars.end()) return;
      int last = *vars.begin(), cur = 0;
      for(int const&e:vars){
        if(e == last) continue;
        if(cur == 0) cur = e;
        else {
          add_or(-cur, -e);
          int new_cur = add_variable();
          cur = add_implication(cur, new_cur);
          add_implication(e, new_cur);
          cur = new_cur;
        }
      }
      if(cur != 0) add_or(-cur, -last);
```

```
  }
  bool solve() {
    val.assign(2*n, 0); valid.assign(1, 0);
    for(int i=0; i<val.size(); i+=2) {
      if(!valid[abs(val[i])]) {
        valid.push_back(1);
        if(!dfs(i)) {
          valid.back()=0;
          valid.push_back(1);
          if(!dfs(i+1)) return false;
        }
      }
    }
    return true;
  }
};
// Taken from https://github.com/dacin21/
    dacin21_codebook/blob/master/dfs_stuff/2sat.cpp
// 2-sat in linear time via backtracking.
class Two_Sat {
    int N; // number of variables
    vector<int> val; // assignment of x is at val[2x]
        and -x at val[2x+1]
    vector<char> valid; // changes made at time i are
        kept iff valid[i]
    vector<vector<int> > G; // graph of implications G[x
        ][i] = y means (x -> y)

    Two_Sat(int N_) : N(N_) { // create a formula over N
         variables (numbered 1 to N)
        G.resize(2*N);
    }

    int add_variable() {
        G.emplace_back();
        G.emplace_back();
        return N++;
    }

private:
    // converts a signed variable index to its position
        in val[] and G[]
    int to_ind(int x) {
        return 2*(abs(x)-1) + (x<0);
    }

    // Add a directed edge to the graph.
    // You most likely do not want to call this yourself
        !
    void add_edge(int a, int b) {
        G[to_ind(a)].push_back(to_ind(b));
    }

    int time() {
        return valid.size()-1;
    }

    bool dfs(int x) {
        if(valid[abs(val[x])]) return val[x]>0;
        val[x] = time();
        val[x^1] = -time();
        for(int e:G[x])
            if(!dfs(e))
                return false;
        return true;
    }
public:
    // Add the or-clause: (a or b)
    void add_or(int a, int b) {
        add_edge(-a,b);
        add_edge(-b,a);
    }

    // Add the implication: a -> b
    void add_implication(int a, int b) {
        add_or(-a, b);
    }

    // Add condition: x is true
    void add_true(int x) {
        add_or(x,x);
    }

    // At most one with linear number of clauses
    template<typename T>
    void add_at_most_one(T vars) {
        if(vars.begin() == vars.end()) return;
        int last = *vars.begin();
        int cur = 0;
        for(int const&e:vars){
            if(e == last) continue;
            if(cur == 0) cur = e;
            else {
                add_or(-cur, -e);
                int new_cur = add_variable();
                cur = add_implication(cur, new_cur);
                add_implication(e, new_cur);
                cur = new_cur;
            }
        }
        if(cur != 0){
            add_or(-cur, -last);
        }
    }

    bool solve() {
        val.assign(2*n, 0);
        valid.assign(1, 0);
        for(int i=0; i<val.size(); i+=2) {
            if(!valid[abs(val[i])]) {
                valid.push_back(1);
                if(!dfs(i)) {
                    valid.back()=0;
                    valid.push_back(1);
                    if(!dfs(i+1)) return false;
                }
            }
        }
        return true;
    }
};
```

7.2 Merge Insertion

```
// Sorting in O(n^2) time with near-optimal number of
    comparisons
```

```cpp
// Number of comparisons used is: n lg n - 1.415 n
// The lower bound is: lg (n!) = n lg n - 1.443 n
// Binary search insertion sort would need: n log n - n
struct Merge_Insertion_Sort{
  template<typename F>
  static void apply_permutation(vector<int> const&p, F
      get){
    const int n = p.size();
    vector<int> q(n);
    for(int i=0;i<n;++i) q[p[i]] = i;
    for(int i=0;i<n;++i){
      while(q[i] != i){
        swap(get(i), get(q[i]));
        swap(q[i], q[q[i]]);
      }
    }
  }
  // ret.first is the sorted vector ret.second[i] is the
  // index of the i-th smallest element in the original
      vector
  // i.e. ret.second is the permutation that was applied
      to sort
  template<typename T, typename F>
  static pair<vector<T>, vector<int> > sort(vector<T> v,
      F comp){
    const int n = v.size();
    if(n <= 1) return {move(v), {{0}}};
    vector<int> preperm(n);
    iota(preperm.begin(), preperm.end(), 0);
    const int M = n-n/2;
    for(int i=0;i<n/2;++i){
      if(comp(v[M+i], v[i])){
        swap(v[M+i], v[i]);
        swap(preperm[M+i], preperm[i]);
      }
    }
    auto ret=sort(vector<T>(v.begin(),v.begin()+n/2),
        comp);
    apply_permutation(ret.second,[&preperm,M](int const&
        i)->int&{return preperm[M+i];});
    apply_permutation(ret.second,[&preperm](int const&i)
        ->int&{return preperm[i];});
    apply_permutation(ret.second,[&v,M](int const&i)->T
        &{return v[M+i];});
    iota(ret.second.begin(), ret.second.end(), 0);
    // insert one element without comparisons
    ret.first.push_back(v.back());
    ret.second.push_back(n-1);
    // now insert the rest in blocks that optimize the
        binary search sizes
    for(int it=1,r=n-1,s=2; r>n/2; ++it, r-=s, s=(1<<it)
        -s){
      for(int i=r-s;i<r;++i){
        if(i>=n/2){
          int a = find(ret.second.begin(), ret.second.
              end(), i-M) - ret.second.begin();
          int b = ret.first.size();
          if(a==b) {
            assert(i==M-1);
            a = -1;
          }
          while(a+1 < b){
            const int m = a+(b-a)/2;
            if(comp(ret.first[m], v[i])) a = m;
            else b = m;
          }
          ret.first.insert(ret.first.begin()+b, v[i]);
          ret.second.insert(ret.second.begin()+b, i);
        }
      }
    }
    // compose permutations
    apply_permutation(ret.second, [&preperm](int const&i
        )->int&{return preperm[i];});
    ret.second.swap(preperm);
    return ret;
  }
  template<typename T>
  static pair<vector<T>, vector<int> > sort(vector<T> v)
      {
    return sort(move(v), std::less<T>{});
  }
};
```

7.3 DP Optimizations

```
A[i][j] : The smallest k that gives optimal answer

Divide and Conquer:
  dp[i][j] = min(k < j){dp[i - 1][k] + C[k][j]}
  O(kn^2) -> O(knlog(n))

  Conditions:
    A[i][j] <= A[i][j + 1] OR
    C[a][d] + C[b][c] >= C[a][c] + C[b][d] where a < b <
        c < d

  Short Description:
    A[i][1] <= A[i][2] <= ... <= A[i][n]

Knuth Optimization:
  dp[i][j] = min(i < k < j){dp[i][k] + dp[k][j]} + C[i][
      j]
  O(n^3) -> O(n^2)

  Conditions:
    A[i, j - 1] <= A[i, j] <= A[i + 1, j] OR
    C[a][d] + C[b][c] >= C[a][c] + C[b][d] AND
    C[b][c] <= C[a][d] where a <= b <= c <= d

  Short Description:
    For dp[i][j], loop k from A[i][j - 1] to A[i + 1][j]
```

7.4 Convex Hull Trick (Dynamic)

```cpp
struct Line {
  long long m, b;
  mutable function<const Line*()> succ;
  bool operator<(const Line& rhs) const{
    if(rhs.b!=-(1ll<<62)) return m>rhs.m; // < for max
    const Line* s = succ();
    if (!s) return 0;
    return b-s->b > (s->m -m)*rhs.m; // < for max
  }
};
```

```
struct HullDynamic : public multiset<Line> {
  bool bad(iterator y) {
    auto z = next(y);
    if(y==begin()){
      if(z==end())return 0;
      return y->m == z->m && y->b >= z->b; // <= for max
    }
    auto x = prev(y);
    if (z==end()) return y->m == x->m && y->b >= x->b;
        // <= for max
    return (x->b - y->b)*1.0*(z->m - y->m) >= (y->b - z
        ->b)*1.0*(y->m - x->m);
  }
  void insert_line(long long m, long long b) {
    auto y = insert({ m, b });
    y->succ = [=]{return next(y)==end()? 0:&*next(y);};
    if(bad(y)) { erase(y); return; }
    while(next(y)!=end() && bad(next(y)))erase(next(y));
    while(y!=begin() && bad(prev(y)))erase(prev(y));
  }
  long long eval(long long x) {
    auto l = *lower_bound((Line){x,-(1ll<<62)});
    return l.m * x + l.b;
  }
};
```

7.5 Convex Hull Trick (Static)

```
struct ConvexHullTrick {
  typedef long long LL;
  vector<LL> M;
  vector<LL> B;
  vector<double> left;
  ConvexHullTrick() {}
  bool bad(LL m1, LL b1, LL m2, LL b2, LL m3, LL b3) {
    // Careful, this may overflow
    return (b3-b1)*(m1-m2) < (b2-b1)*(m1-m3);
  }
  // Add a new line to the structure, y = mx + b.
  // Lines must be added in decreasing order of slope.
  void add(LL m, LL b) {
    while (M.size() >= 2 && bad(M[M.size()-2], B[B.size
        ()-2], M.back(), B.back(), m, b)) {
      M.pop_back(); B.pop_back(); left.pop_back();
    }
    if (M.size() && M.back() == m) {
      if (B.back() > b) {
        M.pop_back(); B.pop_back(); left.pop_back();
      } else {
        return;
      }
    }
    if (M.size() == 0) {
      left.push_back(-numeric_limits<double>::infinity()
          );
    } else {
      left.push_back((double)(b - B.back())/(M.back() -
          m));
    }
    M.push_back(m);
    B.push_back(b);
  }
  // Get the minimum value of mx + b among all lines in
      the structure.
  // There must be at least one line.
  LL query(LL x) {
    int i = upper_bound(left.begin(), left.end(), x) -
        left.begin();
    return M[i-1]*x + B[i-1];
  }
};
```

7.6 BigInt library

```
struct bignum {
  typedef unsigned int uint;
  vector<uint> digits;
  static const uint RADIX = 1000000000;
  bignum(): digits(1, 0) {}
  bignum(const bignum& x): digits(x.digits) {}
  bignum(unsigned long long x) {*this = x;}
  bignum(const char* x) {*this = x;}
  bignum(const string& s) {*this = s;}
  bignum& operator=(const bignum& y)
    {digits = y.digits; return *this;}
  bignum& operator=(unsigned long long x){
    digits.assign(1, x%RADIX);
    if (x >= RADIX)
      digits.push_back(x/RADIX);
    return *this;
  }
  bignum& operator=(const char* s) {
    int slen=strlen(s),i,l;
    digits.resize((slen+8)/9);
    for (l=0; slen>0; l++,slen-=9) {
      digits[l]=0;
      for (i=slen>9?slen-9:0; i<slen; i++)
        digits[l]=10*digits[l]+s[i]-'0';
    }
    while (digits.size() > 1 && !digits.back()) digits.
        pop_back();
    return *this;
  }
  bignum& operator=(const string& s)
    {return *this = s.c_str();}

  void add(const bignum& x) {
    int l = max(digits.size(), x.digits.size());
    digits.resize(l+1);
    for (int d=0, carry=0; d<=l; d++) {
      uint sum=carry;
      if (d<digits.size()) sum+=digits[d];
      if (d<x.digits.size()) sum+=x.digits[d];
      digits[d]=sum;
      if (digits[d]>=RADIX)
        digits[d]-=RADIX, carry=1;
      else
        carry=0;
    }
    if (!digits.back()) digits.pop_back();
  }
  void sub(const bignum& x) {
```

```
    // if ((*this)<x) throw; //negative numbers not yet
        supported
    for (int d=0, borrow=0; d<digits.size(); d++) {
      digits[d]-=borrow;
      if (d<x.digits.size()) digits[d]-=x.digits[d];
      if (digits[d]>>31) { digits[d]+=RADIX; borrow=1; }
          else borrow=0;
    }
    while (digits.size() > 1 && !digits.back()) digits.
        pop_back();
  }
  void mult(const bignum& x) {
    vector<uint> res(digits.size() + x.digits.size());
    unsigned long long y,z;
    for (int i=0; i<digits.size(); i++) {
      for (int j=0; j<x.digits.size(); j++) {
        unsigned long long y=digits[i]; y*=x.digits[j];
        unsigned long long z=y/RADIX;
        res[i+j+1]+=z; res[i+j]+=y-RADIX*z; //mod is
            slow
        if (res[i+j] >= RADIX) { res[i+j] -= RADIX; res[
            i+j+1]++; }
        for (int k = i+j+1; res[k] >= RADIX; res[k] -=
            RADIX, res[++k]++);
      }
    }
    digits = res;
    while (digits.size() > 1 && !digits.back()) digits.
        pop_back();
  }
  // returns the remainder
  bignum div(const bignum& x) {
    bignum dividend(*this);
    bignum divisor(x);
    fill(digits.begin(), digits.end(), 0);
    // shift divisor up
    int pwr = dividend.digits.size() - divisor.digits.
        size();
    if (pwr > 0) {
      divisor.digits.insert(divisor.digits.begin(), pwr,
          0);
    }
    while (pwr >= 0) {
      if (dividend.digits.size() > divisor.digits.size()
          ) {
        unsigned long long q = dividend.digits.back();
        q *= RADIX; q += dividend.digits[dividend.digits
            .size()-2];
        q /= 1+divisor.digits.back();
        dividend -= divisor*q; digits[pwr] = q;
        if (dividend >= divisor) { digits[pwr]++;
            dividend -= divisor; }
        assert(dividend.digits.size() <= divisor.digits.
            size()); continue;
      }
      while (dividend.digits.size() == divisor.digits.
          size()) {
        uint q = dividend.digits.back() / (1+divisor.
            digits.back());
        if (q == 0) break;
        digits[pwr] += q; dividend -= divisor*q;
      }
      if (dividend >= divisor) { dividend -= divisor;
          digits[pwr]++; }
      pwr--; divisor.digits.erase(divisor.digits.begin()
          );
    }
    while (digits.size() > 1 && !digits.back()) digits.
        pop_back();
    return dividend;
  }
  string to_string() const {
    ostringstream oss;
    oss << digits.back();
    for (int i = digits.size() - 2; i >= 0; i--) {
      oss << setfill('0') << setw(9) << digits[i];
    }
    return oss.str();
  }
  bignum operator+(const bignum& y) const
    {bignum res(*this); res.add(y); return res;}
  bignum operator-(const bignum& y) const
    {bignum res(*this); res.sub(y); return res;}
  bignum operator*(const bignum& y) const
    {bignum res(*this); res.mult(y); return res;}
  bignum operator/(const bignum& y) const
    {bignum res(*this); res.div(y); return res;}
  bignum operator%(const bignum& y) const
    {bignum res(*this); return res.div(y);}
  bignum& operator+=(const bignum& y)
    {add(y); return *this;}
  bignum& operator-=(const bignum& y)
    {sub(y); return *this;}
  bignum& operator*=(const bignum& y)
    {mult(y); return *this;}
  bignum& operator/=(const bignum& y)
    {div(y); return *this;}
  bignum& operator%=(const bignum& y)
    {*this = div(y);}
  bool operator==(const bignum& y)
    {return digits == y.digits;}
  bool operator<(const bignum& y) const {
    if (digits.size() < y.digits.size()) return true;
    if (digits.size() > y.digits.size()) return false;
    for (int i = digits.size()-1; i >= 0; i--)
      if (digits[i] < y.digits[i])
        return true;
      else if (digits[i] > y.digits[i])
        return false;
    return false;
  }
  bool operator>(const bignum& y) const
    {return y<*this;}
  bool operator<=(const bignum& y) const
    {return !(y<*this);}
  bool operator>=(const bignum& y) const
    {return !(*this<y);}
};
```

7.7 Manachers algorithm

```
// Ex: "opposes" -> [0,1,0,1,4,1,0,1,0,1,0,3,0,1,0]
vector<int> fastLongestPalindromes(string str) {
  int i=0,j,d,s,e,lLen,palLen=0;
  vector<int> res;
  while (i < str.length()) {
    if (i > palLen && str[i-palLen-1] == str[i]) {
      palLen += 2; i++; continue;
    }
    res.push_back(palLen);
    s = res.size()-2;
    e = s-palLen;
    bool b = true;
    for (j=s; j>e; j--) {
      d = j-e-1;
      if (res[j] == d) { palLen = d; b = false; break; }
      res.push_back(min(d, res[j]));
    }
    if (b) { palLen = 1; i++; }
  }
  res.push_back(palLen);
  lLen = res.size();
  s = lLen-2;
  e = s-(2*str.length()+1-lLen);
  for (i=s; i>e; i--) { d = i-e-1; res.push_back(min(d,
     res[i])); }
  return res;
}
```

7.8 Dates

```
// Months are expressed as integers from 1 to 12, Days
    are expressed as integers from 1 to 31, and Years are
     expressed as 4-digit integers.
string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri",
    "Sat", "Sun"};

//converts Gregorian date to integer(Julian day number)
int dateToInt (int m, int d, int y){
  return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075;
}

// converts integer (Julian day number) to Gregorian
    date: month/day/year
void intToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;
  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to day of week
string intToDay (int jd){
  return dayOfWeek[jd % 7];
}
```

7.9 Bitset (Text)

```
Remember _Find_first() and _Find_next() ex- ist, and run
     in O(N/W), where W is word size of machine.
```

7.10 Template

```
g++ -std=c++17 -DLOCAL -O2 -Wall -Wshadow -Wextra -
    pedantic -Wfloat-equal -Wlogical-op

#pragma comment(linker, "/stack:200000000")
#pragma GCC optimize("Ofast")
#pragma GCC optimize ("unroll-loops")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,
    mmx,avx,tune=native") // codeforces
//#pragma GCC target("avx,avx2,fma")
//#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,
    abm,mmx,tune=native") // yandex

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <bits/stdc++.h>

using namespace __gnu_pbds;
using namespace std;

typedef long double ld;
typedef long long ll;
typedef pair <int, int> pii;
typedef pair <ll, ll> pll;

mt19937 rng(std::chrono::duration_cast<std::chrono::
    nanoseconds>(chrono::high_resolution_clock::now().
    time_since_epoch()).count());

template<typename has_less>
using ordered_set =
tree<has_less,
        null_type,
        less<has_less>,
        rb_tree_tag,
        tree_order_statistics_node_update>;

//insert using pref_trie.insert
//get range for prefix using pref_trie.prefix_range
//use iterator from range.first until != range.second
typedef
trie<string,
        null_type,
        trie_string_access_traits<>,
        pat_trie_tag,
        trie_prefix_search_node_update>
        pref_trie;

struct chash {
    int operator()(int x) const { return x ^ RANDOM; }
};
gp_hash_table<key, int, chash> table;

int main(){}
```

7.11 Numerical Integration

```cpp
// different schemes for numerical integration
// approximatively ordered by accuracy
// do NOT use integer types for integration range!
struct Integration_Midpoint{
  template<typename Func, typename S>
  static typename result_of<Func(S)>::type
     integrate_step(Func f, S l, S r){
    S m = (l+r)/2;
    return f(m) * (r-l);
  }
};
struct Integration_Simpson{
  template<typename Func, typename S>
  static typename result_of<Func(S)>::type
     integrate_step(Func f, S l, S r){
    S m = (l+r)/2;
    return (f(l) + 4*f(m) + f(r))/6 * (r-l);
  }
};
struct Integration_Gauss_2{
  static constexpr long double A = 1.0l/sqrtl(3)/2, x1
     =0.5l-A, x2 = 0.5l+A;
  template<typename Func, typename S>
  static typename result_of<Func(S)>::type
     integrate_step(Func f, S l, S r){
    return (f(l*x1 + r*x2) + f(l*x2+r*x1))/2 * (r-l);
  }
};
struct Integration_NCotes_Open_4{
  template<typename Func, typename S>
  static typename result_of<Func(S)>::type
     integrate_step(Func f, S l, S r){
    S h = (r-l)/5;
    return (11*f(l+h) + f(l+2*h) + f(r-2*h) + 11*f(r-h))
       /24 * (r-l);
  }
};
struct Integration_Gauss_3{
  static constexpr long double A = sqrtl(3.0l/5.0l)/2,
     x1=0.5-A, x2 = 0.5+A;
  template<typename Func, typename S>
  static typename result_of<Func(S)>::type
     integrate_step(Func f, S l, S r){
    return (5*f(l*x1 + r*x2) + 8*f((l+r)/2) + 5*f(l*x2+r
       *x1))/18 * (r-l);
  }
};

template<typename Integration_Method>
struct Integrator_Fixedstep{
  template<typename Func, typename S>
  static typename result_of<Func(S)>::type integrate(
     Func f, S const l, S const r, size_t const steps){
    assert(steps>0);
    typename result_of<Func(S)>::type ret(0);
    S cur_l = l, cur_r;
    for(size_t i=0;i<steps;++i){
      cur_r = (l*(steps-i-1) + r*(i+1))/steps;
      ret+=Integration_Method::integrate_step(f, cur_l,
         cur_r);
      cur_l = cur_r;
    }
    return ret;
  }
};
template<typename Integration_Method>
class Integrator_Adaptive{
private:
  template<size_t depth_limit, typename Func, typename S
     >
  static typename result_of<Func(S)>::type integrate(
     Func f, S const l, S const r, typename result_of<
     Func(S)>::type const val, typename result_of<Func(S
     )>::type const eps, const size_t depth){
    if(depth>=depth_limit){
      return val;
    }
    S const m = (l+r)/2;
    typename result_of<Func(S)>::type val_l =
       Integration_Method::integrate_step(f, l, m);
    typename result_of<Func(S)>::type val_r =
       Integration_Method::integrate_step(f, m, r);
    typename result_of<Func(S)>::type error = abs(val -
       val_l - val_r);
    if(error < eps){
      return val_l + val_r;
    }
    return integrate<depth_limit>(f, l, m, val_l, eps/2,
       depth+1)
      + integrate<depth_limit>(f, m, r, val_r, eps/2,
         depth+1);
  }
public:
  template<size_t depth_limit, typename Func, typename S
     >
  static typename result_of<Func(S)>::type integrate(
     Func f, S const l, S const r, typename result_of<
     Func(S)>::type const eps){
    return integrate<depth_limit>(f, l, r,
       Integration_Method::integrate_step(f, l, r), eps,
       0);
  }
};
```