Contents

1 Combinatorial optimization

1.1 Dinic's

```
// Reference: My (forked) ICPC codebook: https://github.
    com/aayush9/ICPC-CodeBook/blob/master/code/Dinic.cc
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 max-flow

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

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

using namespace std;

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

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

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

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

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

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

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

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

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

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

// BEGIN CUT
// The following code solves UVA problem #10594: Data
    Flow

int main() {
  int N, M;

  while (scanf("%d%d", &N, &M) == 2) {
    VVL v(M, VL(3));
    for (int i = 0; i < M; i++)
      scanf("%Ld%Ld%Ld", &v[i][0], &v[i][1], &v[i][2]);
    L D, K;
    scanf("%Ld%Ld", &D, &K);
```

```
    MinCostMaxFlow mcmf(N+1);
    for (int i = 0; i < M; i++) {
      mcmf.AddEdge(int(v[i][0]), int(v[i][1]), K, v[i
          ][2]);
      mcmf.AddEdge(int(v[i][1]), int(v[i][0]), K, v[i
          ][2]);
    }
    mcmf.AddEdge(0, 1, D, 0);

    pair<L, L> res = mcmf.GetMaxFlow(0, N);

    if (res.first == D) {
      printf("%Ld\n", res.second);
    } else {
      printf("Impossible.\n");
    }
  }

  return 0;
}

// END CUT
```

1.3 Edmonds Max Matching

```
/*
GETS:
V->number of vertices
E->number of edges
pair of vertices as edges (vertices are 1..V)

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

#include <bits/stdc++.h>
using namespace std;
const int M=505;
struct struct_edge{int v;struct_edge* n;};
typedef struct_edge* edge;
struct_edge pool[M*M*2];
edge top=pool,adj[M];
int V,E,match[M],qh,qt,q[M],father[M],base[M];
bool inq[M],inb[M],ed[M][M];
void add_edge(int u,int v)
{
  top->v=v,top->n=adj[u],adj[u]=top++;
  top->v=u,top->n=adj[v],adj[v]=top++;
}
int LCA(int root,int u,int v)
{
  static bool inp[M];
  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<V;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<V;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->n)
        {
    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<V;u++)
    if (match[u]==-1)
      matchc+=augment_path(u,find_augmenting_path(u));
  return matchc;
}
int main()
{
  int u,v;
  cin>>V>>E;
  while(E--)
    {
      cin>>u>>v;
      if (!ed[u-1][v-1])
  {
    add_edge(u-1,v-1);
    ed[u-1][v-1]=ed[v-1][u-1]=true;
  }
    }
  cout<<edmonds()<<endl;
  for (int i=0;i<V;i++)
    if (i<match[i])
      cout<<i+1<<" "<<match[i]+1<<endl;
}
```

1.4 Max bipartite matching

```
// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in
    practice
//
//   INPUT: w[i][j] = edge between row node i and column
     node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if
    unassigned
//           mc[j] = assignment for column node j, -1 if
     unassigned
//           function returns number of matches made

#include <vector>

using namespace std;

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

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &
    seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)
          ) {
        mr[i] = j;
        mc[j] = i;
        return true;
      }
    }
  }
  return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
  mr = VI(w.size(), -1);
  mc = VI(w[0].size(), -1);

  int ct = 0;
  for (int i = 0; i < w.size(); i++) {
    VI seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

1.5 Global min-cut

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

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

using namespace std;

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

const int INF = 1000000000;

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

  for (int phase = N-1; phase >= 0; phase--) {
    VI w = weights[0];
    VI added = used;
    int prev, last = 0;
```

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

  // BEGIN CUT
  // The following code solves UVA problem #10989: Bomb,
      Divide and Conquer
  int main() {
    int N;
    cin >> N;
    for (int i = 0; i < N; i++) {
      int n, m;
      cin >> n >> m;
      VVI weights(n, VI(n));
      for (int j = 0; j < m; j++) {
        int a, b, c;
        cin >> a >> b >> c;
        weights[a-1][b-1] = weights[b-1][a-1] = c;
      }
      pair<int, VI> res = GetMinCut(weights);
      cout << "Case #" << i+1 << ": " << res.first << endl
          ;
    }
  }
  // END CUT
```

2 Geometry

2.1 Convex hull

```
typedef pair<long long, long long> PT;
long double dist(PT a, PT b){
  return sqrt(pow(a.first-b.first,2)+pow(a.second-b.
      second,2));
}
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;
}
vector<PT> convexhull(){
  vector<PT> hull;
  sort(a,a+n,[](PT i, PT j){
    if(i.second!=j.second)
      return i.second < j.second;
    return i.first < j.first;
  });
  for(int i=0;i<n;++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=n-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;
}
```

2.2 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;
}

int main() {

  cerr << RotateCCW90(PT(2,5)) << endl;

  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;

  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;

  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7));

  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

  cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) ;

  cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) ;

  cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) ;

  cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;

  cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

  vector<PT> v;
  v.push_back(PT(0,0));
  v.push_back(PT(5,0));
  v.push_back(PT(5,5));
  v.push_back(PT(0,5));

  cerr << PointInPolygon(v, PT(2,2)) ;

  cerr << PointOnPolygon(v, PT(2,2))
  vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

  u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
  for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

  PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
  vector<PT> p(pa, pa+4);
  PT c = ComputeCentroid(p);
  cerr << "Area: " << ComputeArea(p) << endl;
  cerr << "Centroid: " << c << endl;
}
```

2.3 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));
  }
}
```

2.4 Slow Delaunay triangulation

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

#include<vector>
using namespace std;

typedef double T;

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

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

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

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

```
}

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

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

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

3 Numerical algorithms

3.1 Number theory (modular, Chinese remainder, linear Diophantine)

```
// This is a collection of useful code for solving
   problems that
// involve modular linear equations.  Note that all of
   the
// algorithms described here work on nonnegative
   integers.

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

using namespace std;

typedef vector<int> VI;
typedef pair<int, int> PII;

// return a % b (positive value)
int mod(int a, int b) {
        return ((a%b) + b) % b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
        while (b) { int t = a%b; a = b; b = t; }
        return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
        return a / gcd(a, b)*b;
}

// (a^b) mod m via successive squaring
int powermod(int a, int b, int m)
{
        int ret = 1;
        while (b)
        {
                if (b & 1) ret = mod(ret*a, m);
                a = mod(a*a, m);
                b >>= 1;
        }
        return ret;
}

// 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);
}

// Chinese remainder theorem: find z such that
// z % m[i] = r[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (m[i]).  Return (z, M). On
// failure, M = -1. Note that we do not require the a[i
   ]'s
// to be relatively prime.
```

```
PII chinese_remainder_theorem(const VI &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;
}

int main() {
        // expected: 2
        cout << gcd(14, 30) << endl;

        // expected: 2 -2 1
        int x, y;
        int g = extended_euclid(14, 30, x, y);
        cout << g << " " << x << " " << y << endl;

        // expected: 95 451
        VI sols = modular_linear_equation_solver(14, 30,
             100);
        for (int i = 0; i < sols.size(); i++) cout <<
            sols[i] << " ";
        cout << endl;

        // expected: 8
        cout << mod_inverse(8, 9) << endl;

        // expected: 23 105
        //           11 12
        PII ret = chinese_remainder_theorem(VI({ 3, 5, 7
             }), VI({ 2, 3, 2 }));
        cout << ret.first << " " << ret.second << endl;
        ret = chinese_remainder_theorem(VI({ 4, 6 }), VI
            ({ 3, 5 }));
        cout << ret.first << " " << ret.second << endl;

        // expected: 5 -15
        if (!linear_diophantine(7, 2, 5, x, y)) cout <<
            "ERROR" << endl;
        cout << x << " " << y << endl;
        return 0;
}
```

3.2 Systems of linear equations, matrix inverse, determinant

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

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

using namespace std;

const double EPS = 1e-10;

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

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

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

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

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

    return det;
  }

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

    double det = GaussJordan(a, b);

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

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

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

3.3 Reduced row echelon form, matrix rank

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

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

using namespace std;

const double EPSILON = 1e-10;

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

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

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

int main() {
  const int n = 5, m = 4;
  double A[n][m] = {
    {16,  2,  3, 13},
    { 5, 11, 10,  8},
    { 9,  7,  6, 12},
    { 4, 14, 15,  1},
    {13, 21, 21, 13}};
  VVT a(n);
  for (int i = 0; i < n; i++)
    a[i] = VT(A[i], A[i] + m);

  int rank = rref(a);
```

```
    // expected: 3
    cout << "Rank: " << rank << endl;

    // expected: 1 0 0 1
    //           0 1 0 3
    //           0 0 1 -3
    //           0 0 0 3.10862e-15
    //           0 0 0 2.22045e-15
    cout << "rref: " << endl;
    for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 4; j++)
        cout << a[i][j] << ' ';
      cout << endl;
    }
  }
```

3.4 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).

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

using namespace std;

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

const DOUBLE EPS = 1e-9;

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

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

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

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

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n
        + 1]) r = i;
    if (D[r][n + 1] < -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return
          -numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] ==
              D[i][s] && N[j] < N[s]) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::
```

```
      infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] =
        D[i][n + 1];
    return D[m][n + 1];
  }
};
int main() {
  const int m = 4;
  const int n = 3;
  DOUBLE _A[m][n] = {
    { 6, -1, 0 },
    { -1, -5, 0 },
    { 1, 5, 1 },
    { -1, -5, -1 }
  };
  DOUBLE _b[m] = { 10, -4, 5, -5 };
  DOUBLE _c[n] = { 1, -1, 0 };
  VVD A(m);
  VD b(_b, _b + m);
  VD c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n
      );
  LPSolver solver(A, b, c);
  VD x;
  DOUBLE value = solver.Solve(x);
  cerr << "VALUE: " << value << endl; // VALUE: 1.29032
  cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x
      [i];
  cerr << endl;
  return 0;
}
```

3.5 Fast Fourier transform (C++)

```
// Source: My (forked) ICPC codebook: https://github.com
    /aayush9/ICPC-CodeBook/blob/master/code/FFT.cc
namespace FFT{
  vector<complex<long double>> A;
  int n,L;
  int ReverseBits(int k){
    int ret = 0;
    for(int i=0;i++<L;k/=2)
      ret = (ret<<1)|(k&1);
    return ret;
  }
  void BitReverseCopy(vector<complex<long double>> a){
    for (n=1,L=0;n<a.size();n<<=1,L++);
    A.resize(n);
    for (int k = 0; k < n; k++)
      A[ReverseBits(k)] = a[k];
  }
  vector<complex<long double>> DFT(vector<complex<long
      double>> a, bool inverse) {
    BitReverseCopy(a);
    for (int s = 1; s <= L; s++) {
      int m = 1 << s;
      complex<long double> wm = exp(complex<long double
          >(0, 2.0 * M_PI / m));
      if (inverse) wm = complex<long double>(1, 0) / wm;
      for (int k = 0; k < n; k += m) {
        complex<long double> w = 1;
        for (int j = 0; j < m/2; j++) {
          complex<long double> t = w * A[k + j + m/2];
          complex<long double> u = A[k + j];
          A[k + j] = u + t;
          A[k + j + m/2] = u - t;
          w = w * wm;
        }
      }
    }
    if (inverse) for (int i = 0; i < n; i++) A[i] /= n
        ;
    return A;
  }
  vector<long double> Convolution(vector<long double> a,
      vector<long double> b) {
    int L = 1;
    while ((1 << L) < a.size()) L++;
    while ((1 << L) < b.size()) L++;
    int n = 1 << (L+1);
    vector<complex<long double>> aa, bb;
    for (size_t i = 0; i < n; i++) aa.push_back(i < a.
        size() ? complex<long double>(a[i], 0) : 0);
    for (size_t i = 0; i < n; i++) bb.push_back(i < b.
        size() ? complex<long double>(b[i], 0) : 0);
    vector<complex<long double>> AA = DFT(aa, false);
    vector<complex<long double>> BB = DFT(bb, false);
    vector<complex<long double>> CC;
    for (size_t i = 0; i < AA.size(); i++) CC.push_back(
        AA[i] * BB[i]);
    vector<complex<long double>> cc = DFT(CC, true);
    vector<long double> c;
    for (int i=0;i<a.size()+b.size()-1;++i)
      c.push_back(cc[i].real());
    return c;
  }
}
```

4 Graph algorithms

4.1 Bellman-Ford shortest paths with negative edge weights

```
// This function runs the Bellman-Ford algorithm for
    single source
// shortest paths with negative edge weights.  The
    function returns
// false if a negative weight cycle is detected.
    Otherwise, the
// function returns true and dist[i] is the length of
    the shortest
// path from start to i.
```

```
//
// Running time: O(|V|^3)
//
//   INPUT:   start, w[i][j] = cost of edge from i to j
//   OUTPUT:  dist[i] = min weight path from start to i
//            prev[i] = previous node on the best path
   from the
//                     start node

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

using namespace std;

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

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

bool BellmanFord (const VVT &w, VT &dist, VI &prev, int
   start){
  int n = w.size();
  prev = VI(n, -1);
  dist = VT(n, 1000000000);
  dist[start] = 0;

  for (int k = 0; k < n; k++){
    for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
        if (dist[j] > dist[i] + w[i][j]){
          if (k == n-1) return false;
          dist[j] = dist[i] + w[i][j];
          prev[j] = i;
        }
      }
    }
  }

  return true;
}
```

4.2 Strongly connected components

```
#include<memory.h>
struct edge{int e, nxt;};
int V, E;
edge e[MAXE], er[MAXE];
int sp[MAXV], spr[MAXV];
int group_cnt, group_num[MAXV];
bool v[MAXV];
int stk[MAXV];
void fill_forward(int x)
{
  int i;
  v[x]=true;
  for(i=sp[x];i;i=e[i].nxt) if(!v[e[i].e]) fill_forward(
     e[i].e);
  stk[++stk[0]]=x;
}
void fill_backward(int x)
{
  int i;
  v[x]=false;
  group_num[x]=group_cnt;
  for(i=spr[x];i;i=er[i].nxt) if(v[er[i].e])
     fill_backward(er[i].e);
}
void add_edge(int v1, int v2) //add edge v1->v2
{
  e [++E].e=v2; e [E].nxt=sp [v1]; sp [v1]=E;
  er[  E].e=v1; er[E].nxt=spr[v2]; spr[v2]=E;
}
void SCC()
{
  int i;
  stk[0]=0;
  memset(v, false, sizeof(v));
  for(i=1;i<=V;i++) if(!v[i]) fill_forward(i);
  group_cnt=0;
  for(i=stk[0];i>=1;i--) if(v[stk[i]]){group_cnt++;
     fill_backward(stk[i]);}
}
```

4.3 Eulerian path

```
struct Edge;
typedef list<Edge>::iterator iter;

struct Edge
{
        int next_vertex;
        iter reverse_edge;

        Edge(int next_vertex)
                :next_vertex(next_vertex)
                { }
};

const int max_vertices = ;
int num_vertices;
list<Edge> adj[max_vertices];           // adjacency
   list

vector<int> path;

void find_path(int v)
{
        while(adj[v].size() > 0)
        {
                int vn = adj[v].front().next_vertex;
                adj[vn].erase(adj[v].front().
                   reverse_edge);
                adj[v].pop_front();
                find_path(vn);
        }
        path.push_back(v);
}

void add_edge(int a, int b)
{
        adj[a].push_front(Edge(b));
```

```
        iter ita = adj[a].begin();
        adj[b].push_front(Edge(a));
        iter itb = adj[b].begin();
        ita->reverse_edge = itb;
        itb->reverse_edge = ita;
}
```

4.4 Minimum spanning trees

```
// This function runs Prim's algorithm for constructing minimum
// weight spanning trees.
//
// Running time: O(|V|^2)
//
//   INPUT:   w[i][j] = cost of edge from i to j
//
//            NOTE: Make sure that w[i][j] is nonnegative and
//            symmetric.  Missing edges should be given -1
//            weight.
//
//   OUTPUT:  edges = list of pair<int,int> in minimum spanning tree
//            return total weight of tree

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

using namespace std;

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

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

T Prim (const VVT &w, VPII &edges){
  int n = w.size();
  VI found (n);
  VI prev (n, -1);
  VT dist (n, 1000000000);
  int here = 0;
  dist[here] = 0;

  while (here != -1){
    found[here] = true;
    int best = -1;
    for (int k = 0; k < n; k++) if (!found[k]){
      if (w[here][k] != -1 && dist[k] > w[here][k]){
        dist[k] = w[here][k];
        prev[k] = here;
      }
      if (best == -1 || dist[k] < dist[best]) best = k;
    }
    here = best;
  }

  T tot_weight = 0;
  for (int i = 0; i < n; i++) if (prev[i] != -1){
    edges.push_back (make_pair (prev[i], i));
    tot_weight += w[prev[i]][i];
  }
  return tot_weight;
}

int main(){
  int ww[5][5] = {
    {0, 400, 400, 300, 600},
    {400, 0, 3, -1, 7},
    {400, 3, 0, 2, 0},
    {300, -1, 2, 0, 5},
    {600, 7, 0, 5, 0}
  };
  VVT w(5, VT(5));
  for (int i = 0; i < 5; i++)
    for (int j = 0; j < 5; j++)
      w[i][j] = ww[i][j];

  // expected: 305
  //           2 1
  //           3 2
  //           0 3
  //           2 4

  VPII edges;
  cout << Prim (w, edges) << endl;
  for (int i = 0; i < edges.size(); i++)
    cout << edges[i].first << " " << edges[i].second << endl;
}
```

5 Data structures

5.1 Suffix array

```
// Suffix array construction in O(L log^2 L) time.  Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:   string s
//
// OUTPUT:  array suffix[] such that suffix[i] = index (from 0 to L-1)
//          of substring s[i...L-1] in the list of sorted suffixes.
//          That is, if we take the inverse of the permutation suffix[],
//          we get the actual suffix array.

#include <vector>
#include <iostream>
#include <string>

using namespace std;
```

```
struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P;
  vector<pair<pair<int,int>,int> > M;

  SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M(L) {
    for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
    for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
      P.push_back(vector<int>(L, 0));
      for (int i = 0; i < L; i++)
        M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ? P[level-1][i + skip] : -1000), i);
      sort(M.begin(), M.end());
      for (int i = 0; i < L; i++)
        P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[level][M[i-1].second] : i;
    }
  }

  vector<int> GetSuffixArray() { return P.back(); }

  // returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]
  int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
      if (P[k][i] == P[k][j]) {
        i += 1 << k;
        j += 1 << k;
        len += 1 << k;
      }
    }
    return len;
  }
};

// BEGIN CUT
// The following code solves UVA problem 11512: GATTACA.
#define TESTING
#ifdef TESTING
int main() {
  int T;
  cin >> T;
  for (int caseno = 0; caseno < T; caseno++) {
    string s;
    cin >> s;
    SuffixArray array(s);
    vector<int> v = array.GetSuffixArray();
    int bestlen = -1, bestpos = -1, bestcount = 0;
    for (int i = 0; i < s.length(); i++) {
      int len = 0, count = 0;
      for (int j = i+1; j < s.length(); j++) {
        int l = array.LongestCommonPrefix(i, j);
        if (l >= len) {
          if (l > len) count = 2; else count++;
          len = l;
        }
      }
      if (len > bestlen || len == bestlen && s.substr(bestpos, bestlen) > s.substr(i, len)) {
        bestlen = len;
        bestcount = count;
        bestpos = i;
      }
    }
    if (bestlen == 0) {
      cout << "No repetitions found!" << endl;
    } else {
      cout << s.substr(bestpos, bestlen) << " " << bestcount << endl;
    }
  }
}

#else
// END CUT
int main() {

  // bobocel is the 0'th suffix
  //  obocel is the 5'th suffix
  //   bocel is the 1'st suffix
  //    ocel is the 6'th suffix
  //     cel is the 2'nd suffix
  //      el is the 3'rd suffix
  //       l is the 4'th suffix
  SuffixArray suffix("bobocel");
  vector<int> v = suffix.GetSuffixArray();

  // Expected output: 0 5 1 6 2 3 4
  //                  2
  for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << endl;
  cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
// BEGIN CUT
#endif
// END CUT
```

5.2 KD-tree

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

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

```
#include <limits>
#include <cstdlib>
using namespace std;

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

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

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

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

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

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

// bounding box for a set of points
struct bbox{
    ntype x0, x1, y0, y1;

    bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-
       sentry) {}

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

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

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

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

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

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

    // recursively builds a kd-tree from a given cloud
       of points
    void construct(vector<point> &vp){
        // compute bounding box for points at this node
        bound.compute(vp);

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

            // divide by taking half the array for each
               child
            // (not best performance if many duplicates
               in the middle)
            int half = vp.size()/2;
            vector<point> vl(vp.begin(), vp.begin()+half
```

```
                );
            vector<point> vr(vp.begin()+half, vp.end());
            first = new kdnode();   first->construct(vl)
                ;
            second = new kdnode();  second->construct(vr
                );
        }
    }
};

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

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

    // recursive search method returns squared distance
        to nearest point
    ntype search(kdnode *node, const point &p)
    {
        if (node->leaf) {
            // commented special case tells a point not
                to find itself
//              if (p == node->pt) return sentry;
//              else
                return pdist2(p, node->pt);
        }

        ntype bfirst = node->first->intersect(p);
        ntype bsecond = node->second->intersect(p);

        // choose the side with the closest bounding box
            to search first
        // (note that the other side is also searched if
            needed)
        if (bfirst < bsecond) {
            ntype best = search(node->first, p);
            if (bsecond < best)
                best = min(best, search(node->second, p)
                    );
            return best;
        }
        else {
            ntype best = search(node->second, p);
            if (bfirst < best)
                best = min(best, search(node->first, p))
                    ;
            return best;
        }
    }

    // squared distance to the nearest
    ntype nearest(const point &p) {
        return search(root, p);
    }
};

// some basic test code here

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

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

5.3 Splay tree

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

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

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

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

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

inline void pushDown(Node *x)
{
  if(x->isTurned)
  {
    makeTurned(x->ch[0]);
```

```
    makeTurned(x->ch[1]);
    x->isTurned ^= 1;
  }
}
inline void rotate(Node *x, int c)
{
  Node *y = x->pre;
  x->pre = y->pre;
  if(y->pre != null)
    y->pre->ch[y == y->pre->ch[1]] = x;
  y->ch[!c] = x->ch[c];
  if(x->ch[c] != null)
    x->ch[c]->pre = y;
  x->ch[c] = y, y->pre = x;
  update(y);
  if(y == root)
    root = x;
}
void splay(Node *x, Node *p)
{
  while(x->pre != p)
  {
    if(x->pre->pre == p)
      rotate(x, x == x->pre->ch[0]);
    else
    {
      Node *y = x->pre, *z = y->pre;
      if(y == z->ch[0])
      {
        if(x == y->ch[0])
          rotate(y, 1), rotate(x, 1);
        else
          rotate(x, 0), rotate(x, 1);
      }
      else
      {
        if(x == y->ch[1])
          rotate(y, 0), rotate(x, 0);
        else
          rotate(x, 1), rotate(x, 0);
      }
    }
  }
  update(x);
}
void select(int k, Node *fa)
{
  Node *now = root;
  while(1)
  {
    pushDown(now);
    int tmp = now->ch[0]->size + 1;
    if(tmp == k)
      break;
    else if(tmp < k)
      now = now->ch[1], k -= tmp;
    else
      now = now->ch[0];
  }
  splay(now, fa);
}
Node *makeTree(Node *p, int l, int r)
{
  if(l > r)
    return null;
  int mid = (l + r) / 2;
  Node *x = allocNode(mid);
  x->pre = p;
  x->ch[0] = makeTree(x, l, mid - 1);
  x->ch[1] = makeTree(x, mid + 1, r);
  update(x);
  return x;
}
int main()
{
  int n, m;
  null = allocNode(0);
  null->size = 0;
  root = allocNode(0);
  root->ch[1] = allocNode(oo);
  root->ch[1]->pre = root;
  update(root);

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

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

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

5.4 Lazy segment tree

```
template <typename T>
class SegmentTree{
        vector<T> segtree,lazy;
        public:
        SegmentTree(int size){
                segtree.resize(4*size,0);
                lazy.resize(4*size,0);
        }
        void update(int u, int a, int b, int i, int j, T
            x){
                if(lazy[u]){
                        segtree[u]+=(b-a+1)*lazy[u];
                        if(a!=b)
                                lazy[u*2]+=lazy[u],lazy
```

```
                        [2*u+1]+=lazy[u];
                lazy[u]=0;
            }
            if(j<a || i>b || a>b) return;
            if(j>=b && i<=a){
                segtree[u]+=(b-a+1)*x;
                if(a!=b) lazy[u*2]+=x,lazy[2*u
                    +1]+=x;
                return;
            }
            update(u*2,a,(a+b)/2,i,j,x); update(u
                *2+1,(a+b)/2+1,b,i,j,x);
            segtree[u]=segtree[u*2]+segtree[u*2+1];
        }
        void update(int i, T x){
            update(1, 0, segtree.size()/4-1, i, i, x
                );
        }
        void update(int i, int j, T x){
            update(1, 0, segtree.size()/4-1, i, j, x
                );
        }
        T query(int u, int a, int b, int i, int j){
            if(j<a || i>b || a>b) return 0;
            if(lazy[u]){
                segtree[u]+=(b-a+1)*lazy[u];
                if(a!=b)
                    lazy[u*2]+=lazy[u],lazy
                        [2*u+1]+=lazy[u];
                lazy[u]=0;
            }
            if(j>=b && i<=a) return segtree[u];
            return query(u*2,a,(a+b)/2,i,j)+query(u
                *2+1,1+(a+b)/2,b,i,j);
        }
        T query(int i, int j){
            return query(1,0,segtree.size()/4-1,i,j)
                ;
        }
};
```

5.5 Centroid decomposition

```
set<int> v[100005];
map<int,int> mp[100005];
int n,up[100005][17],lvl[100005],par[100005],CNT,siz
    [100005],tin[100005],tout[100005];
void dfspre(int u, int dad=1, int depth = 0){
    static int clk = 0;
    tin[u]=clk++;
    up[u][0] = dad;
    lvl[u] = depth;
    for(int i=1;i<17;++i)
        up[u][i] = up[up[u][i-1]][i-1];
    for(int i:v[u]) if(i!=dad)
        dfspre(i,u,depth+1);
    tout[u]=clk++;
}
int dfs(int u, int dad){
    siz[u] = 1;
    for(int i:v[u]) if(i!=dad)
        siz[u] += dfs(i,u);
    return siz[u];
}
int centroid(int u, int dad){
    for(int i:v[u]) if(i!=dad && siz[i]>CNT)
        return centroid(i,u);
    return u;
}
void decompose(int u, int dad){
    CNT = dfs(u,dad)/2;
    int centre = centroid(u,dad);
    par[centre] = dad;
    for(int i:v[centre]) if(i!=dad){
        v[i].erase(centre);
        decompose(i,centre);
    }
    v[centre].clear();
}
int lca(int u, int v){
    if(lvl[u]>lvl[v]) swap(u,v);
    if(tin[u]<=tin[v] && tout[v]<=tout[u]) return u;
    for(int i=17;i--;)
        if(!(tin[up[u][i]]<=tin[v] && tout[v]<=tout[up[u
            ][i]]))
            u = up[u][i];
    return up[u][0];
}
void update(int u){
    for(int node = u;u;u = par[u])
        ++mp[u][lvl[node]+lvl[u] - 2*lvl[lca(u,node)]];
}
int get(int u){
    int ans = INT_MAX;
    for(int node = u; u; u = par[u])
        ans = min(ans,lvl[u]+lvl[node]-2*lvl[lca(u,node)
            ]+(*mp[u].begin()).first);
    return ans;
}
```

5.6 Heavy-Light decomposition

```
#include <bits/stdc++.h>
using namespace std;
int a[100005],sz[100005],lvl[100005];
int seg_id[100005],pos_id[100005],parent[100005];
int CNT;
int e_to_v[100005];
vector<pair<int,int>> edges(100005);
vector<pair<int,int>> v[100005];
vector<int> chain[100005];
template <typename T>
class SegmentTree{
  vector<T> segtree,lazy;
  public:
  SegmentTree(int size){
    segtree.resize(4*size,0);
    lazy.resize(4*size,0);
  }
  void update(int u, int a, int b, int i, int j, T x){
```

```cpp
    if(lazy[u]){
      segtree[u]+=(b-a+1)*lazy[u];
      if(a!=b)
        lazy[u*2]+=lazy[u],lazy[2*u+1]+=lazy[u];
      lazy[u]=0;
    }
    if(j<a || i>b || a>b) return;
    if(j>=b && i<=a){
      segtree[u]=x;
      if(a!=b) lazy[u*2]+=x,lazy[2*u+1]+=x;
      return;
    }
    update(u*2,a,(a+b)/2,i,j,x); update(u*2+1,(a+b)/2+1,
       b,i,j,x);
    segtree[u]=max(segtree[u*2],segtree[u*2+1]);
  }
  void update(int i, T x){
    update(1, 0, segtree.size()/4-1, i, i, x);
  }
  void update(int i, int j, T x){
    update(1, 0, segtree.size()/4-1, i, j, x);
  }
  T query(int u, int a, int b, int i, int j){
    if(j<a || i>b || a>b) return 0;
    if(lazy[u]){
      segtree[u]+=(b-a+1)*lazy[u];
      if(a!=b)
        lazy[u*2]+=lazy[u],lazy[2*u+1]+=lazy[u];
      lazy[u]=0;
    }
    if(j>=b && i<=a) return segtree[u];
    return max(query(u*2,a,(a+b)/2,i,j),query(u*2+1,1+(a
       +b)/2,b,i,j));
  }
  T query(int i, int j){
    return query(1,0,segtree.size()/4-1,i,j);
  }
};
int dfs(int u, int dad=1, int depth=1){
  lvl[u] = depth;
  sz[u] = 1;
  for(auto i:v[u]) if(i.first!=dad)
    sz[u] += dfs(i.first,u,depth+1);
  return sz[u];
}
void hld(int u, int dad = 1, int chain_no = 0, int
   chain_parent = 0){
  seg_id[u] = chain_no;
  pos_id[u] = chain[chain_no].size();
  parent[u] = chain_parent;
  chain[chain_no].push_back(u);
  int max_sz = 0, heavy_child = -1;
  for(auto i:v[u]) if(i.first!=dad){
    a[i.first] = i.second;
    if(max_sz < sz[i.first])
      max_sz = sz[i.first], heavy_child = i.first;
  }
  if(heavy_child!=-1)
    hld(heavy_child, u, chain_no, chain_parent);
  for(auto i:v[u]) if(i.first!=dad && i.first!=
     heavy_child)
    hld(i.first,u,++CNT, u);
}
int main(){
  ios_base::sync_with_stdio(0);
  cin.tie(0); cout.tie(0);
  int T, n, x, y;
  for(cin>>T;T-- && cin>>n;){
    CNT = 0;
    for(int i=0;i<100005;chain[i].clear(), v[i].clear(),
       i++);

    for(int i=1,c;i<n;++i){
      cin>>x>>y>>c;
      edges[i] = {x,y};
      v[x].push_back({y,c});
      v[y].push_back({x,c});
    }

    dfs(1);

    for(int i=1;i<n;++i){
      int u = edges[i].first, v = edges[i].second;
      e_to_v[i] = (lvl[u]<lvl[v]?v:u);
    }

    hld(1);

    vector<SegmentTree<int>> ST;
    for(int i=0;i<=CNT;++i){
      ST.push_back(SegmentTree<int>(chain[i].size()));
      for(auto u:chain[i])
        ST[i].update(pos_id[u],a[u]);
    }

    for(string type;cin>>type && type!="DONE";){
      cin>>x>>y;
      if(type=="QUERY"){
        int res = 0;
        while(x!=y){
          if(seg_id[x]>seg_id[y]) swap(x,y);
          else if(seg_id[x] == seg_id[y]){
            if(pos_id[x]>pos_id[y]) swap(x,y);
            res = max(res,ST[seg_id[y]].query(pos_id[x
               ]+1,pos_id[y]));
            break;
          }
          res = max(res, ST[seg_id[y]].query(0,pos_id[y
             ]));
          y = parent[y];
        }
        cout<<res<<'\n';
      }
      else if(type == "CHANGE"){
        int u = e_to_v[x];
        ST[seg_id[u]].update(pos_id[u],y);
      }
    }
  }
}
```

6 Miscellaneous

6.1 Dynamic Programming(DnC)

```
long long dp[21][100005];
void cost(int x, int y);
void computeDP(int idx,int jleft,int jright,int kleft,
    int kright){
        if(jleft>jright) return;
        int jmid=(jleft+jright)/2;
        int bestk=jmid;
        for(int k=kleft;k<=min(kright,jmid);++k){
                cost(k,jmid);
                if(dp[idx-1][k-1]+tot<dp[idx][jmid])
                        dp[idx][jmid]=dp[idx-1][k-1]+tot
                            ,bestk=k;
        }
        computeDP(idx,jleft,jmid-1,kleft,bestk);
        computeDP(idx,jmid+1,jright,bestk,kright);
}
int main(){
        for(int i=0;i<=k;++i)
                for(int j=0;j<=n;dp[i][j++]=1e17);
        dp[0][0]=0;
        for(int i=1;i<=k;++i)
                computeDP(i,1,n,1,n);
        cout<<dp[k][n];
}
```

6.2 Longest increasing subsequence

```
// Given a list of numbers of length n, this routine
    extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
//   INPUT: a vector of integers
//   OUTPUT: a vector containing the longest increasing
    subsequence

typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

#define STRICTLY_INCREASNG

VI LongestIncreasingSubsequence(VI v) {
  VPII best;
  VI dad(v.size(), -1);

  for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
    PII item = make_pair(v[i], 0);
    VPII::iterator it = lower_bound(best.begin(), best.
        end(), item);
    item.second = i;
#else
    PII item = make_pair(v[i], i);
    VPII::iterator it = upper_bound(best.begin(), best.
        end(), item);
#endif
    if (it == best.end()) {
      dad[i] = (best.size() == 0 ? -1 : best.back().
          second);
      best.push_back(item);
    } else {
      dad[i] = it == best.begin() ? -1 : prev(it)->
          second;
      *it = item;
    }
  }

  VI ret;
  for (int i = best.back().second; i >= 0; i = dad[i])
    ret.push_back(v[i]);
  reverse(ret.begin(), ret.end());
  return ret;
}
```

6.3 Knuth-Morris-Pratt

```
typedef vector<int> VI;

void buildPi(string& p, VI& pi){
  pi = VI(p.length());
  int k = -2;
  for(int i = 0; i < p.length(); i++) {
    while(k >= -1 && p[k+1] != p[i])
      k = (k == -1) ? -2 : pi[k];
    pi[i] = ++k;
  }
}

int KMP(string& t, string& p){
  VI pi;
  buildPi(p, pi);
  int k = -1;
  for(int i = 0; i < t.length(); i++) {
    while(k >= -1 && p[k+1] != t[i])
      k = (k == -1) ? -2 : pi[k];
    k++;
    if(k == p.length() - 1) {
      // p matches t[i-m+1, ..., i]
      cout << "matched at index " << i-k << ": ";
      cout << t.substr(i-k, p.length()) << endl;
      k = (k == -1) ? -2 : pi[k];
    }
  }
  return 0;
}

int main(){
  string a = "AABAACAADAABAABA", b = "AABA";
  KMP(a, b); // expected matches at: 0, 9, 12
}
```