UW-Madison ACM Reference 2014

Graph

Geometry and Num

String manipulation

Maximum matching (C++)

```
// This code performs maximum bipartite matching.
// It has a heuristic that will give excellent performance on complete graphs
// where rows <= columns.
//
//   INPUT: w[i][j] = cost from row node i and column node j or NO_EDGE
//   OUTPUT: mr[i] = assignment for row node i or -1 if unassigned
//           mc[j] = assignment for column node j or -1 if unassigned
//
//   BipartiteMatching returns the number of matches made.
//
// Contributed by Andy Lutomirski.

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

const int NO_EDGE = -(1<<30);  // Or any other value.

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

```
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] != NO_EDGE && mr[i] != j) {
      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);
  VI seen(w.size());

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

Maximum flow (C++)

```
/****************
 * Maximum flow * (Dinic's on an adjacency list + matrix)
 ****************
 * Takes a weighted directed graph of edge capacities as an adjacency
 * matrix 'cap' and returns the maximum flow from s to t.
 *
 * PARAMETERS:
 *      - cap (global): adjacency matrix where cap[u][v] is the capacity
 *          of the edge u->v. cap[u][v] is 0 for non-existent edges.
 *      - n: the number of vertices ([0, n-1] are considered as vertices).
 *      - s: source vertex.
 *      - t: sink.
 * RETURNS:
 *      - the flow
 *      - prev contains the minimum cut. If prev[v] == -1, then v is not
 *          reachable from s; otherwise, it is reachable.
 * RUNNING TIME:
 *      - O(n^3)
 **/

// the maximum number of vertices
#define NN 1024
const int INF = 2000000000;

// adjacency matrix (fill this up)
// If you fill adj[][] yourself, make sure to include both u->v and v->u.
int cap[NN][NN], deg[NN], adj[NN][NN];

// BFS stuff
int q[NN], prev[NN];
```

```
int dinic( int n, int s, int t ) {
    int flow = 0;
    while( true ) {
        memset( prev, -1, sizeof( prev ) );
        int qf = 0, qb = 0;
        prev[q[qb++] = s] = -2;
        while ( qb > qf && prev[t] == -1 )
            for ( int u = q[qf++], i = 0, v; i < deg[u]; i++ )
                if( prev[v = adj[u][i]] == -1 && cap[u][v] )
                    prev[q[qb++] = v] = u;
        if ( prev[t] == -1 ) break;
        for ( int z = 0; z < n; z++ ) if( cap[z][t] && prev[z] != -1 ) {
            int bot = cap[z][t];
            for ( int v = z, u = prev[v]; u >= 0; v = u, u = prev[v] )
                bot = min(bot, cap[u][v]);
            if ( !bot ) continue;
            cap[z][t] -= bot;
            cap[t][z] += bot;
            for ( int v = z, u = prev[v]; u >= 0; v = u, u = prev[v] ) {
                cap[u][v] -= bot;
                cap[v][u] += bot;
            }
            flow += bot;
        }
    }

    return flow;
}

//----------------- EXAMPLE USAGE -----------------
int main() {
    // read a graph into cap[][]
    memset( cap, 0, sizeof( cap ) );
    int n, s, t, m;
    scanf( " %d %d %d %d", &n, &s, &t, &m );
    while( m-- ) {
        int u, v, c; scanf( " %d %d %d", &u, &v, &c );
        cap[u][v] = c;
    }

    // init the adjacency list adj[][] from cap[][]
    memset( deg, 0, sizeof( deg ) );
    for( int u = 0; u < n; u++ )
        for( int v = 0; v < n; v++ ) if( cap[u][v] || cap[v][u] )
            adj[u][deg[u]++] = v;

    printf( "%d\n", dinic( n, s, t ) );
    return 0;
}
```

Strongly connected components (C++)

```
#define mp make_pair
#define pb push_back
#define MAXV 100
#define MAXE 100
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]);
    }
}
int main() {
  int t;
  scanf("%d", &t);
  while (t--) {
    memset(stk, -1, sizeof(stk));
    memset(sp, -1, sizeof(sp));
    memset(spr, -1, sizeof(spr));
```

```
    int m;
    E = 0;
    scanf("%d%d", &V, &m);
    for (int i = 0; i < m; i++) {
      int x, y;
      scanf("%d%d", &x, &y);
      add_edge(x, y);
    }
    SCC();
    printf("%d\n", group_cnt);
    for (int i = 1; i <= V; i++)
      printf("%d stays in group %d\n", i, group_num[i]);
    printf("\n");
  }
}
```

Minimum spanning tree (C++)

```
#define mp make_pair
#define pb push_back
#define Max 1000100
using namespace std;
vector< pair<int, pair<int, int> > >v, mst;
int parent[Max], rank[Max];

void makeSet(int n) {
        for (int i = 0; i < n; i++)
                parent[i] = i,
                rank[i] = 0;
}

int find(int x) {
        if (x != parent[x])
                parent[x] = find(parent[x]);
        return parent[x];
}

void merge(int x, int y) {
        int Px = find(x);
        int Py = find(y);

        if (rank[Px] > rank[Py])
                parent[Py] = Px;
        else
                parent[Px] = Py;

        if (rank[Px] == rank[Py])
                ++rank[Py];

}

int main(int argc, char *argv[]) {
    //freopen("in.txt", "r", stdin);
    //freopen("out.txt", "w", stdout);

    int n, m;
    scanf("%d%d", &n, &m);
    makeSet(n);
```

```
    for (int i = 0; i < m; i++) {
        int x, y, w;
        scanf("%d%d%d", &x, &y, &w);
        x--, --y;
        v.pb(mp(w, mp(x, y)));
    }
    sort(v.begin(), v.end());

    long long ret = 0ll;

    for (int i = 0, k = 0; k < n-1; i++) {
        int x = find(v[i].second.first);
        int y = find(v[i].second.second);
        if (x != y) {
                mst.pb(v[i]);
                ret += 1ll*v[i].first;
                merge(x, y);
                k++;
        }
    }

    printf("%lld\n", ret);

    return 0;
}
```

Suurballe (C++)

```
// Suurballe's algorithm.
// finds smallest possible sum of weight of two disjoint paths from s to t.
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <climits>
#include <cctype>

#include <vector>
#include <map>
#include <queue>
#include <algorithm>

using namespace std;

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

//XXX XXX define me
//#define PRINT_PATH
//XXX XXX define me

#define MAX_E 99999 // should be 2*actual maximum
#define MAX_V 9999

int e_from[MAX_E], e_to[MAX_E], e_cap[MAX_E], e_flow[MAX_E], e_cost[MAX_E], e_dual[MAX_E];
VVI edges;
int s, t;
```

```
int back_e[MAX_V];
int cost[MAX_V];

int augment() {
    memset(cost, 0x10, sizeof(cost));
    priority_queue<PII> q;
    cost[s] = 0;
    q.push(make_pair(0,s));
    while( q.size() > 0 ) {
        PII cur_p = q.top(); q.pop();
        int cur = cur_p.second;
        int curcost = -cur_p.first;
        if( cost[cur] < curcost ) { continue; }
        if( cur == t ) {
            // augment path
            int v = t;
            #ifdef PRINT_PATH
            VI path; path.push_back(v/2);
            #endif
            while( v != s ) {
                int e = back_e[v];
                int d = e_dual[e];
                e_flow[e] += 1;
                e_flow[d] -= 1;
                v = e_from[e];
                #ifdef PRINT_PATH
                if( v/2 != *path.rbegin() ) { path.push_back(e_cost[e]); path.push_back(v/2);
                #endif
            }
            #ifdef PRINT_PATH
            reverse(path.begin(), path.end()); printf("%d", path[0]);
            for( int i = 1; i < path.size(); i += 2 ) {
              printf(" --[%d]--> %d", path[i], path[i+1]);
            } puts("");
            #endif
            return curcost;
        }
        for( int i = 0; i < edges[cur].size(); ++i ) {
            int e = edges[cur][i];
            if( e_flow[e] >= e_cap[e] ) { continue; }
            int next = e_to[e];
            int nextcost = curcost + e_cost[e];
            if( cost[next] > nextcost ) {
                back_e[next] = e;
                cost[next] = nextcost;
                q.push(make_pair(-nextcost, next));
            }
        }
    }
    return -1;
}

void solve() {
    int v, e; if( scanf("%d%d", &v, &e) != 2 ) exit(0);
    edges = VVI(2*v,VI());
    for( int i = 0; i < v; ++i ) {
        int idx = 2*i;
        e_from[idx] = 2*i;
```

```
        e_to[idx] = 2*i+1;
        e_cost[idx] = e_cost[idx+1] = 0;
        e_from[idx+1] = e_to[idx];
        e_to[idx+1] = e_from[idx];
        e_cap[idx] = e_cap[idx+1] = 1;
        e_flow[idx] = 0; e_flow[idx+1] = 1;
        e_dual[idx] = idx+1; e_dual[idx+1] = idx;
        edges[e_from[idx]].push_back(idx);
        edges[e_from[idx+1]].push_back(idx+1);
    }
    for( int i = 0; i < e; ++i ) {
        int idx = 2*i+2*v;
        scanf("%d%d%d", &e_from[idx], &e_to[idx], &e_cost[idx]);
        e_from[idx] -= 1; e_to[idx] -= 1;
        e_from[idx] *= 2; e_to[idx] *= 2;
        e_from[idx] += 1;
        e_from[idx+1] = e_to[idx];
        e_to[idx+1] = e_from[idx];
        e_cost[idx+1] = -e_cost[idx];
        e_cap[idx] = e_cap[idx+1] = 1;
        e_flow[idx] = 0; e_flow[idx+1] = 1;
        e_dual[idx] = idx+1; e_dual[idx+1] = idx;
        edges[e_from[idx]].push_back(idx);
        edges[e_from[idx+1]].push_back(idx+1);
    }
    s = 1; t = 2*(v-1);
    int total = 0;
    for( int i = 0; i < 2; ++i ) {
        total += augment();
    }
    printf("%d\n" , total);
}
int main() {
    while( true ) solve();
}
```

Dijkstra (C++)

```
#define mp make_pair
#define pb push_back
#define Pair pair<int, int> // V W
#define xx first
#define yy second
#define Max 100010
struct cmp {
        bool operator() (const Pair &a, const Pair &b) const {
                return a.yy > b.yy;
        }
};
const long long INF = 12345678987654321LL;
priority_queue< Pair, vector< Pair >, cmp> q;
vector< Pair >v[Max];
int n, m;
bool used[Max];
int uu[Max];
long long d[Max];
void dfs(int i) {
        if (i == 0) {
```

```
                printf("1 ");
                return;
        }
        dfs(uu[i]);
        printf("%d ", i+1);
}
int main(int argc, char *argv[]) {
    scanf("%d%d", &n, &m);
    while (m--) {
        int a, b, w;
        scanf("%d%d%d", &a, &b, &w);
        a--, --b;
        v[a].pb(mp(b, w));
        v[b].pb(mp(a, w));
    }
    uu[0] = 0; //from vertex 0.
    for (int i = 0; i < n; i++)
        d[i] = INF;
    d[0] = 0ll;
    q.push(mp(0, 0));
    while (q.size()) {
        int u = q.top().xx;
        q.pop();
        int sz = v[u].size();
        for (int i = 0; i < sz; i++) {
                int vv = v[u][i].xx;
                int ww = v[u][i].yy;

                if(!used[vv] && d[u]+ww < d[vv]) {
                        d[vv] = d[u]+ww;
                        q.push(mp(vv, d[vv]));
                        uu[vv] = u;
                }

        }
        used[u] = 1;
    }
    if (d[n-1] == INF)
        printf("-1\n");
    else
        dfs(n-1);
    return 0;
}
```

Bellman ford (C++)

```
// 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
```

```
typedef int TYPE;
typedef vector<TYPE> 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;
}
```

Articulation point (C++)

```
#define N 400001
#define pb push_back
int num[N] = {0}, low[N] = {0};
int visit[N] = {0};
int parentt[N] = {0};
int V, E;
vector<int> ad[N];
int art[N] = {0};
int counter;
int root, child = 0;

void findartd(int ver) {
    visit[ver] = 1;
          low[ver] = num[ver] = counter++;
    vector<int>::iterator it;
    for (it = ad[ver].begin(); it < ad[ver].end(); it++)
         if (visit[*it] == 0) {
             if (root == ver)
                 child++;
             parentt[*it] = ver;
             int tm = *it;
             findartd(tm);
                   if (low[*it] >= num[ver]) {

                 if (art[ver] == 0 && root != ver)
                     art[ver] = 1;

             }
             low[ver]=min(low[ver],low[*it]);
```

```
            } else
                if (parentt[ver] != *it)
                    low[ver] = min(low[ver], num[*it]);
}
int main() {
        while (true) {
                scanf("%d %d", &V, &E);
                if(V == 0 && E == 0) break;
    for (int i = 0; i < E; i++) {
        int s, t;
        scanf("%d %d", &s, &t);
        ad[s].pb(t);
        ad[t].pb(s);
    }
    for (int i = 0; i < V; i++)
        if (!visit[i]) {
            counter = 1;
            child = 0;
            root = i;
            findartd(root);
            if (child > 1) art[root] = 1;
        }
    for (int i = 0; i < V; i++)
         if(art[i] == 1)
            printf("%d\n", i);
    for (int i = 0; i < V; i++) ad[i].clear();
    memset(num,0,sizeof(num));
          memset(visit,0,sizeof(visit));
          memset(parentt,0,sizeof(parentt));
    memset(art,0,sizeof(art));
          memset(low,0,sizeof(low));

  }//while(true)..

return 0; }
```

Floyd (C++)

```
typedef double TYPE;
typedef vector<TYPE> VT;
typedef vector<VT> VVT;
typedef vector<int> VI;
typedef vector<VI> VVI;
bool FloydWarshall (VVT &w, VVI &prev){
  int n = w.size();
  prev = VVI (n, VI(n, -1));

  for (int k = 0; k < n; k++){
    for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
        if (w[i][j] > w[i][k] + w[k][j]){
          w[i][j] = w[i][k] + w[k][j];
          prev[i][j] = k;
        }
      }
    }
  }
  // check for negative weight cycles
```

```
  for (int i = 0; i < n; i++)
    if (w[i][i] < 0) return false;
  return true;
}
```

Euclid (C++)

```
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) {
  int tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}
// computes lcm(a,b)
int lcm(int a, int b) {
  return a/gcd(a,b)*b;
}
// returns d = 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 solutions;
  int d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod (x*(b/d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back(mod(x + i*(n/d), n));
  }
  return solutions;
}
// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
  int x, y;
  int d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}
// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b.  Here, z is unique modulo M = lcm(x,y).
// Return (z,M).  On failure, M = -1.
PII chinese_remainder_theorem(int x, int a, int y, int b) {
```

```
  int s, t;
  int d = extended_euclid(x, y, s, t);
  if (a%d != b%d) return make_pair(0, -1);
  return make_pair(mod(s*b*x+t*a*y,x*y)/d, x*y/d);
}
// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i.  Note that the solution is
// unique modulo M = lcm_i (x[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 &x, const VI &a) {
  PII ret = make_pair(a[0], x[0]);
  for (int i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.second, ret.first, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}
// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine(int a, int b, int c, int &x, int &y) {
  int d = gcd(a,b);
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}
```

Geometry (C++)

```
double INF = 1e100;
double 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;
}

// Creates a d offseted polygon by given polygon points.
double offsetPolygonArea(const vector<PT> &p, double D) {
  vector<PT> q;
  for (int i = 0; i < p.size(); i++) {
      double a1 = p[(i+1)%p.size()].y - p[i].y;
```

```
        double b1 = p[i].x - p[(i+1)%p.size()].x;
        double t = sqrt(a1 * a1 + b1 * b1);
        a1 /= t;
        b1 /= t;
        double c1 = a1 * p[i].x + b1 * p[i].y + D;

        double a2 = p[(i+2)%p.size()].y - p[(i+1)%p.size()].y;
        double b2 = p[(i+1)%p.size()].x - p[(i+2)%p.size()].x;
        t = sqrt(a2 * a2 + b2 * b2);
        a2 /= t;
        b2 /= t;
        double c2 = a2 * p[(i+1)%p.size()].x + b2 * p[(i+1)%p.size()].y + D;

        double det = a1 * b2 - a2 * b1;
        double xx = (c1 * b2 - c2 * b1) / det;
        double yy = (a1 * c2 - a2 * c1) / det;
        q.pb(PT(xx, yy));
    }
    //printf("equals? %d\n", q.size() == p.size());
    return ComputeArea(q);
}

int main() {
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    // expected: (-5,2)
    cerr << RotateCCW90(PT(2,5)) << endl;

    // expected: (5,-2)
    cerr << RotateCW90(PT(2,5)) << endl;

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

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

    // expected: (5,2) (7.5,3) (2.5,1)
    cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
         << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
         << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

    // expected: 6.78903
    cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

    // expected: 1 0 1
    cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
         << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
         << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

    // expected: 0 0 1
    cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
         << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
         << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

    // expected: 1 1 1 0
    cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
         << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
```

```
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

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

    // expected: (1,1)
    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));

    // expected: 1 1 1 0 0
    cerr << PointInPolygon(v, PT(2,2)) << " "
        << PointInPolygon(v, PT(2,0)) << " "
        << PointInPolygon(v, PT(0,2)) << " "
        << PointInPolygon(v, PT(5,2)) << " "
        << PointInPolygon(v, PT(2,5)) << endl;

    // expected: 0 1 1 1 1
    cerr << PointOnPolygon(v, PT(2,2)) << " "
        << PointOnPolygon(v, PT(2,0)) << " "
        << PointOnPolygon(v, PT(0,2)) << " "
        << PointOnPolygon(v, PT(5,2)) << " "
        << PointOnPolygon(v, PT(2,5)) << endl;

    // expected: (1,6)
    //           (5,4) (4,5)
    //           blank line
    //           (4,5) (5,4)
    //           blank line
    //           (4,5) (5,4)
    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 = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
    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;

    // area should be 5.0
    // centroid should be (1.1666666, 1.166666)
    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;

    return 0;
}
```

Convex hull (C++)

```
#define REMOVE_REDUNDANT
typedef double T;
const T EPS = 1e-7;
struct PT {
  T x, y;
  PT() {}
  PT(T x, T y) : x(x), y(y) {}
  bool operator<(const PT &rhs) const { return make_pair(y,x) < make_pair(rhs.y,rhs.x); }
  bool operator==(const PT &rhs) const { return make_pair(y,x) == make_pair(rhs.y,rhs.x); }
};
T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
  return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
  sort(pts.begin(), pts.end());
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
  vector<PT> up, dn;
  for (int i = 0; i < pts.size(); i++) {
    while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop_back();
    while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop_back();
    up.push_back(pts[i]);
    dn.push_back(pts[i]);
  }
  pts = dn;
  for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
  if (pts.size() <= 2) return;
  dn.clear();
  dn.push_back(pts[0]);
  dn.push_back(pts[1]);
  for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
    dn.push_back(pts[i]);
  }
  if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
  }
  pts = dn;
#endif
}
int main() {
    int n;
    vector<PT>pts;
    scanf("%d", &n);
    while (n--) {
         double x, y;
         scanf("%lf%lf", &x, &y);
```

```
            pts.push_back(PT(x, y));
        }
        ConvexHull(pts);
        printf("%d\n", (int)pts.size());
        for (int i = 0; i < pts.size(); i++)
            printf("%d %d\n", (int)pts[i].x, (int)pts[i].y);
}
```

Combination (C++)

```
// Combination.
#define mp make_pair
#define pb push_back
#define lmax 2147483647 //32 bit max signed!
#define lmin -2147483647 //32 bit min
#define ulmax 4294967295 //32 bit unsigned max(2^32 - 1)!
#define mx (1<<20)
#define md 1000000009
int f[mx+10]; // count factorials before use!!
/* This function calculates (a^b)%MOD */
long long pow(int a, int b, int MOD) {
    long long x = 1, y = a;
    while (b > 0) {
        if(b%2 == 1) {
            x = (x*y);
            if (x>MOD) x %= MOD;
        }
        y = (y*y);
        if (y>MOD) y%=MOD;
        b /= 2;
    }
    return x;
}

/*  Modular Multiplicative Inverse
    Using Euler's Theorem
    a^(phi(m)) = 1 (mod m)
    a^(-1) = a^(m-2) (mod m) */
long long InverseEuler(int n, int MOD){
    return pow(n, MOD-2, MOD);
}

long long C(int n, int r, int MOD) {
        if (r > n)
                return 0;
    return (f[n]*((InverseEuler(f[r], MOD) * InverseEuler(f[n-r], MOD)) % MOD)) % MOD;
}
int pw[21];
int main(int argc, char *argv[]) {
    f[0] = pw[0] = 1;
    for (int i = 1; i <= mx; i++)
        f[i] = (1ll*f[i-1]*i)%md;
    for (int i = 1; i <= 20; i++)
        pw[i] = (1<<i);
    int k;
    scanf("%d", &k);
    for (int i = 1; i <= (1<<k); i++) {
        long long a = (1ll*2*C(i-1, pw[k-1]-1, md))%md;
```

```
        long long ff = (1ll*f[pw[k-1]]*f[pw[k-1]])%md;
        long long b = ( 1ll*((1ll*a)%md)*ff)%md;
        cout<<b<<endl;
      }

    return 0;
}
```

Gauss jordan (C++)

```
// 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[][]
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;
  }
}
```

Knut morris pratt (C++)

```
typedef vector<int> VI;

void buildTable(string& w, VI& t)
{
  t = VI(w.length());
```

```
  int i = 2, j = 0;
  t[0] = -1; t[1] = 0;

  while(i < w.length())
  {
    if(w[i-1] == w[j]) { t[i] = j+1; i++; j++; }
    else if(j > 0) j = t[j];
    else { t[i] = 0; i++; }
  }
}

int KMP(string& s, string& w)
{
  int m = 0, i = 0;
  VI t;

  buildTable(w, t);
  while(m+i < s.length())
  {
    if(w[i] == s[m+i])
    {
      i++;
      if(i == w.length()) return m;
    }
    else
    {
      m += i-t[i];
      if(i > 0) i = t[i];
    }
  }
  return s.length();
}

int main()
{
  string a = (string) "The example above illustrates the general technique for assembling "+
    "the table with a minimum of fuss. The principle is that of the overall search: "+
    "most of the work was already done in getting to the current position, so very "+
    "little needs to be done in leaving it. The only minor complication is that the "+
    "logic which is correct late in the string erroneously gives non-proper "+
    "substrings at the beginning. This necessitates some initialization code.";

  string b = "table";

  int p = KMP(a, b);
  cout << p << ": " << a.substr(p, b.length()) << " " << b << endl;
}
```

LIS (C++)

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

#define STRICTLY_INCREASNG 0

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] = dad[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;
}

int main() {
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    int n;
    VI v;
    scanf("%d", &n);
    while (n--) {
      int k;
      scanf("%d", &k);
      v.push_back(k);
    }
    VI ret = LongestIncreasingSubsequence(v);
    for (int i = 0; i < ret.size(); i++)
        printf("%d ", ret[i]);
}
```

Suffix array (C++)

```
struct prefix_cmp {
        size_t prefix_len;
        size_t *rank;
        size_t n;
        bool operator() (size_t i, size_t j) {
                if( rank[i] != rank[j] ) return rank[i] < rank[j];
                i += prefix_len; j += prefix_len;
                if( i < n && j < n ) return rank[i] < rank[j];
                else                 return i > j;
        }
};

// given `string' of length `n', construct the suffix array in SA
```

```
// (SA assumed to have size >= n)
void suffix_array( size_t *string, size_t *SA, size_t n ) {
        size_t * rank[2]; rank[0] = new size_t[n]; rank[1] = new size_t[n];
        for( size_t i = 0; i < n; ++i ) { SA[i] = i; rank[0][i] = string[i]; }
        prefix_cmp cmp; cmp.n = n; cmp.prefix_len = 1;
        for( size_t x = 0;; ) {
                cmp.rank = rank[x];
                sort( SA, SA+n, cmp );
                x ^= 1;
                rank[x][SA[0]] = 0;
                for( size_t i = 0; i < n-1; ++i ) {
                        rank[x][ SA[i+1] ] = rank[x][ SA[i] ];
                        if( cmp(SA[i], SA[i+1]) ) ++rank[x][ SA[i+1] ];
                }
                cmp.prefix_len *= 2;
                if( rank[x][ SA[n-1] ] == n-1 ) break;
        }
        delete[] rank[0]; delete[] rank[1];
}
```