ICPC Team Notebook (2018-19)

Contents

1 Combinatorial optimization

1.1 Dense max-flow

```
// Adjacency matrix implementation of Dinic's blocking flow algorithm.
//
// Running time:
//     O(|V|^4)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow, look at positive values only.
typedef vector<int> VI;
typedef vector<VI> VVI;

const int INF = 1000000000;

struct MaxFlow {
  int N;
  VVI cap, flow;
  VI dad, Q;

  MaxFlow(int N) :
    N(N), cap(N, VI(N)), flow(N, VI(N)), dad(N), Q(N) {}

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

  int BlockingFlow(int s, int t) {
    fill(dad.begin(), dad.end(), -1);
    dad[s] = -2;

    int head = 0, tail = 0;
    Q[tail++] = s;
    while (head < tail) {
      int x = Q[head++];
      for (int i = 0; i < N; i++) {
        if (dad[i] == -1 && cap[x][i] - flow[x][i] > 0) {
          dad[i] = x;
          Q[tail++] = i;
        }
      }
    }

    if (dad[t] == -1) return 0;

    int totflow = 0;
    for (int i = 0; i < N; i++) {
      if (dad[i] == -1) continue;
      int amt = cap[i][t] - flow[i][t];
      for (int j = i; amt && j != s; j = dad[j])
        amt = min(amt, cap[dad[j]][j] - flow[dad[j]][j]);
      if (amt == 0) continue;
      flow[i][t] += amt;
      flow[t][i] -= amt;
      for (int j = i; j != s; j = dad[j]) {
        flow[dad[j]][j] += amt;
        flow[j][dad[j]] -= amt;
      }
      totflow += amt;
    }

    return totflow;
  }

  int GetMaxFlow(int source, int sink) {
    int totflow = 0;
    while (int flow = BlockingFlow(source, sink))
      totflow += flow;
    return totflow;
  }
};

int main() {

  MaxFlow mf(5);
  mf.AddEdge(0, 1, 3);
  mf.AddEdge(0, 2, 4);
  mf.AddEdge(0, 3, 5);
  mf.AddEdge(0, 4, 5);
  mf.AddEdge(1, 2, 2);
  mf.AddEdge(2, 3, 4);
  mf.AddEdge(2, 4, 1);
  mf.AddEdge(3, 4, 10);

  // should print out "15"
  cout << mf.GetMaxFlow(0, 4) << endl;
}

// BEGIN CUT
// The following code solves SPOJ problem #203: Potholers (POTHOLE)

#ifdef COMMENT
int main() {
  int t;
  cin >> t;
  for (int i = 0; i < t; i++) {
    int n;
    cin >> n;
    MaxFlow mf(n);
    for (int j = 0; j < n-1; j++) {
      int m;
      cin >> m;
      for (int k = 0; k < m; k++) {
        int p;
        cin >> p;
        p--;
        int cap = (j == 0 || p == n-1) ? 1 : INF;
        mf.AddEdge(j, p, cap);
      }
    }

    cout << mf.GetMaxFlow(0, n-1) << endl;
  }
  return 0;
}
#endif

// END CUT
```

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.
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 Min-cost matching

```
///////////////////////////////////////////////////////////////////////////
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs.  In practice, it solves 1000x1000 problems in around 1
// second.
//
//   cost[i][j] = cost for pairing left node i with right node j
//   Lmate[i] = index of right node that left node i pairs with
//   Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative.  To perform
// maximization, simply negate the cost[][] matrix.
///////////////////////////////////////////////////////////////////////////
typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

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

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

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

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

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

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

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

    int j = 0;
    while (true) {

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

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

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

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

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

    mated++;
  }

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

  return value;
}
```

1.4 Sparse max-flow

```
// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//     O(|V|^2 |E|)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source and sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain actual flow values, look at edges with capacity > 0
//       (zero capacity edges are residual edges).
typedef long long LL;

struct Edge {
  int u, v;
  LL cap, flow;
  Edge() {}
  Edge(int u, int v, LL cap): u(u), v(v), cap(cap), flow(0) {}
};

struct Dinic {
  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, LL cap) {
    if (u != v) {
      E.emplace_back(u, v, cap);
      g[u].emplace_back(E.size() - 1);
      E.emplace_back(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;
  }

  LL DFS(int u, int T, LL 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) {
        LL amt = e.cap - e.flow;
        if (flow != -1 && amt > flow) amt = flow;
        if (LL pushed = DFS(e.v, T, amt)) {
          e.flow += pushed;
          oe.flow -= pushed;
          return pushed;
        }
      }
    }
    return 0;
  }

  LL MaxFlow(int S, int T) {
    LL total = 0;
    while (BFS(S, T)) {
      fill(pt.begin(), pt.end(), 0);
      while (LL flow = DFS(S, T))
        total += flow;
    }
    return total;
  }
};

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (
      FASTFLOW)

int main()
{
  int N, E;
  scanf("%d%d", &N, &E);
  Dinic dinic(N);
  for(int i = 0; i < E; i++)
  {
    int u, v;
    LL cap;
    scanf("%d%d%lld", &u, &v, &cap);
    dinic.AddEdge(u - 1, v - 1, cap);
    dinic.AddEdge(v - 1, u - 1, cap);
  }
  printf("%lld\n", dinic.MaxFlow(0, N - 1));
  return 0;
}

// END CUT
```

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)
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
```

1.6 Push-relabel max-flow

```
// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic.  This implementation is
// significantly faster than straight Ford-Fulkerson.  It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//     - source
//     - sink
//
// OUTPUT:
//     - maximum flow value
//     - To obtain the actual flow values, look at all edges with
//       capacity > 0 (zero capacity edges are residual edges).
typedef long long LL;

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

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

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

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

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

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

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

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

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

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

    while (!Q.empty()) {
      int v = Q.front();
      Q.pop();
```

```
      active[v] = false;
      Discharge(v);
    }

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

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (
     FASTFLOW)

int main() {
  int n, m;
  scanf("%d%d", &n, &m);

  PushRelabel pr(n);
  for (int i = 0; i < m; i++) {
   int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    if (a == b) continue;
    pr.AddEdge(a-1, b-1, c);
    pr.AddEdge(b-1, a-1, c);
  }
  printf("%Ld\n", pr.GetMaxFlow(0, n-1));
  return 0;
}

// END CUT
```

1.7 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
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. max matching should have been found first
2. change each edge in matching into a directed edge from right to
     left
3. change each edge not used in matching into a directed edge from
     left to right
4. compute T: set of vertices reachable from unmatched vertices on the
     left (including themselves)
5. MVC = vertices cover consists of all vertices on the right that are
      in T, and all vertices on the left that are not in T
*/
// OUTPUT: <row/col idx, 0:row/1:col>
const int LEFT = 0;
const int RIGHT = 1;
vii MVC(vvi &w, vi &mr, vi &mc) {
    set<ii> T;
    queue<ii> q;
    for(int i=0;i<mr.size();i++){
        if(mr[i]==-1) {
            q.push({i, LEFT});
            T.insert({i, LEFT});
        }
    }
    while(!q.empty()) {
        ii curr = q.front(); q.pop();
        int u = curr.first;
        int type = curr.second;
        if(type == LEFT) {
            for(int v=0;v<mc.size();v++){
                ii next = {v, RIGHT};
                if(w[u][v] && mr[u]!=v && !T.count(next)) {
                    T.insert(next);
                    q.push(next);
                }
            }
        } else {
            // RIGHT
            for(int v=0;v<mr.size();v++){
                ii next = {v, LEFT};
                if(w[v][u] && mr[v]==u && !T.count(next)) {
                    T.insert(next);
                    q.push(next);
                }
            }
        }
    }

    vii mvc;
    for(int i=0;i<mr.size();i++) if(!T.count({i, LEFT})) mvc.push_back
         ({i, LEFT});
    for(int i=0;i<mc.size();i++) if(T.count({i, RIGHT})) mvc.push_back
         ({i, RIGHT});

    return mvc;
}
```

1.8 Graph cut inference

```
// Special-purpose {0,1} combinatorial optimization solver for
// problems of the following by a reduction to graph cuts:
//
//        minimize         sum_i  psi_i(x[i])
//  x[1]...x[n] in {0,1}      + sum_{i < j}  phi_{ij}(x[i], x[j])
//
// where
//      psi_i : {0, 1} --> R
//   phi_{ij} : {0, 1} x {0, 1} --> R
//
// such that
//   phi_{ij}(0,0) + phi_{ij}(1,1) <= phi_{ij}(0,1) + phi_{ij}(1,0)
//     (*)
//
// This can also be used to solve maximization problems where the
// direction of the inequality in (*) is reversed.
//
// INPUT: phi -- a matrix such that phi[i][j][u][v] = phi_{ij}(u, v)
//        psi -- a matrix such that psi[i][u] = psi_i(u)
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution
//
// To use this code, create a GraphCutInference object, and call the
// DoInference() method.  To perform maximization instead of
//      minimization,
// ensure that #define MAXIMIZATION is enabled.
typedef vector<int> VI;
typedef vector<VI> VVI;
typedef vector<VVI> VVVI;
typedef vector<VVVI> VVVVI;

const int INF = 1000000000;

// comment out following line for minimization
#define MAXIMIZATION

struct GraphCutInference {
  int N;
  VVI cap, flow;
  VI reached;

  int Augment(int s, int t, int a) {
    reached[s] = 1;
    if (s == t) return a;
    for (int k = 0; k < N; k++) {
      if (reached[k]) continue;
      if (int aa = min(a, cap[s][k] - flow[s][k])) {
        if (int b = Augment(k, t, aa)) {
          flow[s][k] += b;
          flow[k][s] -= b;
          return b;
        }
      }
    }
    return 0;
  }

  int GetMaxFlow(int s, int t) {
    N = cap.size();
    flow = VVI(N, VI(N));
    reached = VI(N);

    int totflow = 0;
    while (int amt = Augment(s, t, INF)) {
      totflow += amt;
      fill(reached.begin(), reached.end(), 0);
    }
    return totflow;
  }

  int DoInference(const VVVVI &phi, const VVI &psi, VI &x) {
    int M = phi.size();
    cap = VVI(M+2, VI(M+2));
    VI b(M);
    int c = 0;

    for (int i = 0; i < M; i++) {
      b[i] += psi[i][1] - psi[i][0];
      c += psi[i][0];
      for (int j = 0; j < i; j++)
        b[i] += phi[i][j][1][1] - phi[i][j][0][1];
      for (int j = i+1; j < M; j++) {
        cap[i][j] = phi[i][j][0][1] + phi[i][j][1][0] - phi[i][j
             ][0][0] - phi[i][j][1][1];
        b[i] += phi[i][j][1][0] - phi[i][j][0][0];
        c += phi[i][j][0][0];
      }
    }

#ifdef MAXIMIZATION
    for (int i = 0; i < M; i++) {
      for (int j = i+1; j < M; j++)
        cap[i][j] *= -1;
      b[i] *= -1;
    }
    c *= -1;
#endif

    for (int i = 0; i < M; i++) {
      if (b[i] >= 0) {
        cap[M][i] = b[i];
      } else {
        cap[i][M+1] = -b[i];
        c += b[i];
      }
    }

    int score = GetMaxFlow(M, M+1);
    fill(reached.begin(), reached.end(), 0);
    Augment(M, M+1, INF);
    x = VI(M);
    for (int i = 0; i < M; i++) x[i] = reached[i] ? 0 : 1;
    score += c;
#ifdef MAXIMIZATION
    score *= -1;
#endif

    return score;
  }

};

int main() {

  // solver for "Cat vs. Dog" from NWERC 2008

  int numcases;
```

```
  cin >> numcases;
  for (int caseno = 0; caseno < numcases; caseno++) {
    int c, d, v;
    cin >> c >> d >> v;

    VVVVI phi(c+d, VVVI(c+d, VVI(2, VI(2))));
    VVI psi(c+d, VI(2));
    for (int i = 0; i < v; i++) {
      char p, q;
      int u, v;
      cin >> p >> u >> q >> v;
      u--; v--;
      if (p == 'C') {
        phi[u][c+v][0][0]++;
        phi[c+v][u][0][0]++;
      } else {
        phi[v][c+u][1][1]++;
        phi[c+u][v][1][1]++;
      }
    }

    GraphCutInference graph;
    VI x;
    cout << graph.DoInference(phi, psi, x) << endl;
  }

  return 0;
}
```

1.9 General Matching

```
// Unweighted general matching.
// Vertices are numbered from 1 to V.
// G is an adjlist.
// G[x][0] contains the number of neighbours of x.
// The neigbours are then stored in G[x][1] .. G[x][G[x][0]].
// Mate[x] will contain the matching node for x.
// V and E are the number of edges and vertices.
// Slow Version (2x on random graphs) of Gabow's implementation
// of Edmonds' algorithm (O(V^3)).
const int MAXV = 250;
int G[MAXV][MAXV];
int VLabel[MAXV];
int  Queue[MAXV];
int   Mate[MAXV];
int   Save[MAXV];
int   Used[MAXV];
int    Up, Down;
int          V;

void ReMatch(int x, int y)
{
  int m = Mate[x]; Mate[x] = y;
  if (Mate[m] == x)
    {
      if (VLabel[x] <= V)
        {
          Mate[m] = VLabel[x];
          ReMatch(VLabel[x], m);
        }
      else
        {
          int a = 1 + (VLabel[x] - V - 1) / V;
          int b = 1 + (VLabel[x] - V - 1) % V;
          ReMatch(a, b); ReMatch(b, a);
        }
    }
}

void Traverse(int x)
{
  for (int i = 1; i <= V; i++) Save[i] = Mate[i];
  ReMatch(x, x);
  for (int i = 1; i <= V; i++)
    {
      if (Mate[i] != Save[i]) Used[i]++;
      Mate[i] = Save[i];
    }
}

void ReLabel(int x, int y)
{
  for (int i = 1; i <= V; i++) Used[i] = 0;
  Traverse(x); Traverse(y);
  for (int i = 1; i <= V; i++)
    {
      if (Used[i] == 1 && VLabel[i] < 0)
        {
          VLabel[i] = V + x + (y - 1) * V;
          Queue[Up++] = i;
        }
    }
}

// Call this after constructing G
void Solve()
{
  for (int i = 1; i <= V; i++)
    if (Mate[i] == 0)
      {
        for (int j = 1; j <= V; j++) VLabel[j] = -1;
        VLabel[i] = 0; Down = 1; Up = 1; Queue[Up++] = i;
        while (Down != Up)
          {
            int x = Queue[Down++];
            for (int p = 1; p <= G[x][0]; p++)
              {
                int y = G[x][p];
                if (Mate[y] == 0 && i != y)
                  {
                    Mate[y] = x; ReMatch(x, y);
                    Down = Up; break;
                  }
                if (VLabel[y] >= 0)
                  {
                    ReLabel(x, y);
                    continue;
                  }
                if (VLabel[Mate[y]] < 0)
                  {
                    VLabel[Mate[y]] = x;
                    Queue[Up++] = Mate[y];
                  }
              }
          }
      }
}

// Call this after Solve(). Returns number of edges in matching (half
      the number of matched vertices)
int Size()
{
  int Count = 0;
  for (int i = 1; i <= V; i++)
    if (Mate[i] > i) Count++;
  return Count;
}
```

1.10 Edmond Blossom Matching

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

class edmond_blossom {
private:
    std::vector<std::vector<int>> adj_list;
    std::vector<int> match;
    std::vector<int> parent;
    std::vector<int> blossom_root;
    std::vector<bool> in_queue;
    std::vector<bool> in_blossom;
    std::queue<int> process_queue;
    int num_v;

    /// Find the lowest common ancestor between u and v.
    /// The specified root represents an upper bound.
    int get_lca(int root, int u, int v) {
        std::vector<bool> in_path(num_v, false);

        for (u = blossom_root[u]; ; u = blossom_root[parent[match[u
              ]]]) {
            in_path[u] = true;
            if (u == root) {
                break;
            }
        }

        for (v = blossom_root[v]; ; v = blossom_root[parent[match[v
              ]]]) {
            if (in_path[v]) {
                return v;
            }
        }
    }

    /// Mark the vertices between u and the specified lowest
    /// common ancestor for contraction where necessary.
    void mark_blossom(int lca, int u) {
        while (blossom_root[u] != lca) {
            int v = match[u];
            in_blossom[blossom_root[u]] = true;
            in_blossom[blossom_root[v]] = true;

            u = parent[v];
            if (blossom_root[u] != lca) {
                parent[u] = v;
            }
        }
    }

    /// Contract the blossom that is formed after processing
    /// the edge u-v.
    void contract_blossom(int source, int u, int v) {
        int lca = get_lca(source, u, v);
        std::fill(in_blossom.begin(), in_blossom.end(), false);

        mark_blossom(lca, u);
        mark_blossom(lca, v);

        if (blossom_root[u] != lca) {
            parent[u] = v;
        }
        if (blossom_root[v] != lca) {
            parent[v] = u;
        }

        for (int i = 0; i < num_v; ++i) {
            if (in_blossom[blossom_root[i]]) {
                blossom_root[i] = lca;
                if (!in_queue[i]) {
                    process_queue.push(i);
                    in_queue[i] = true;
                }
            }
        }
    }

    /// Return the vertex at the end of an augmenting path
    /// starting at the specified source, or -1 if none exist.
    int find_augmenting_path(int source) {
        for (int i = 0; i < num_v; ++i) {
            in_queue[i] = false;
            parent[i] = -1;
            blossom_root[i] = i;
        }

        // Empty the queue
        process_queue = std::queue<int>();

        process_queue.push(source);
        in_queue[source] = true;

        while (!process_queue.empty()) {
            int u = process_queue.front();
            process_queue.pop();

            for (int v : adj_list[u]) {
                if (blossom_root[u] != blossom_root[v] && match[u] !=
                      v) {
                    // Process if
                    // + u-v is not an edge in the matching
                    // && u and v are not in the same blossom (yet)
                    if (v == source || (match[v] != -1 && parent[match
                          [v]] != -1)) {
                        // Contract a blossom if
                        // + v is the source
                        // || v is matched and v's match has a parent.
                        //
                        // The fact that parents are assigned to
                              vertices
                        // with odd distances from the source is used
                              to
                        // check if a cycle is odd or even. u is
                              always an
                        // even distance away from the source, so if v
```

```
                        's
                    // match is assigned a parent, you have an odd
                        cycle.
                    contract_blossom(source, u, v);
                } else if (parent[v] == -1) {
                    parent[v] = u;

                    if (match[v] == -1) {
                        // v is unmatched; augmenting path found
                        return v;
                    } else {
                        // Enqueue v's match.
                        int w = match[v];
                        if (!in_queue[w]) {
                            process_queue.push(w);
                            in_queue[w] = true;
                        }
                    }
                }
            }
        }
    }

    return -1;
}

/// Augment the path that ends with the specified vertex
/// using the parent and match fields. Returns the increase
/// in the number of matchings. (i.e. 1 if the path is valid,
/// 0 otherwise)
int augment_path(int end) {
    int u = end;
    while (u != -1) {
        // Currently w===v----u
        int v = parent[u];
        int w = match[v];

        // Change to w---v===u
        match[v] = u;
        match[u] = v;

        u = w;
    }

    // Return 1 if the augmenting path is valid
    return end == -1 ? 0 : 1;
}

public:

    edmond_blossom(int v) :
        adj_list(v),
        match(v, -1),
        parent(v),
        blossom_root(v),
        in_queue(v),
        in_blossom(v),
        num_v(v) {}


    /// Add a bidirectional edge from u to v.
    void add_edge(int u, int v) {
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
    }

    /// Returns the maximum cardinality matching
    int get_max_matching() {
        int ans = 0;

        // Reset
        std::fill(match.begin(), match.end(), -1);

        for (int u = 0; u < num_v; ++u) {
            if (match[u] == -1) {
                int v = find_augmenting_path(u);
                if (v != -1) {
                    // An augmenting path exists
                    ans += augment_path(v);
                }
            }
        }

        return ans;
    }

    /// Constructs the maximum cardinality matching
    std::vector<std::pair<int, int>> construct_matching() {
        std::vector<std::pair<int, int>> output;
        std::vector<bool> is_processed(num_v, false);

        for (int u = 0; u < num_v; ++u) {
            if (!is_processed[u] && match[u] != -1) {
                output.emplace_back(u, match[u]);
                is_processed[u] = true;
                is_processed[match[u]] = true;
            }
        }

        return output;
    }
};

int main() {
    /*
    10 18
    0 1
    0 2
    1 2
    1 3
    1 4
    3 4
    2 4
    2 5
    4 5
    3 6
    3 7
    4 7
    4 8
    5 8
    5 9
    6 7
    7 8
    8 9
    */
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);

    int num_vertices;
    int num_edges;

    std::cin >> num_vertices >> num_edges;

    edmond_blossom eb(num_vertices);
    for (int i = 0; i < num_edges; ++i) {
        int u, v;
        std::cin >> u >> v;

        eb.add_edge(u, v);
    }

    std::cout << "Maximum Cardinality: " << eb.get_max_matching() << "
        \n";

    std::vector<std::pair<int, int>> matching = eb.construct_matching
        ();
    for (auto& match : matching) {
        std::cout << match.first << " " << match.second << "\n";
    }

}
```

1.11 Min Edge Cover

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

1.12 Stable Marriage Problem

```
// Gale-Shapley algorithm for the stable marriage problem.
// madj[i][j] is the jth highest ranked woman for man i.
// fpref[i][j] is the rank woman i assigns to man j.
// Returns a pair of vectors (mpart, fpart), where mpart[i] gives the
    partner of man i, and fpart is analogous
pair<vector<int>, vector<int> > stable_marriage(vector<vector<int> >&
    madj, vector<vector<int> >& fpref) {
        int n = madj.size();
        vector<int> mpart(n, -1), fpart(n, -1);
        vector<int> midx(n);
        queue<int> mfree;
        for (int i = 0; i < n; i++) {
                mfree.push(i);
        }
        while (!mfree.empty()) {
                int m = mfree.front(); mfree.pop();
                int f = madj[m][midx[m]++];
                if (fpart[f] == -1) {
                        mpart[m] = f; fpart[f] = m;
                } else if (fpref[f][m] < fpref[f][fpart[f]]) {
                        mpart[fpart[f]] = -1; mfree.push(fpart[f]);
                        mpart[m] = f; fpart[f] = m;
                } else {
                        mfree.push(m);
                }
        }
        return make_pair(mpart, fpart);
}
```

2 Geometry

2.1 Lines

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

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0) // important constant; alternative #define PI
    (2.0 * acos(0.0))

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

// struct point_i { int x, y; };    // basic raw form, minimalist mode
struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                       // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };        // user-defined

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                       // default constructor
  point(double _x, double _y) : x(_x), y(_y) {}       // user-defined
  bool operator < (point other) const { // override less than operator
    if (fabs(x - other.x) > EPS)                 // useful for sorting
      return x < other.x;          // first criteria , by x-coordinate
    return y < other.y; }          // second criteria, by y-coordinate
  // use EPS (1e-9) when testing equality of two floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

double dist(point p1, point p2) {                // Euclidean distance
                      // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
  return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);    // multiply theta with PI / 180.0
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }

struct line { double a, b, c; };          // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;           // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }
```

```
// not needed since we will use the more robust form: ax + by + c = 0
      (see above)
struct line2 { double m, c; };      // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {
 if (abs(p1.x - p2.x) < EPS) {          // special case: vertical line
   l.m = INF;                     // l contains m = INF and c = x_value
   l.c = p1.x;                   // to denote vertical line x = x_value
   return 0;   // we need this return variable to differentiate result
 }
 else {
   l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
   l.c = p1.y - l.m * p1.x;
   return 1;     // l contains m and c of the line equation y = mx + c
} }

bool areParallel(line l1, line l2) {       // check coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {           // also check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;            // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }

struct vec { double x, y;  // name: 'vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }                 // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }

// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
  l.a = -m;                                               // always -m
  l.b = 1;                                                 // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }                 // compute this

void closestPoint(line l, point p, point &ans) {
  line perpendicular;         // perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {              // special case 1: vertical line
    ans.x = -(l.c);   ans.y = p.y;      return; }

  if (fabs(l.a) < EPS) {            // special case 2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);   return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular);          // normal line
  // intersect line l with this perpendicular line
  // the intersection point is the closest point
  areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
  point b;
  closestPoint(l, p, b);                      // similar to distToLine
  vec v = toVec(p, b);                              // create a vector
  ans = translate(translate(p, v), v); }          // translate p twice

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

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

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));                   // translate a to c
  return dist(p, c); }           // Euclidean distance between p and c

// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);                   // closer to a
    return dist(p, a); }          // Euclidean distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);                   // closer to b
    return dist(p, b); }          // Euclidean distance between p and b
  return distToLine(p, a, b, c); }          // run distToLine as above

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

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

//// another variant
//int area2(point p, point q, point r) { // returns 'twice' the area
      of this triangle A-B-c
//  return p.x * q.y - p.y * q.x +
//         q.x * r.y - q.y * r.x +
//         r.x * p.y - r.y * p.x;
//}

// note: to accept collinear points, we have to change the '> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

int main() {
  point P1, P2, P3(0, 1); // note that both P1 and P2 are (0.00, 0.00)
  printf("%d\n", P1 == P2);                                    // true
  printf("%d\n", P1 == P3);                                   // false

  vector<point> P;
  P.push_back(point(2, 2));
  P.push_back(point(4, 3));
  P.push_back(point(2, 4));
  P.push_back(point(6, 6));
  P.push_back(point(2, 6));
  P.push_back(point(6, 5));

  // sorting points demo
  sort(P.begin(), P.end());
  for (int i = 0; i < (int)P.size(); i++)
    printf("(%.2lf, %.2lf)\n", P[i].x, P[i].y);

  // rearrange the points as shown in the diagram below
  P.clear();
  P.push_back(point(2, 2));
  P.push_back(point(4, 3));
  P.push_back(point(2, 4));
  P.push_back(point(6, 6));
  P.push_back(point(2, 6));
  P.push_back(point(6, 5));
  P.push_back(point(8, 6));

  /*
  // the positions of these 7 points (0-based indexing)
  6   P4      P3  P6
  5           P5
  4   P2
  3       P1
  2   P0
  1
  0 1 2 3 4 5 6 7 8
  */

  double d = dist(P[0], P[5]);
  printf("Euclidean distance between P[0] and P[5] = %.2lf\n", d); //
        should be 5.000

  // line equations
  line l1, l2, l3, l4;
  pointsToLine(P[0], P[1], l1);
  printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l1.a, l1.b, l1.c);
        // should be -0.50 * x + 1.00 * y - 1.00 = 0.00

  pointsToLine(P[0], P[2], l2); // a vertical line, not a problem in "
        ax + by + c = 0" representation
  printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l2.a, l2.b, l2.c);
        // should be 1.00 * x + 0.00 * y - 2.00 = 0.00

  // parallel, same, and line intersection tests
  pointsToLine(P[2], P[3], l3);
  printf("l1 & l2 are parallel? %d\n", areParallel(l1, l2)); // no
  printf("l1 & l3 are parallel? %d\n", areParallel(l1, l3)); // yes,
        l1 (P[0]-P[1]) and l3 (P[2]-P[3]) are parallel

  pointsToLine(P[2], P[4], l4);
  printf("l1 & l2 are the same? %d\n", areSame(l1, l2)); // no
  printf("l2 & l4 are the same? %d\n", areSame(l2, l4)); // yes, l2 (P
        [0]-P[2]) and l4 (P[2]-P[4]) are the same line (note, they are
        two different line segments, but same line)

  point p12;
  bool res = areIntersect(l1, l2, p12); // yes, l1 (P[0]-P[1]) and l2
        (P[0]-P[2]) are intersect at (2.0, 2.0)
  printf("l1 & l2 are intersect? %d, at (%.2lf, %.2lf)\n", res, p12.x,
        p12.y);

  // other distances
  point ans;
  d = distToLine(P[0], P[2], P[3], ans);
  printf("Closest point from P[0] to line         (P[2]-P[3]): (%.2lf,
        %.2lf), dist = %.2lf\n", ans.x, ans.y, d);
  closestPoint(l3, P[0], ans);
  printf("Closest point from P[0] to line V2      (P[2]-P[3]): (%.2lf,
        %.2lf), dist = %.2lf\n", ans.x, ans.y, dist(P[0], ans));

  d = distToLineSegment(P[0], P[2], P[3], ans);
  printf("Closest point from P[0] to line SEGMENT (P[2]-P[3]): (%.2lf,
        %.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to A (or
        P[2]) = (2.00, 4.00)
  d = distToLineSegment(P[1], P[2], P[3], ans);
  printf("Closest point from P[1] to line SEGMENT (P[2]-P[3]): (%.2lf,
        %.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to
       midway between AB = (3.20, 4.60)
  d = distToLineSegment(P[6], P[2], P[3], ans);
  printf("Closest point from P[6] to line SEGMENT (P[2]-P[3]): (%.2lf,
        %.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to B (or
        P[3]) = (6.00, 6.00)

  reflectionPoint(l4, P[1], ans);
  printf("Reflection point from P[1] to line     (P[2]-P[4]): (%.2lf,
        %.2lf)\n", ans.x, ans.y); // should be (0.00, 3.00)

  printf("Angle P[0]-P[4]-P[3] = %.2lf\n", RAD_to_DEG(angle(P[0], P
        [4], P[3]))); // 90 degrees
  printf("Angle P[0]-P[2]-P[1] = %.2lf\n", RAD_to_DEG(angle(P[0], P
        [2], P[1]))); // 63.43 degrees
  printf("Angle P[4]-P[3]-P[6] = %.2lf\n", RAD_to_DEG(angle(P[4], P
        [3], P[6]))); // 180 degrees

  printf("P[0], P[2], P[3] form A left turn? %d\n", ccw(P[0], P[2], P
        [3])); // no
  printf("P[0], P[3], P[2] form A left turn? %d\n", ccw(P[0], P[3], P
        [2])); // yes

  printf("P[0], P[2], P[3] are collinear? %d\n", collinear(P[0], P[2],
        P[3])); // no
  printf("P[0], P[2], P[4] are collinear? %d\n", collinear(P[0], P[2],
        P[4])); // yes

  point p(3, 7), q(11, 13), r(35, 30); // collinear if r(35, 31)
  printf("r is on the %s of line p-r\n", ccw(p, q, r) ? "left" : "
        right"); // right

  /*
  // the positions of these 6 points
     E<--  4
           3       B D<--
           2   A C
           1
  -4-3-2-1 0 1 2 3 4 5 6
          -1
          -2
   F<--   -3
  */

  // translation
  point A(2.0, 2.0);
  point B(4.0, 3.0);
  vec v = toVec(A, B); // imagine there is an arrow from A to B (see
        the diagram above)
  point C(3.0, 2.0);
  point D = translate(C, v); // D will be located in coordinate (3.0 +
        2.0, 2.0 + 1.0) = (5.0, 3.0)
  printf("D = (%.2lf, %.2lf)\n", D.x, D.y);
  point E = translate(C, scale(v, 0.5)); // E will be located in
        coordinate (3.0 + 1/2 * 2.0, 2.0 + 1/2 * 1.0) = (4.0, 2.5)
  printf("E = (%.2lf, %.2lf)\n", E.x, E.y);

  // rotation
  printf("B = (%.2lf, %.2lf)\n", B.x, B.y); // B = (4.0, 3.0)
  point F = rotate(B, 90); // rotate B by 90 degrees COUNTER clockwise
        , F = (-3.0, 4.0)
```

```
  printf("F = (%.2lf, %.2lf)\n", F.x, F.y);
  point G = rotate(B, 180); // rotate B by 180 degrees COUNTER
        clockwise, G = (-4.0, -3.0)
  printf("G = (%.2lf, %.2lf)\n", G.x, G.y);

  return 0;
}
```

2.2 Circles

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

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                        // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };            // constructor

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                        // default constructor
  point(double _x, double _y) : x(_x), y(_y) {} };      // constructor

int insideCircle(point_i p, point_i c, int r) { // all integer version
  int dx = p.x - c.x, dy = p.y - c.y;
  int Euc = dx * dx + dy * dy, rSq = r * r;              // all integer
  return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } //inside/border/outside

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

int main() {
  // circle equation, inside, border, outside
  point_i pt(2, 2);
  int r = 7;
  point_i inside(8, 2);
  printf("%d\n", insideCircle(inside, pt, r));             // 0-inside
  point_i border(9, 2);
  printf("%d\n", insideCircle(border, pt, r));          // 1-at border
  point_i outside(10, 2);
  printf("%d\n", insideCircle(outside, pt, r));           // 2-outside

  double d = 2 * r;
  printf("Diameter = %.2lf\n", d);
  double c = PI * d;
  printf("Circumference (Perimeter) = %.2lf\n", c);
  double A = PI * r * r;
  printf("Area of circle = %.2lf\n", A);

  printf("Length of arc   (central angle = 60 degrees) = %.2lf\n",
        60.0 / 360.0 * c);
  printf("Length of chord (central angle = 60 degrees) = %.2lf\n",
        sqrt((2 * r * r) * (1 - cos(DEG_to_RAD(60.0)))));
  printf("Area of sector  (central angle = 60 degrees) = %.2lf\n",
        60.0 / 360.0 * A);

  point p1;
  point p2(0.0, -1.0);
  point ans;
  circle2PtsRad(p1, p2, 2.0, ans);
  printf("One of the center is (%.2lf, %.2lf)\n", ans.x, ans.y);
  circle2PtsRad(p2, p1, 2.0, ans);     // we simply reverse p1 with p2
  printf("The other center  is (%.2lf, %.2lf)\n", ans.x, ans.y);

  return 0;
}
```

2.3 Triangles

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

#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                        // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };            // constructor

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                        // default constructor
  point(double _x, double _y) : x(_x), y(_y) {} };      // constructor

double dist(point p1, point p2) {
  return hypot(p1.x - p2.x, p1.y - p2.y); }

double perimeter(double ab, double bc, double ca) {
  return ab + bc + ca; }

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

double area(double ab, double bc, double ca) {
  // Heron's formula, split sqrt(a * b) into sqrt(a) * sqrt(b); in
        implementation
  double s = 0.5 * perimeter(ab, bc, ca);
  return sqrt(s) * sqrt(s - ab) * sqrt(s - bc) * sqrt(s - ca); }

double area(point a, point b, point c) {
  return area(dist(a, b), dist(b, c), dist(c, a)); }

//====================================================================
// from ch7_01_points_lines
struct line { double a, b, c; }; // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;            // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

bool areParallel(line l1, line l2) {        // check coefficient a + b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;            // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }

struct vec { double x, y;  // name: 'vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }
//====================================================================

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

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

// assumption: the required points/lines functions have been written
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0;                     // no inCircle center

  line l1, l2;                    // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);

  areIntersect(l1, l2, ctr);           // get their intersection point
  return 1; }

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

double rCircumCircle(point a, point b, point c) {
  return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written
// returns 1 if there is a circumCenter center, returns 0 otherwise
// if this function returns 1, ctr will be the circumCircle center
// and r is the same as rCircumCircle
int circumCircle(point p1, point p2, point p3, point &ctr, double &r){
  double a = p2.x - p1.x, b = p2.y - p1.y;
  double c = p3.x - p1.x, d = p3.y - p1.y;
  double e = a * (p1.x + p2.x) + b * (p1.y + p2.y);
  double f = c * (p1.x + p3.x) + d * (p1.y + p3.y);
  double g = 2.0 * (a * (p3.y - p2.y) - b * (p3.x - p2.x));
  if (fabs(g) < EPS) return 0;

  ctr.x = (d*e - b*f) / g;
  ctr.y = (a*f - c*e) / g;
  r = dist(p1, ctr);  // r = distance from center to 1 of the 3 points
  return 1; }

// returns true if point d is inside the circumCircle defined by a,b,c
int inCircumCircle(point a, point b, point c, point d) {
  return (a.x - d.x) * (b.y - d.y) * ((c.x - d.x) * (c.x - d.x) + (c.y
        - d.y) * (c.y - d.y)) +
         (a.y - d.y) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y
           - d.y)) * (c.x - d.x) +
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.
           x - d.x) * (c.y - d.y) -
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.
           y - d.y) * (c.x - d.x) -
         (a.y - d.y) * (b.x - d.x) * ((c.x - d.x) * (c.x - d.x) + (c.y
           - d.y) * (c.y - d.y)) -
         (a.x - d.x) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y
           - d.y)) * (c.y - d.y) > 0 ? 1 : 0;
}

bool canFormTriangle(double a, double b, double c) {
  return (a + b > c) && (a + c > b) && (b + c > a); }

int main() {
  double base = 4.0, h = 3.0;
  double A = 0.5 * base * h;
  printf("Area = %.2lf\n", A);

  point a;                                           // a right triangle
  point b(4.0, 0.0);
  point c(4.0, 3.0);

  double p = perimeter(a, b, c);
  double s = 0.5 * p;
  A = area(a, b, c);
  printf("Area = %.2lf\n", A);               // must be the same as above

  double r = rInCircle(a, b, c);
  printf("R1 (radius of incircle) = %.2lf\n", r);               // 1.00
  point ctr;
  int res = inCircle(a, b, c, ctr, r);
  printf("R1 (radius of incircle) = %.2lf\n", r);         // same, 1.00
  printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y);   // (3.00, 1.00)

  printf("R2 (radius of circumcircle) = %.2lf\n", rCircumCircle(a, b,
        c)); // 2.50
  res = circumCircle(a, b, c, ctr, r);
  printf("R2 (radius of circumcircle) = %.2lf\n", r);   // same, 2.50
  printf("Center = (%.2lf, %.2lf)\n", ctr.x, ctr.y);   // (2.00, 1.50)
```

```
  point d(2.0, 1.0);                // inside triangle and circumCircle
  printf("d inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, d));
  point e(2.0, 3.9);   // outside the triangle but inside circumCircle
  printf("e inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, e));
  point f(2.0, -1.1);                                 // slightly outside
  printf("f inside circumCircle (a, b, c) ? %d\n", inCircumCircle(a, b, c, f));

  // Law of Cosines
  double ab = dist(a, b);
  double bc = dist(b, c);
  double ca = dist(c, a);
  double alpha = RAD_to_DEG(acos((ca * ca + ab * ab - bc * bc) / (2.0 * ca * ab)));
  printf("alpha = %.2lf\n", alpha);
  double beta  = RAD_to_DEG(acos((ab * ab + bc * bc - ca * ca) / (2.0 * ab * bc)));
  printf("beta  = %.2lf\n", beta);
  double gamma = RAD_to_DEG(acos((bc * bc + ca * ca - ab * ab) / (2.0 * bc * ca)));
  printf("gamma = %.2lf\n", gamma);

  // Law of Sines
  printf("%.2lf == %.2lf == %.2lf\n", bc / sin(DEG_to_RAD(alpha)), ca / sin(DEG_to_RAD(beta)), ab / sin(DEG_to_RAD(gamma)));

  // Phytagorean Theorem
  printf("%.2lf^2 == %.2lf^2 + %.2lf^2\n", ca, ab, bc);

  // Triangle Inequality
  printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 5,
      canFormTriangle(3, 4, 5)); // yes
  printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 7,
      canFormTriangle(3, 4, 7)); // no, actually straight line
  printf("(%d, %d, %d) => can form triangle? %d\n", 3, 4, 8,
      canFormTriangle(3, 4, 8)); // no

  return 0;
}
```

2.4 Polygon

```
#include <algorithm>
#include <cstdio>
#include <cmath>
#include <stack>
#include <vector>
using namespace std;

#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                      // default constructor
  point(double _x, double _y) : x(_x), y(_y) {}        // user-defined
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

struct vec { double x, y;  // name: 'vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

double dist(point p1, point p2) {                // Euclidean distance
  return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double

// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)  // remember that P[0] = P[n-1]
    result += dist(P[i], P[i+1]);
  return result; }

// returns the area, which is half the determinant
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }

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

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

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

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

// note: to accept collinear points, we have to change the '> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

// returns true if we always make the same turn while examining
// all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false;   // a point/sz=2 or a line/sz=3 is not convex
  bool isLeft = ccw(P[0], P[1], P[2]);               // remember one result
  for (int i = 1; i < sz-1; i++)            // then compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;            // different sign -> this polygon is concave
  return true; }                                   // this polygon is convex

// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;    // assume the first vertex is equal to the last vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
         sum += angle(P[i], pt, P[i+1]);                   // left turn/ccw
    else sum -= angle(P[i], pt, P[i+1]); }                 // right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }

// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs(a * p.x + b * p.y + c);
  double v = fabs(a * q.x + b * q.y + c);
  return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v)); }

// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);       // Q[i] is on the left of ab
    if (left1 * left2 < -EPS)        // edge (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());        // make P's first point = P's last point
  return P; }

point pivot;
bool angleCmp(point a, point b) {                 // angle-sorting function
  if (collinear(pivot, a, b))                               // special case
    return dist(pivot, a) < dist(pivot, b);    // check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }   // compare two angles

vector<point> CH(vector<point> P) {   // the content of P may be reshuffled
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner case
    return P;                           // special case, the CH is P itself
  }

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;

  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;    // swap P[P0] with P[0]

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];                    // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp);              // we do not sort P[0]

  // third, the ccw tests
  vector<point> S;
  S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);   // initial S
  i = 2;                                         // then, we check the rest
  while (i < n) {           // note: N must be >= 3 for this method to work
    j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);  // left turn, accept
    else S.pop_back(); }   // or pop the top of S until we have a left turn
  return S; }                                          // return the result

int main() {
  // 6 points, entered in counter clockwise order, 0-based indexing
  vector<point> P;
  P.push_back(point(1, 1));
  P.push_back(point(3, 3));
  P.push_back(point(9, 1));
  P.push_back(point(12, 4));
  P.push_back(point(9, 7));
  P.push_back(point(1, 7));
  P.push_back(P[0]); // loop back

  printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // 31.64
  printf("Area of polygon = %.2lf\n", area(P)); // 49.00
  printf("Is convex = %d\n", isConvex(P)); // false (P1 is the culprit)

  //// the positions of P6 and P7 w.r.t the polygon
  //7 P5--------------P4
  //6 |                  \
  //5 |                    \
  //4 |   P7                 P3
  //3 |   P1___            /
  //2 | / P6    \ ___    /
  //1 P0              P2
  //0 1 2 3 4 5 6 7 8 9 101112

  point P6(3, 2); // outside this (concave) polygon
  printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P)); // false
  point P7(3, 4); // inside this (concave) polygon
  printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P)); // true

  // cutting the original polygon based on line P[2] -> P[4] (get the left side)
  //7 P5--------------P4
  //6 |               |  \
  //5 |               |    \
  //4 |               |     P3
  //3 |   P1___       |    /
```

```
  //2 | /       \ ___ |  /
  //1 P0              P2
  //0 1 2 3 4 5 6 7 8 9 101112
  // new polygon (notice the index are different now):
  //7 P4--------------P3
  //6 |               |
  //5 |               |
  //4 |               |
  //3 |   P1___       |
  //2 | /       \ ___ |
  //1 P0              P2
  //0 1 2 3 4 5 6 7 8 9

  P = cutPolygon(P[2], P[4], P);
  printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // smaller
       now 29.15
  printf("Area of polygon = %.2lf\n", area(P)); // 40.00

  // running convex hull of the resulting polygon (index changes again
       )
  //7 P3--------------P2
  //6 |               |
  //5 |               |
  //4 |   P7          |
  //3 |               |
  //2 |               |
  //1 P0--------------P1
  //0 1 2 3 4 5 6 7 8 9

  P = CH(P); // now this is a rectangle
  printf("Perimeter of polygon = %.2lf\n", perimeter(P)); // precisely
        28.00
  printf("Area of polygon = %.2lf\n", area(P)); // precisely 48.00
  printf("Is convex = %d\n", isConvex(P)); // true
  printf("Point P6 is inside this polygon = %d\n", inPolygon(P6, P));
       // true
  printf("Point P7 is inside this polygon = %d\n", inPolygon(P7, P));
       // true

  return 0;
}
```

2.5 Convex hull

```
// Compute the 2D convex hull of a set of points using the monotone
     chain
// algorithm.  Eliminate redundant points from the hull if
     REMOVE_REDUNDANT is
// #defined.
//
// Running time: O(n log n)
//
//   INPUT:   a vector of input points, unordered.
//   OUTPUT:  a vector of points in the convex hull, counterclockwise,
      starting
//            with bottommost/leftmost point

#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
}

// BEGIN CUT
// The following code solves SPOJ problem #26: Build the Fence (BSHEEP
     )

int main() {
  int t;
  scanf("%d", &t);
  for (int caseno = 0; caseno < t; caseno++) {
    int n;
    scanf("%d", &n);
    vector<PT> v(n);
    for (int i = 0; i < n; i++) scanf("%lf%lf", &v[i].x, &v[i].y);
    vector<PT> h(v);
    map<PT,int> index;
    for (int i = n-1; i >= 0; i--) index[v[i]] = i+1;
    ConvexHull(h);

    double len = 0;
    for (int i = 0; i < h.size(); i++) {
      double dx = h[i].x - h[(i+1)%h.size()].x;
      double dy = h[i].y - h[(i+1)%h.size()].y;
      len += sqrt(dx*dx+dy*dy);
    }

    if (caseno > 0) printf("\n");
    printf("%.2f\n", len);
    for (int i = 0; i < h.size(); i++) {
      if (i > 0) printf(" ");
      printf("%d", index[h[i]]);
    }
    printf("\n");
  }
}

// END CUT
```

2.6 3D Convex Hull

```
const double eps = 1e-8;
int mark[1005][1005];
Point info[1005];
int n, cnt;
double mix(const Point &a, const Point &b, const Point &c) {
    return a.dot(b.cross(c));}
double area(int a, int b, int c) {
    return ((info[b] - info[a]).cross(info[c] - info[a])).length();}
double volume(int a, int b, int c, int d) {
    return mix(info[b] - info[a], info[c] - info[a], info[d] - info[a
          ]);}
struct Face {
    int a, b, c;
    Face() {}
    Face(int a, int b, int c): a(a), b(b), c(c) {}
    int &operator [](int k) { return k==0?a:k==1?b:c; }
};
vector <Face> face;
inline void insert(int a, int b, int c) { face.push_back(Face(a, b, c)
     );}
void add(int v) {
    vector <Face> tmp;
    int a, b, c;
    cnt++;
    for (int i = 0; i < SIZE(face); i++) {
        a = face[i][0];  b = face[i][1];  c = face[i][2];
        if (Sign(volume(v, a, b, c)) < 0)
            mark[a][b] = mark[b][a] = mark[b][c] = mark[c][b] = mark[c
                 ][a] = mark[a][c] = cnt;
        else tmp.push_back(face[i]);
    }
    face = tmp;
    for (int i = 0; i < SIZE(tmp); i++) {
        a = face[i][0]; b = face[i][1]; c = face[i][2];
        if (mark[a][b] == cnt) insert(b, a, v);
        if (mark[b][c] == cnt) insert(c, b, v);
        if (mark[c][a] == cnt) insert(a, c, v);
    }
}
int Find() {
    for (int i = 2; i < n; i++) {
        Point ndir = (info[0] - info[i]).cross(info[1] - info[i]);
        if (ndir == Point()) continue;
        swap(info[i], info[2]);
        for (int j = i + 1; j < n; j++)
            if (Sign(volume(0, 1, 2, j)) != 0) {
                swap(info[j], info[3]);
                insert(0, 1, 2);  insert(0, 2, 1);
                return 1;
            }
    }
    return 0;
}
int main() {
    for (; scanf("%d", &n) == 1; ) {
        for (int i = 0; i < n; i++)
            info[i].Input();
        sort(info, info + n);
        n = unique(info, info + n) - info;
        face.clear();
        random_shuffle(info, info + n);
        if (Find()) {
            memset(mark, 0, sizeof(mark));
            cnt = 0;
            for (int i = 3; i < n; i++) add(i);
            vector<Point> Ndir;
            for (int i = 0; i < SIZE(face); ++i) {
                Point p = (info[face[i][0]] - info[face[i][1]]).cross(
                      info[face[i][2]] - info[face[i][1]]);
                p = p / p.length();
                Ndir.push_back(p);
            }
            sort(Ndir.begin(), Ndir.end());
            int ans = unique(Ndir.begin(), Ndir.end()) - Ndir.begin();
            printf("%d\n", ans);
        } else {
            printf("1\n");
        }
    }
}
```

2.7 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
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;
}
```

2.8 Closest Pair

```
// Source: e-maxx.ru
#define upd_ans(x, y) {}
#define MAXN 100
double mindist = 1e20; // will be the result
void rec(int l, int r, Point a[]) {
    if (r - l <= 3) {
        for (int i=l; i<=r; ++i)
            for (int j=i+1; j<=r; ++j)
                    upd_ans(a[i], a[j]);
        sort(a+l, a+r+1); // compare by y
        return;
    }

    int m = (l + r) >> 1;
    int midx = a[m].x;
    rec(l, m, a), rec(m+1, r, a);
    static Point t[MAXN];
    merge(a+l, a+m+1, a+m+1, a+r+1, t); // compare by y
    copy(t, t+r-l+1, a+l);

    int tsz = 0;
    for (int i=l; i<=r; ++i)
        if (fabs(a[i].x - midx) < mindist) {
            for (int j=tsz-1; j>=0 && a[i].y - t[j].y < mindist; --j)
                upd_ans(a[i], t[j]);
            t[tsz++] = a[i];
        }
}
```

2.9 Rotating Caliphers

```
// Rotating calipers
double convex_diameter(Polygon pt) {
    const int n = pt.size();
    int is = 0, js = 0;
    for (int i = 1; i < n; ++i) {
        if (pt[i].y > pt[is].y) is = i;
        if (pt[i].y < pt[js].y) js = i;
    }
    double maxd = (pt[is]-pt[js]).norm();
    int i, maxi, j, maxj;
    i = maxi = is;
    j = maxj = js;
    do {
        int jj = j+1; if (jj == n) jj = 0;
        if ((pt[i] - pt[jj]).norm() > (pt[i] - pt[j]).norm()) j = (j
             +1) % n;
        else i = (i+1) % n;
        if ((pt[i]-pt[j]).norm() > maxd) {
            maxd = (pt[i]-pt[j]).norm();
            maxi = i; maxj = j;
        }
    } while (i != is || j != js);
    return maxd; /* farthest pair is (maxi, maxj). */
}
```

3 Numerical algorithms

3.1 Fast exponentiation

```
/*
Uses powers of two to exponentiate numbers and matrices. Calculates
n^k in O(log(k)) time when n is a number. If A is an n x n matrix,
calculates A^k in O(n^3*log(k)) time.
*/
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T power(T x, int k) {
  T ret = 1;

  while(k) {
    if(k & 1) ret *= x;
    k >>= 1; x *= x;
  }
  return ret;
}

VVT multiply(VVT& A, VVT& B) {
  int n = A.size(), m = A[0].size(), k = B[0].size();
  VVT C(n, VT(k, 0));

  for(int i = 0; i < n; i++)
    for(int j = 0; j < k; j++)
      for(int l = 0; l < m; l++)
        C[i][j] += A[i][l] * B[l][j];

  return C;
}

VVT power(VVT& A, int k) {
  int n = A.size();
  VVT ret(n, VT(n)), B = A;
  for(int i = 0; i < n; i++) ret[i][i]=1;

  while(k) {
    if(k & 1) ret = multiply(ret, B);
    k >>= 1; B = multiply(B, B);
  }
  return ret;
}

int main()
{
  /* Expected Output:
     2.37^48 = 9.72569e+17

     376 264 285 220 265
     550 376 529 285 484
     484 265 376 264 285
     285 220 265 156 264
     529 285 484 265 376 */
  double n = 2.37;
  int k = 48;

  cout << n << "^" << k << " = " << power(n, k) << endl;

  double At[5][5] = {
    { 0, 0, 1, 0, 0 },
    { 1, 0, 0, 1, 0 },
    { 0, 0, 0, 0, 1 },
    { 1, 0, 0, 0, 0 },
    { 0, 1, 0, 0, 0 } };

  vector <vector <double> > A(5, vector <double>(5));
  for(int i = 0; i < 5; i++)
    for(int j = 0; j < 5; j++)
      A[i][j] = At[i][j];

  vector <vector <double> > Ap = power(A, k);

  cout << endl;
  for(int i = 0; i < 5; i++) {
    for(int j = 0; j < 5; j++)
      cout << Ap[i][j] << " ";
    cout << endl;
  }
}
```

3.2 Prime numbers

```
typedef unsigned long long ll;
typedef vector<ll> vll;
typedef vector<int> vi;

ll _sieve_size;
bitset<10000010> bs;
vll primes;

void sieve(ll upper) {
    _sieve_size = upper + 1;
    bs.set(); // set all to one
    bs[0] = bs[1] = 0;
    for(ll i = 2; i < _sieve_size; i++) if (bs[i]) {
            for(ll j = i*i; j < _sieve_size; j+= i) {
                bs[j] = 0;
            }
            primes.push_back((int) i);
        }
}

bool isPrime(ll n) {
    if (n <= _sieve_size) return bs[n];
    for(int i = 0; i < (int) primes.size(); i++) {
        if (n % primes[i] == 0) return false;
        if (primes[i] * primes[i] > n) return true;
    }
    return true;
}

bool isPrime_slow(ll n) {
    if(n < 2) return false;
    if(n == 2 || n == 3) return true;
    if(n % 2 == 0 || n % 3 == 0) return false;
    int limit = sqrt(n);
    for(int i = 5; i <= limit; i += 6) {
        if(n % i == 0 || n % (i+2) == 0)
            return false;
    }
    return true;
}


vi primeFactors(ll N) {
    vi factors;
    ll PF_index = 0; ll PF = primes[PF_index];
    while(PF*PF <= N) {
        while(N%PF == 0) {
            N /= PF; factors.push_back(PF);
        }
        PF = primes[++PF_index];
    }
    if(N != 1) factors.push_back(N);
    return factors;
}


// Primes less than 1000:
```

```
//      2     3     5     7    11    13    17    19    23    29    31
       37
//     41    43    47    53    59    61    67    71    73    79    83
       89
//     97   101   103   107   109   113   127   131   137   139   149
      151
//    157   163   167   173   179   181   191   193   197   199   211
      223
//    227   229   233   239   241   251   257   263   269   271   277
      281
//    283   293   307   311   313   317   331   337   347   349   353
      359
//    367   373   379   383   389   397   401   409   419   421   431
      433
//    439   443   449   457   461   463   467   479   487   491   499
      503
//    509   521   523   541   547   557   563   569   571   577   587
      593
//    599   601   607   613   617   619   631   641   643   647   653
      659
//    661   673   677   683   691   701   709   719   727   733   739
      743
//    751   757   761   769   773   787   797   809   811   821   823
      827
//    829   839   853   857   859   863   877   881   883   887   907
      911
//    919   929   937   941   947   953   967   971   977   983   991
      997

// Other primes: largest prime smaller than X is Y
//    10 is 7.
//    100 is 97.
//    1000 is 997.
//    10000 is 9973.
//    100000 is 99991.
//    1000000 is 999983.
//    10000000 is 9999991.
//    100000000 is 99999989.
//    1000000000 is 999999937.
//    10000000000 is 9999999967.
//    100000000000 is 99999999977.
//    1000000000000 is 999999999989.
//    10000000000000 is 9999999999971.
//    100000000000000 is 99999999999973.
//    1000000000000000 is 999999999999989.
//    10000000000000000 is 9999999999999937.
//    100000000000000000 is 99999999999999997.
//    1000000000000000000 is 999999999999999989.
```

3.3 Miller-Rabin Primality Test (C)

```
// Randomized Primality Test (Miller-Rabin):
//   Error rate: 2^(-TRIAL)
//   Almost constant time. srand is needed

#include <stdlib.h>
#define EPS 1e-7

typedef long long LL;

LL ModularMultiplication(LL a, LL b, LL m)
{
        LL ret=0, c=a;
        while(b)
        {
                if(b&1) ret=(ret+c)%m;
                b>>=1; c=(c+c)%m;
        }
        return ret;
}
LL ModularExponentiation(LL a, LL n, LL m)
{
        LL ret=1, c=a;
        while(n)
        {
                if(n&1) ret=ModularMultiplication(ret, c, m);
                n>>=1; c=ModularMultiplication(c, c, m);
        }
        return ret;
}
bool Witness(LL a, LL n)
{
        LL u=n-1;
  int t=0;
        while(!(u&1)){u>>=1; t++;}
        LL x0=ModularExponentiation(a, u, n), x1;
        for(int i=1;i<=t;i++)
        {
                x1=ModularMultiplication(x0, x0, n);
                if(x1==1 && x0!=1 && x0!=n-1) return true;
                x0=x1;
        }
        if(x0!=1) return true;
        return false;
}
LL Random(LL n)
{
  LL ret=rand(); ret*=32768;
        ret+=rand(); ret*=32768;
        ret+=rand(); ret*=32768;
        ret+=rand();
  return ret%n;
}
bool IsPrimeFast(LL n, int TRIAL)
{
  while(TRIAL--)
  {
    LL a=Random(n-2)+1;
    if(Witness(a, n)) return false;
  }
  return true;
}
```

3.4 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.
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;
}
```

3.5 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[][]
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;
  }
}
```

3.6 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[][]
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;
  }
}
```

3.7 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).
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;
}
```

3.8 Fast Fourier transform (C++)

```
// Convolution using the fast Fourier transform (FFT).
//
// INPUT:
//     a[1...n]
//     b[1...m]
//
// OUTPUT:
//     c[1...n+m-1] such that c[k] = sum_{i=0}^k a[i] b[k-i]
//
// Alternatively, you can use the DFT() routine directly, which will
// zero-pad your input to the next largest power of 2 and compute the
// DFT or inverse DFT.
typedef long double DOUBLE;
typedef complex<DOUBLE> COMPLEX;
typedef vector<DOUBLE> VD;
typedef vector<COMPLEX> VC;

struct FFT {
  VC A;
  int n, L;

  int ReverseBits(int k) {
    int ret = 0;
    for (int i = 0; i < L; i++) {
      ret = (ret << 1) | (k & 1);
      k >>= 1;
    }
    return ret;
  }

  void BitReverseCopy(VC 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];
  }

  VC DFT(VC a, bool inverse) {
    BitReverseCopy(a);
    for (int s = 1; s <= L; s++) {
      int m = 1 << s;
      COMPLEX wm = exp(COMPLEX(0, 2.0 * M_PI / m));
      if (inverse) wm = COMPLEX(1, 0) / wm;
      for (int k = 0; k < n; k += m) {
        COMPLEX w = 1;
        for (int j = 0; j < m/2; j++) {
          COMPLEX t = w * A[k + j + m/2];
          COMPLEX 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;
  }

  // c[k] = sum_{i=0}^k a[i] b[k-i]
  VD Convolution(VD a, VD b) {
    int L = 1;
    while ((1 << L) < a.size()) L++;
    while ((1 << L) < b.size()) L++;
    int n = 1 << (L+1);

    VC aa, bb;
    for (size_t i = 0; i < n; i++) aa.push_back(i < a.size() ? COMPLEX
         (a[i], 0) : 0);
    for (size_t i = 0; i < n; i++) bb.push_back(i < b.size() ? COMPLEX
         (b[i], 0) : 0);

    VC AA = DFT(aa, false);
    VC BB = DFT(bb, false);
    VC CC;
    for (size_t i = 0; i < AA.size(); i++) CC.push_back(AA[i] * BB[i])
         ;
    VC cc = DFT(CC, true);

    VD c;
    for (int i = 0; i < a.size() + b.size() - 1; i++) c.push_back(cc[i
         ].real());
    return c;
  }

};

int main() {
  double a[] = {1, 3, 4, 5, 7};
  double b[] = {2, 4, 6};

  FFT fft;
  VD c = fft.Convolution(VD(a, a + 5), VD(b, b + 3));

  // expected output: 2 10 26 44 58 58 42
  for (int i = 0; i < c.size(); i++) cerr << c[i] << " ";
  cerr << endl;

  return 0;
}
```

3.9 Pollard Rho Algorithm

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

#define abs_val(a) (((a)>=0)?(a):-(a))
typedef long long ll;

ll mulmod(ll a, ll b, ll c) { // returns (a * b) % c, and minimize
     overflow
  ll x = 0, y = a % c;
  while (b > 0) {
    if (b % 2 == 1) x = (x + y) % c;
    y = (y * 2) % c;
    b /= 2;
  }
  return x % c;
}

ll gcd(ll a, ll b) { return !b ? a : gcd(b, a % b); }       //
     standard gcd

ll pollard_rho(ll n) {
  int i = 0, k = 2;
  ll x = 3, y = 3;                  // random seed = 3, other values
       possible
  while (1) {
    i++;
    x = (mulmod(x, x, n) + n - 1) % n;                 // generating
         function
    ll d = gcd(abs_val(y - x), n);                         // the key
         insight
    if (d != 1 && d != n) return d;         // found one non-trivial
         factor
    if (i == k) y = x, k *= 2;
} }

int main() {
  ll n = 2063512844981574047LL;    // we assume that n is not a large
       prime
  ll ans = pollard_rho(n);          // break n into two non trivial
       factors
  if (ans > n / ans) ans = n / ans;          // make ans the smaller
       factor
  printf("%lld %lld\n", ans, n / ans);  // should be: 1112041493
       1855607779
} // return 0;
```

3.10 Big Number

```
// Depending on your application it's pretty unlikely that you'll have
      to type out the entirety of this struct. For example, in most
     cases you won't need division, and you can leave out most of the
      operators too.
// NB: Thse are fairly terrible implementations. Multiplication is
     about twice as slow as Python, and division is about 20 times
     slower. Use only as a last resort when for some reason you can't
      use Java's native bignums.
struct bignum {
        typedef unsigned int uint;

        vector<uint> digits;
        static const uint RADIX = 1000000000;

        bignum(): digits(1, 0) {}

        bignum(const bignum& x): digits(x.digits) {}

        bignum(unsigned long long x) {
                *this = x;
        }

        bignum(const char* x) {
                *this = x;
        }

        bignum(const string& s) {
                *this = s;
        }

        bignum& operator=(const bignum& y) {
```

```
        digits = y.digits; return *this;
}

bignum& operator=(unsigned long long x) {
        digits.assign(1, x%RADIX);
        if (x >= RADIX) {
                digits.push_back(x/RADIX);
        }
        return *this;
}

bignum& operator=(const char* s) {
        int slen=strlen(s),i,l;
        digits.resize((slen+8)/9);
        for (l=0; slen>0; l++,slen-=9) {
                digits[l]=0;
                for (i=slen>9?slen-9:0; i<slen; i++) {
                        digits[l]=10*digits[l]+s[i]-'0';
                }
        }
        while (digits.size() > 1 && !digits.back()) digits.
              pop_back();
        return *this;
}

bignum& operator=(const string& s) {
        return *this = s.c_str();
}

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

void sub(const bignum& x) {
        // if ((*this)<x) throw; //negative numbers not yet
              supported
        for (int d=0, borrow=0; d<digits.size(); d++) {
                digits[d]-=borrow;
                if (d<x.digits.size()) digits[d]-=x.digits[d];
                if (digits[d]>>31) { digits[d]+=RADIX; borrow
                      =1; } else borrow=0;
        }
        while (digits.size() > 1 && !digits.back()) digits.
              pop_back();
}

void mult(const bignum& x) {
        vector<uint> res(digits.size() + x.digits.size());
        unsigned long long y,z;
        for (int i=0; i<digits.size(); i++) {
                for (int j=0; j<x.digits.size(); j++) {
                        unsigned long long y=digits[i]; y*=x.
                              digits[j];
                        unsigned long long z=y/RADIX;
                        res[i+j+1]+=z; res[i+j]+=y-RADIX*z; //
                              mod is slow
                        if (res[i+j] >= RADIX) { res[i+j] -=
                              RADIX; res[i+j+1]++; }
                        for (int k = i+j+1; res[k] >= RADIX;
                              res[k] -= RADIX, res[++k]++);
                }
        }
        digits = res;
        while (digits.size() > 1 && !digits.back()) digits.
              pop_back();
}

// returns the remainder
bignum div(const bignum& x) {
        bignum dividend(*this);
        bignum divisor(x);
        fill(digits.begin(), digits.end(), 0);
        // shift divisor up
        int pwr = dividend.digits.size() - divisor.digits.size
              ();
        if (pwr > 0) {
                divisor.digits.insert(divisor.digits.begin(),
                      pwr, 0);
        }
        while (pwr >= 0) {
                if (dividend.digits.size() > divisor.digits.
                      size()) {
                  unsigned long long q = dividend.digits
                        .back();
                  q *= RADIX; q += dividend.digits[
                        dividend.digits.size()-2];
                  q /= 1+divisor.digits.back();
                  dividend -= divisor*q; digits[pwr] = q
                        ;
                  if (dividend >= divisor) { digits[pwr
                        ]++; dividend -= divisor; }
                  assert(dividend.digits.size() <=
                        divisor.digits.size()); continue
                        ;
                }
                while (dividend.digits.size() == divisor.
                      digits.size()) {
                  uint q = dividend.digits.back() / (1+
                        divisor.digits.back());
                  if (q == 0) break;
                  digits[pwr] += q; dividend -= divisor*
                        q;
                }
                if (dividend >= divisor) { dividend -= divisor
                      ; digits[pwr]++; }
                pwr--; divisor.digits.erase(divisor.digits.
                      begin());
        }
        while (digits.size() > 1 && !digits.back()) digits.
              pop_back();
        return dividend;
}

string to_string() const {
        ostringstream oss;
        oss << digits.back();
        for (int i = digits.size() - 2; i >= 0; i--) {
                oss << setfill('0') << setw(9) << digits[i];
        }
        return oss.str();
}

bignum operator+(const bignum& y) const {
        bignum res(*this); res.add(y); return res;
}

bignum operator-(const bignum& y) const {
        bignum res(*this); res.sub(y); return res;
}

bignum operator*(const bignum& y) const {
        bignum res(*this); res.mult(y); return res;
}

bignum operator/(const bignum& y) const {
        bignum res(*this); res.div(y); return res;
}

bignum operator%(const bignum& y) const {
        bignum res(*this); return res.div(y);
}

bignum& operator+=(const bignum& y) {
        add(y); return *this;
}

bignum& operator-=(const bignum& y) {
        sub(y); return *this;
}

bignum& operator*=(const bignum& y) {
        mult(y); return *this;
}

bignum& operator/=(const bignum& y) {
        div(y); return *this;
}

bignum& operator%=(const bignum& y) {
        *this = div(y);
}

bool operator==(const bignum& y) const {
        return digits == y.digits;
}

        bool operator<(const bignum& y) const {
                if (digits.size() < y.digits.size()) return true;
                if (digits.size() > y.digits.size()) return false;
                for (int i = digits.size()-1; i >= 0; i--) {
                        if (digits[i] < y.digits[i]) {
                                return true;
                        } else if (digits[i] > y.digits[i]) {
                                return false;
                        }
                }
                return false;
        }

        bool operator>(const bignum& y) const {
                return y<*this;
        }

        bool operator<=(const bignum& y) const {
                return !(y<*this);
        }

        bool operator>=(const bignum& y) const {
                return !(*this<y);
        }
};
```

4 Graph algorithms

4.1 Dijkstra's algorithm

```
int V, E, s, u, v, w;
vector<vii> AdjList;

int main() {
    cin>>V>>E>>s;
    AdjList.assign(V, vii());
    int u,v,w;
    for(int i = 0; i < E; i++) {
        cin>>u>>v>>w;
        AdjList[u].push_back(ii(v, w));
    }
    // Dijkstra routine
    vi dist(V, INF); dist[s] = 0;
    // distance, node
    priority_queue<ii, vector<ii>, greater<ii> > pq;
    pq.push(ii(0,s));

    while(!pq.empty()) {
        ii front = pq.top(); pq.pop();
        int d = front.first, u = front.second;
        if (d > dist[u]) continue; // handle duplicates
        for(int j = 0; j < (int) AdjList[u].size(); j++) {
            ii v = AdjList[u][j];
            if(dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second;
                pq.push(ii(dist[v.first], v.first));
            }
        }
    }

    // SPFA (Faster Bellman Ford)
    queue<int> q; q.push(S);
    vi in_queue(n, 0); in_queue[S] = 1;

    while(!q.empty()) {
        int u = q.front(); q.pop(); in_queue[u] = 0;
        for(auto v : AdjList[u]) {
            if(dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second;
                if (!in_queue[v.first]) {
                    q.push(v.first);  // add to queue only if it's not
                          in queue
                    in_queue[v.first] = 1;
                }
            }
        }
     }

}
```

4.2 Strongly connected components

```
bool adjList[55][55];
vi dfs_num, dfs_low, S, visited;
int dfs_counter = 0, numSCC = 0;
int m, n; // nLocations, nEdges

void tarjanSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfs_counter++;
    S.push_back(u);
    visited[u] = 1;
    for(int v = 0; v < m; v++) {
        if(adjList[u][v]) {
            if(dfs_num[v] == -1) tarjanSCC(v);
            if(visited[v] == 1)
                dfs_low[u] = min(dfs_low[v], dfs_low[u]);
        }
    }

    if(dfs_low[u] == dfs_num[u]) { //root
        printf("SCC %d\n", ++numSCC);
        while(1) {
            int v = S.back(); S.pop_back(); visited[v] = 0;
            printf("%d ", v);
            if(u == v) break;
        }
        printf("\n");
    }
}

int main() {
    m=8,n=9; // edges and vertices

    adjList[0][1] = true; adjList[1][3] = true; adjList[3][4] = true;
    adjList[4][5] = true; adjList[5][7] = true; adjList[7][6] = true;
    adjList[6][4] = true; adjList[3][2] = true; adjList[2][1] = true;

    dfs_num = vi(m, -1); dfs_low = vi(m, 0); visited = vi(m, 0);
    dfs_counter = numSCC = 0;

    for(int i = 0; i < m; i++) {
        if(dfs_num[i] == -1) {
            tarjanSCC(i);
        }
    }
    // 1: 6 7 4 5
    // 2: 2 3 1
    // 3: 0
}
```

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 Kruskal's algorithm

```
/*
Uses Kruskal's Algorithm to calculate the weight of the minimum
      spanning
forest (union of minimum spanning trees of each connected component)
      of
a possibly disjoint graph, given in the form of a matrix of edge
      weights
(-1 if no edge exists). Returns the weight of the minimum spanning
forest (also calculates the actual edges - stored in T). Note: uses a
disjoint-set data structure with amortized (effectively) constant time
       per
union/find. Runs in O(E*log(E)) time.
*/
typedef int T;

struct edge
{
  int u, v;
  T d;
};

struct edgeCmp
{
  int operator()(const edge& a, const edge& b) { return a.d > b.d; }
};

int find(vector <int>& C, int x) { return (C[x] == x) ? x : C[x] =
      find(C, C[x]); }

T Kruskal(vector <vector <T> >& w)
{
  int n = w.size();
  T weight = 0;

  vector <int> C(n), R(n);
  for(int i=0; i<n; i++) { C[i] = i; R[i] = 0; }

  vector <edge> T;
  priority_queue <edge, vector <edge>, edgeCmp> E;

  for(int i=0; i<n; i++)
    for(int j=i+1; j<n; j++)
      if(w[i][j] >= 0)
      {
        edge e;
        e.u = i; e.v = j; e.d = w[i][j];
        E.push(e);
      }

  while(T.size() < n-1 && !E.empty())
  {
    edge cur = E.top(); E.pop();

    int uc = find(C, cur.u), vc = find(C, cur.v);
    if(uc != vc)
    {
      T.push_back(cur); weight += cur.d;

      if(R[uc] > R[vc]) C[vc] = uc;
      else if(R[vc] > R[uc]) C[uc] = vc;
      else { C[vc] = uc; R[uc]++; }
    }
  }

  return weight;
}

int main()
{
  int wa[6][6] = {
    { 0, -1, 2, -1, 7, -1 },
    { -1, 0, -1, 2, -1, -1 },
    { 2, -1, 0, -1, 8, 6 },
    { -1, 2, -1, 0, -1, -1 },
    { 7, -1, 8, -1, 0, 4 },
    { -1, -1, 6, -1, 4, 0 } };

  vector <vector <int> > w(6, vector <int>(6));

  for(int i=0; i<6; i++)
    for(int j=0; j<6; j++)
      w[i][j] = wa[i][j];

  cout << Kruskal(w) << endl;
  cin >> wa[0][0];
}
```

4.5 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
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;
}
```

4.6 Lowest common ancestor

```
const int max_nodes, log_max_nodes;
int num_nodes, log_num_nodes, root;

vector<int> children[max_nodes];        // children[i] contains the
      children of node i
int A[max_nodes][log_max_nodes+1];      // A[i][j] is the 2^j-th
      ancestor of node i, or -1 if that ancestor does not exist
int L[max_nodes];                       // L[i] is the distance
      between node i and the root

// floor of the binary logarithm of n
int lb(unsigned int n)
{
    if(n==0)
        return -1;
    int p = 0;
    if (n >= 1<<16) { n >>= 16; p += 16; }
    if (n >= 1<< 8) { n >>=  8; p +=  8; }
    if (n >= 1<< 4) { n >>=  4; p +=  4; }
    if (n >= 1<< 2) { n >>=  2; p +=  2; }
    if (n >= 1<< 1) {           p +=  1; }
    return p;
}

void DFS(int i, int l)
{
    L[i] = l;
    for(int j = 0; j < children[i].size(); j++)
        DFS(children[i][j], l+1);
}

int LCA(int p, int q)
{
    // ensure node p is at least as deep as node q
    if(L[p] < L[q])
        swap(p, q);

    // "binary search" for the ancestor of node p situated on the same
         level as q
    for(int i = log_num_nodes; i >= 0; i--)
        if(L[p] - (1<<i) >= L[q])
            p = A[p][i];

    if(p == q)
        return p;

    // "binary search" for the LCA
    for(int i = log_num_nodes; i >= 0; i--)
        if(A[p][i] != -1 && A[p][i] != A[q][i])
        {
            p = A[p][i];
            q = A[q][i];
        }

    return A[p][0];
}

int main(int argc,char* argv[])
{
    // read num_nodes, the total number of nodes
    log_num_nodes=lb(num_nodes);

    for(int i = 0; i < num_nodes; i++)
    {
        int p;
        // read p, the parent of node i or -1 if node i is the root

        A[i][0] = p;
        if(p != -1)
            children[p].push_back(i);
        else
            root = i;
    }

    // precompute A using dynamic programming
    for(int j = 1; j <= log_num_nodes; j++)
        for(int i = 0; i < num_nodes; i++)
            if(A[i][j-1] != -1)
                A[i][j] = A[A[i][j-1]][j-1];
            else
                A[i][j] = -1;

    // precompute L
    DFS(root, 0);


    return 0;
}
```

4.7 Bridge and Articulation Points

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

#define FOR(x,n) for(int x = 0; x < n; ++x)

typedef unsigned long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
typedef vector<string> vs;
typedef vector<vi> vvi;

#define UNVISITED 0

vvi AdjList;
int dfsCounter, rootChildren, dfsRoot;
vi dfs_num, dfs_low, dfs_parent, art_vertex;

void artPointAndBridge(int u) {
        dfs_low[u] = dfs_num[u] = dfsCounter++;
        for(auto &v : AdjList[u]) {
                if(dfs_num[v] == UNVISITED) {
                        dfs_parent[v] = u;
                        if(u == dfsRoot) rootChildren++;

                        artPointAndBridge(v);

                        if(dfs_low[v] >= dfs_num[v]) // art point
                                art_vertex[v] = true;
                        if(dfs_low[v] > dfs_num[u]) {
                                // (u,v) is bridge
                                printf("(%d,%d) is a bridge\n", u, v);
                        }
                        //update dfs_low
                        dfs_low[u] = min(dfs_low[u], dfs_low[v]);
                } else if(v != dfs_parent[u]) {
                        // back edge and not direct cycle
                        dfs_low[u] = min(dfs_low[u], dfs_low[v]);
                }
        }
}

int main() {

        int v = 6;

        AdjList.assign(v, vi());
        AdjList[0].push_back(1); AdjList[1].push_back(0);
        AdjList[1].push_back(2); AdjList[2].push_back(1);
        AdjList[1].push_back(3); AdjList[3].push_back(1);
        AdjList[1].push_back(4); AdjList[4].push_back(1);
        AdjList[4].push_back(5); AdjList[5].push_back(4);
        AdjList[1].push_back(5); AdjList[5].push_back(1);

        dfsCounter = 0, dfs_num.assign(v, UNVISITED), dfs_low.assign(v
             , 0);
        dfs_parent.assign(v, 0), art_vertex.assign(v, 0);

        printf("Bridges\n"); // (0,1), (1,2), (1,3)
        for(int i = 0; i < v; i++) {
                if(dfs_num[i] == UNVISITED) {
                        dfsRoot = 1, rootChildren = 0,
                               artPointAndBridge(i);
                        art_vertex[dfsRoot] = (rootChildren > 1); //
                               special case
                }
        }

        printf("Articuation points\n"); // 0,1,2,3
        for(int i = 0; i < v; i++) {
                if(art_vertex[i])
                        printf("Vertex %d\n", i);
        }
}
```

5 Data structures

5.1 Binary Indexed Tree

```
// BIT with range updates, inspired by Petr Mitrichev
struct BIT {
    int n;
    vector<int> slope;
    vector<int> intercept;
    // BIT can be thought of as having entries f[1], ..., f[n]
    // which are 0-initialized
    BIT(int n): n(n), slope(n+1), intercept(n+1) {}
    // returns f[1] + ... + f[idx-1]
    // precondition idx <= n+1
    int query(int idx) {
        int m = 0, b = 0;
        for (int i = idx-1; i > 0; i -= i&-i) {
            m += slope[i];
            b += intercept[i];
        }
        return m*idx + b;
    }
    // adds amt to f[i] for i in [idx1, idx2)
    // precondition 1 <= idx1 <= idx2 <= n+1 (you can't update element
         0)
    void update(int idx1, int idx2, int amt) {
        for (int i = idx1; i <= n; i += i&-i) {
            slope[i] += amt;
            intercept[i] -= idx1*amt;
        }
        for (int i = idx2; i <= n; i += i&-i) {
            slope[i] -= amt;
            intercept[i] += idx2*amt;
        }
    }
};

// BIT with range updates, inspired by Petr Mitrichev
class FenwickTree {
private: vi ft1, ft2;
    int query(vi &ft, int b) {
        int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
        return sum; }
    void adjust(vi &ft, int k, int v) {
        for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
public:
    FenwickTree() {}
    FenwickTree(int n) { ft1.assign(n + 1, 0); ft2.assign(n+1, 0);}
    int query(int a) { return a * query(ft1, a) - query(ft2, a); }
    int query(int a, int b) { return query(b) - (a == 1 ? 0 : query(a
         -1)); }
    void adjust(int a, int b, int value){
        adjust(ft1, a, value);
        adjust(ft1, b+1, -value);
        adjust(ft2, a, value * (a-1));
        adjust(ft2, b+1, -1 * value * b);
    }
    int get(int n) {
        return query(n) - query(n-1);}
};
```

5.2 2D Binary Indexed Tree

```
// WARNING NOT FIELD TESTED YET
class FenwickTree {
private:
  vi ft;
```

```
public:
  FenwickTree() {}
  // initialization: n + 1 zeroes, ignore index 0
  FenwickTree(int n) { ft.assign(n + 1, 0); }

  int rsq(int b) {                                    // returns RSQ
        (1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum; }

  int rsq(int a, int b) {                             // returns RSQ(
        a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }

  // adjusts value of the k-th element by v (v can be +ve/inc or -ve/
        dec)
  void adjust(int k, int v) {                    // note: n = ft.size
        () - 1
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};

class FenwickTree2D {
private:
  vector<FenwickTree> ft2d;

public:
  FenwickTree2D() {}
  FenwickTree2D(int n) { ft2d.assign(n+1, FenwickTree(n)); }

  int rsq(int r, int c) {
    int sum = 0;
    for(; r; r -= LSOne(r)) sum += ft2d[r].rsq(c);
    return sum;
  }

  // top left, bottom right
  int rsq(int r1, int c1, int r2, int c2) {
    return rsq(r2, c2) - rsq(r2, c1-1) - rsq(r1-1, c2) + rsq(r1-1, c2
          -1);
  }

  void adjust(int r, int c, int v) {
    for (; r < (int)ft2d.size(); r += LSOne(r)) ft2d[r].adjust(c, v);
  }
};


int main() {
  FenwickTree2D ft2d(4);
  ft2d.adjust(1, 1, 1);
  ft2d.adjust(2, 2, 1);
  ft2d.adjust(3, 3, 1);
  ft2d.adjust(4, 4, 1);
  printf("%d\n", ft2d.rsq(1,1));      // 1
  printf("%d\n", ft2d.rsq(2,2));      // 2
  printf("%d\n", ft2d.rsq(3,3));      // 3
  printf("%d\n", ft2d.rsq(2,2,3,3));  // 2
  return 0;
}
```

5.3 Union-find

```
struct UnionFind {
    vector<int> C;
    UnionFind(int n) : C(n) { for (int i = 0; i < n; i++) C[i] = i; }
    int find(int x) { return (C[x] == x) ? x : C[x] = find(C[x]); }
    void merge(int x, int y) { C[find(x)] = find(y); }
};
int main()
{
    int n = 5;
    UnionFind uf(n);
    uf.merge(0, 2);
    uf.merge(1, 0);
    uf.merge(3, 4);
    for (int i = 0; i < n; i++) cout << i << " " << uf.find(i) << endl
          ;
    return 0;
}
```

5.4 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
// -----------------------------------------------------------------

#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;
    }

    return 0;
}

//
   ---------------------------------------------------------------------------
```

5.5 Splay tree

```
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);
  }
}
```

5.6 Splay Link Cut Trees

```
const int MAXN = 110000;

typedef struct _node{
  _node *l, *r, *p, *pp;
  int size; bool rev;
  _node();
  explicit _node(nullptr_t){
    l = r = p = pp = this;
    size = rev = 0;
  }
  void push(){
    if (rev){
      l->rev ^= 1; r->rev ^= 1;
      rev = 0; swap(l,r);
    }
  }
  void update();
}* node;
node None = new _node(nullptr);
node v2n[MAXN];
_node::_node(){
  l = r = p = pp = None;
  size = 1; rev = false;
}
void _node::update(){
  size = (this != None) + l->size + r->size;
  l->p = r->p = this;
}
void rotate(node v){
  assert(v != None && v->p != None);
  assert(!v->rev); assert(!v->p->rev);
  node u = v->p;
  if (v == u->l)
    u->l = v->r, v->r = u;
  else
    u->r = v->l, v->l = u;
  swap(u->p,v->p); swap(v->pp,u->pp);
  if (v->p != None){
    assert(v->p->l == u || v->p->r == u);
    if (v->p->r == u) v->p->r = v;
    else v->p->l = v;
  }
  u->update(); v->update();
}
void bigRotate(node v){
  assert(v->p != None);
  v->p->p->push();
  v->p->push();
  v->push();
  if (v->p->p != None){
    if ((v->p->l == v) ^ (v->p->p->r == v->p))
      rotate(v->p);
    else
      rotate(v);
  }
  rotate(v);
}
inline void Splay(node v){
  while (v->p != None) bigRotate(v);
}
inline void splitAfter(node v){
  v->push();
  Splay(v);
  v->r->p = None;
```

```
  v->r->pp = v;
  v->r = None;
  v->update();
}
void expose(int x){
  node v = v2n[x];
  splitAfter(v);
  while (v->pp != None){
    assert(v->p == None);
    splitAfter(v->pp);
    assert(v->pp->r == None);
    assert(v->pp->p == None);
    assert(!v->pp->rev);
    v->pp->r = v;
    v->pp->update();
    v = v->pp;
    v->r->pp = None;
  }
  assert(v->p == None);
  Splay(v2n[x]);
}
inline void makeRoot(int x){
  expose(x);
  assert(v2n[x]->p == None);
  assert(v2n[x]->pp == None);
  assert(v2n[x]->r == None);
  v2n[x]->rev ^= 1;
}
inline void link(int x,int y){
  makeRoot(x); v2n[x]->pp = v2n[y];
}
inline void cut(int x,int y){
  expose(x);
  Splay(v2n[y]);
  if (v2n[y]->pp != v2n[x]){
    swap(x,y);
    expose(x);
    Splay(v2n[y]);
    assert(v2n[y]->pp == v2n[x]);
  }
  v2n[y]->pp = None;
}
inline int get(int x,int y){
  if (x == y) return 0;
  makeRoot(x);
  expose(y); expose(x);
  Splay(v2n[y]);
  if (v2n[y]->pp != v2n[x]) return -1;
  return v2n[y]->size;
}
```

5.7 Sparse Table

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

#define MAX_N 1000                          // adjust this value as
     needed
#define LOG_TWO_N 10              // 2^10 > 1000, adjust this value as
     needed

class RMQ {                                        // Range Minimum
     Query
private:
  int _A[MAX_N], SpT[MAX_N][LOG_TWO_N];
public:
  RMQ(int n, int A[]) {    // constructor as well as pre-processing
       routine
    for (int i = 0; i < n; i++) {
      _A[i] = A[i];
      SpT[i][0] = i; // RMQ of sub array starting at index i + length
           2^0=1
    }
    // the two nested loops below have overall time complexity = O(n
         log n)
    for (int j = 1; (1<<j) <= n; j++) // for each j s.t. 2^j <= n, O(
         log n)
      for (int i = 0; i + (1<<j) - 1 < n; i++)    // for each valid i,
           O(n)
        if (_A[SpT[i][j-1]] < _A[SpT[i+(1<<(j-1))][j-1]])
             // RMQ
          SpT[i][j] = SpT[i][j-1];    // start at index i of length
               2^(j-1)
        else                   // start at index i+2^(j-1) of length
             2^(j-1)
          SpT[i][j] = SpT[i+(1<<(j-1))][j-1];
  }

  int query(int i, int j) {
    int k = (int)floor(log((double)j-i+1) / log(2.0));    // 2^k <= (j
         -i+1)
    if (_A[SpT[i][k]] <= _A[SpT[j-(1<<k)+1][k]]) return SpT[i][k];
    else                                          return SpT[j-(1<<k)
         +1][k];
} };

int main() {
  // same example as in chapter 2: segment tree
  int n = 7, A[] = {18, 17, 13, 19, 15, 11, 20};
  RMQ rmq(n, A);
  for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
      printf("RMQ(%d, %d) = %d\n", i, j, rmq.query(i, j));

  return 0;
}
```

5.8 Lazy Segment Tree

```
#include <iostream>
#include <algorithm>
using namespace std;
const int n=10; // number of elements in the tree should be at most
     (1<<n)
int arr[1<<(n+1)]; // store values
int low[1<<(n+1)]; // left of range
int high[1<<(n+1)]; // right of range
int lazyadd[1<<(n+1)];

// should be commutative and associative. most commonly used: a+b, max
     (a,b), min(a,b)
int acc(int a, int b) {
    return max(a,b);
}
int acc2(int a, int b) {
    // a "acts" on b, the function needs to be distributive over acc,
         commutative + associative
    // common uses: a+max(b,c) = max(a+b, a+c), a*max(b,c)=max(a*b,a*c
         ) (only if a>0)
    // a*(b+c) = (a*b)+(a*c)
    return a+b;
}
void init() {
    for(int i=0; i<(1<<n); i++) {
        low[i+(1<<n)] = i;
        high[i+(1<<n)] = i;
        arr[i+(1<<n)] = 0; // initial value
    }
    for(int i=(1<<n)-1; i>=0; i--) {
        low[i] = min(low[2*i], low[2*i+1]);
        high[i] = max(high[2*i], high[2*i+1]);
        arr[i] = acc(arr[2*i], arr[2*i+1]);
    }
    for(int i=0; i<(1<<(n+1)); i++) {
        lazyadd[i] = 0; // identity of the acc2 function
    }
}
int value(int node) { // gives the true value of the node
    arr[node] = acc2(lazyadd[node], arr[node]);
    if(node<(1<<n)) { // not the leaf, propagate downwards
        lazyadd[2*node] = acc2(lazyadd[node], lazyadd[2*node]); //
             stack values on children
        lazyadd[2*node+1] = acc2(lazyadd[node], lazyadd[2*node+1]); //
              stack values on children
    }
    lazyadd[node] = 0; // reset to identity function
    return arr[node];
}
void update(int node, int left, int right, int change) {
    if(right>=high[node] && left<=low[node]) { // case 1: updated
         range covers node
        lazyadd[node] = acc2(lazyadd[node],change); // stack the
             change
    } else if(right<low[node] || left>high[node]) { // case 2: empty
         intersection
        return;
    } else { // case 3: need to propagate
        update(2*node, left, right, change);
        update(2*node+1, left, right, change);
        arr[node] = acc(value(node*2), value(node*2+1));
    }
}
void update(int left, int right, int change) {
    update(1, left, right, change);
}
int query(int node, int left, int right) {
    value(node); // important to call this!
    if(right>=high[node] && left<=low[node]) {
        return arr[node];
    } else if(right<low[node] || left>high[node]) {
        return -(1<<30); // identity operator of acc
    } else {
        return acc(query(node*2, left, right), query(node*2+1, left,
             right));
    }
}
int query(int left, int right) {
    return query(1, left,right);
}
int main()
{
    init();
    cout << query(1,5); //0
    update(2,5,3); // 0 3 3 3 3 -> max=3
    cout << query(1,5); // 3
    update(2,3,2); // 0 5 5 3 3 -> max=5
    cout << query(1,5); // 5
    update(2,4,-4); // 0 1 1 -1 3 -> max=5
    cout << query(1,5); // 5
    update(5,5,-1); // 0 1 1 -1 2 -> max=2
    cout << query(1,5); // 2
    return 0;
}
```

6 String

6.1 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.
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


// Yan Hao's Implementation
#include <bits/stdc++.h>
using namespace std;

#define all(o) (o).begin(), (o).end()
#define allr(o) (o).rbegin(), (o).rend()
const int INF = 2147483647;
typedef long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
typedef vector<vi> vvi;
typedef vector<vii> vvii;
template <class T> int size(T &x) { return x.size(); }

// assert or gtfo

struct suffix_array {
    struct entry {
        pair<int, int> nr;
        int p;

        bool operator <(const entry &other) const {
            return nr < other.nr;
        }
    };

    string s;
    int n;
    vector<vector<int> > P;
    vector<entry> L;
    vector<int> idx;

    suffix_array(string _s) : s(_s), n(s.size()) {
        L = vector<entry>(n);
        P.push_back(vector<int>(n));
        idx = vector<int>(n);

        for (int i = 0; i < n; i++) {
            P[0][i] = s[i];
        }

        for (int stp = 1, cnt = 1; (cnt >> 1) < n; stp++, cnt <<= 1) {
            P.push_back(vector<int>(n));
            for (int i = 0; i < n; i++) {
                L[i].p = i;
                L[i].nr = make_pair(P[stp - 1][i],
                        i + cnt < n ? P[stp - 1][i + cnt] : -1);
            }

            sort(L.begin(), L.end());
            for (int i = 0; i < n; i++) {
                if (i > 0 && L[i].nr == L[i - 1].nr) {
                    P[stp][L[i].p] = P[stp][L[i - 1].p];
                } else {
                    P[stp][L[i].p] = i;
                }
            }
        }

        for (int i = 0; i < n; i++) {
            idx[P[P.size() - 1][i]] = i;
        }
    }

    int lcp(int x, int y) {
        int res = 0;
        if (x == y) return n - x;
        for (int k = P.size() - 1; k >= 0 && x < n && y < n; k--) {
            if (P[k][x] == P[k][y]) {
                x += 1 << k;
                y += 1 << k;
                res += 1 << k;
            }
        }
        return res;
    }

    int longestRepeatedSubsequence() {
        int ans=0;
        for(int i=1; i<n; i++) {
            ans = max(ans, lcp(idx[i-1], idx[i]));
        }
        return ans;
    }
};

int main() {
    while(true) {
        string x,y;
        getline(cin, x);
        getline(cin, y);
        y.push_back('$');
        if(!cin.good()) break;
        suffix_array sa(y);
        //do binary search
        int low = 0;
        int high = y.length();
        while(high-low>1) {
            int mid = (high+low)/2;
            int z = sa.idx[mid];
            for(int i=0; i<x.length() && z+i<y.length(); i++) {
                if(x[i]<y[z+i]) {
                    high=mid; break;
                } else if(x[i]>y[z+i]) {
                    low=mid; break;
                }
            }
            if(high!=mid && low!=mid) {
                high=mid;
            }
        }
        int low2 = 0;
        int high2 = y.length();
        while(high2-low2>1) {
            int mid = (high2+low2)/2;
            int z = sa.idx[mid];
            for(int i=0; i<x.length() && z+i<y.length(); i++) {
                if(x[i]<y[z+i]) {
                    high2=mid; break;
                } else if(x[i]>y[z+i]) {
                    low2=mid; break;
                }
            }
            if(high2!=mid && low2!=mid) {
                low2=mid;
            }
        }
        vector<int> ans;
        for(int i=low+1; i<=low2; i++) {
            ans.push_back(sa.idx[i]);
        }
        sort(ans.begin(), ans.end());
        for(int v: ans) {
            cout << v << " ";
        }
        cout << "\n";
    }
    return 0;
}
```

6.2 Knuth-Morris-Pratt

```
/*
Finds all occurrences of the pattern string p within the
text string t. Running time is O(n + m), where n and m
are the lengths of p and t, respecitvely.
*/
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
  return 0;
}
```

7 Miscellaneous

7.1 Binary Search

```
// n is size of array, c is value looking for
// sematically equiv to std::lower_bound and std::upper_bound
int lower_bound(int A[], int n, int c) {
        int l = 0, r = n;
        while(l < r) {
                int m = (r-l)/2+l;
                if(A[m] < c) l = m+1; else r=m;
        }
        return l;
}

int upper_bound(int A[], int n, int c) {
        int l = 0, r = n;
        while(l < r) {
                int m = (r-l)/2+l;
                if(A[m] <= c) l = m+1; else r=m;
        }
        return l;
}
```

7.2 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;
}
```

7.3 Median Max/Min Heap

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

int main() {
        priority_queue<int> maxPQ;
        priority_queue<int, vector<int>, greater<int> > minPQ;
        string s;
        while(cin >> s) {
                if (s == "#") {
                        int m = minPQ.top(); minPQ.pop();
                        if (minPQ.size() != maxPQ.size()) {
                                minPQ.push(maxPQ.top());
                                maxPQ.pop();
                        }
                        cout << m << endl;
                } else {
                        int c = stoi(s);
                        if(!minPQ.empty() && c > minPQ.top()) {
                                minPQ.push(c);
                                if(minPQ.size() > maxPQ.size() + 1) {
                                        int d = minPQ.top(); minPQ.pop
                                            ();
                                        maxPQ.push(d);
                                }
                        } else {
                                maxPQ.push(c);
                                if (maxPQ.size() > minPQ.size()) {
                                        minPQ.push(maxPQ.top());
                                        maxPQ.pop();
                                }
                        }
                }
        }
}
```

7.4 Dates

```
// Routines for performing computations on dates.  In these routines,
// months are expressed as integers from 1 to 12, days are expressed
// as integers from 1 to 31, and years are expressed as 4-digit
// integers.
string dayOfWeek[] = {"Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"
    };

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

// converts integer (Julian day number) to Gregorian date: month/day/
    year
void intToDate (int jd, int &m, int &d, int &y){
  int x, n, i, j;

  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

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

int main (int argc, char **argv){
  int jd = dateToInt (3, 24, 2004);
  int m, d, y;
  intToDate (jd, m, d, y);
  string day = intToDay (jd);

  // expected output:
  //    2453089
  //    3/24/2004
  //    Wed
  cout << jd << endl
    << m << "/" << d << "/" << y << endl
    << day << endl;
}
```

7.5 Latitude/longitude

```
/*
Converts from rectangular coordinates to latitude/longitude and vice
versa. Uses degrees (not radians).
*/
struct ll
{
  double r, lat, lon;
};

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

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

  return Q;
}

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

  return P;
}

int main()
{
  rect A;
  ll B;

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

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

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

7.6 Random STL stuff

```
// Example for using stringstreams and next_permutation
int main(void){
  vector<int> v;

  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);

  // Expected output: 1 2 3 4
  //                  1 2 4 3
  //                  ...
  //                  4 3 2 1
  do {
    ostringstream oss;
    oss << v[0] << " " << v[1] << " " << v[2] << " " << v[3];

    // for input from a string s,
    //   istringstream iss(s);
    //   iss >> variable;

    cout << oss.str() << endl;
  } while (next_permutation (v.begin(), v.end()));

  v.clear();
```

```
  v.push_back(1);
  v.push_back(2);
  v.push_back(1);
  v.push_back(3);

  // To use unique, first sort numbers.  Then call
  // unique to place all the unique elements at the beginning
  // of the vector, and then use erase to remove the duplicate
  // elements.

  sort(v.begin(), v.end());
  v.erase(unique(v.begin(), v.end()), v.end());

  // Expected output: 1 2 3
  for (size_t i = 0; i < v.size(); i++)
    cout << v[i] << " ";
  cout << endl;
}
```

7.7 Longest common subsequence

```
/*
Calculates the length of the longest common subsequence of two vectors
    .
Backtracks to find a single subsequence or all subsequences. Runs in
O(m*n) time except for finding all longest common subsequences, which
may be slow depending on how many there are.
*/
typedef int T;
typedef vector<T> VT;
typedef vector<VT> VVT;

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

void backtrack(VVI& dp, VT& res, VT& A, VT& B, int i, int j)
{
  if(!i || !j) return;
  if(A[i-1] == B[j-1]) { res.push_back(A[i-1]); backtrack(dp, res, A,
      B, i-1, j-1); }
  else
  {
    if(dp[i][j-1] >= dp[i-1][j]) backtrack(dp, res, A, B, i, j-1);
    else backtrack(dp, res, A, B, i-1, j);
  }
}

void backtrackall(VVI& dp, set<VT>& res, VT& A, VT& B, int i, int j)
{
  if(!i || !j) { res.insert(VI()); return; }
  if(A[i-1] == B[j-1])
  {
    set<VT> tempres;
    backtrackall(dp, tempres, A, B, i-1, j-1);
    for(set<VT>::iterator it=tempres.begin(); it!=tempres.end(); it++)
    {
      VT temp = *it;
      temp.push_back(A[i-1]);
      res.insert(temp);
    }
  }
  else
  {
    if(dp[i][j-1] >= dp[i-1][j]) backtrackall(dp, res, A, B, i, j-1);
    if(dp[i][j-1] <= dp[i-1][j]) backtrackall(dp, res, A, B, i-1, j);
  }
}

VT LCS(VT& A, VT& B)
{
  VVI dp;
  int n = A.size(), m = B.size();
  dp.resize(n+1);
  for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);

  for(int i=1; i<=n; i++)
    for(int j=1; j<=m; j++)
    {
      if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }

  VT res;
  backtrack(dp, res, A, B, n, m);
  reverse(res.begin(), res.end());
  return res;
}

set<VT> LCSall(VT& A, VT& B)
{
  VVI dp;
  int n = A.size(), m = B.size();
  dp.resize(n+1);
  for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);
  for(int i=1; i<=n; i++)
    for(int j=1; j<=m; j++)
    {
      if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }
  set<VT> res;
  backtrackall(dp, res, A, B, n, m);
  return res;
}

int main()
{
  int a[] = { 0, 5, 5, 2, 1, 4, 2, 3 }, b[] = { 5, 2, 4, 3, 2, 1, 2,
      1, 3 };
  VI A = VI(a, a+8), B = VI(b, b+9);
  VI C = LCS(A, B);

  for(int i=0; i<C.size(); i++) cout << C[i] << " ";
  cout << endl << endl;

  set <VI> D = LCSall(A, B);
  for(set<VI>::iterator it = D.begin(); it != D.end(); it++)
  {
    for(int i=0; i<(*it).size(); i++) cout << (*it)[i] << " ";
    cout << endl;
  }
}
```

8 Language Stuff

8.1 Nifty Tricks

```
// remove duplicated from vector
#define UNIQUE(x) x.erase(unique(x.begin(), x.end()), x.end())
// filters.erase(unique(filters.begin(), filters.end()), filters.end()
    );

// convert string to int
int myint = stoi("123");

// memset
int res[MAX_V][MAX_V];
memset(res, 0, sizeof res);
fill (myvector.begin(),myvector.begin()+4,5);
int myint1 = stoi(str1); // convert string to int

// Convert int to binary string
cout << bitset<32>(val).to_string() << endl;

// Generate all permutations
sort(nodes.begin(), nodes.end());
do {
  int sum = 0;
  for(int i = 1; i < nodes.size(); i++)
    sum += __builtin_popcount(nodes[i] & nodes[i-1]);
  best = min(best, sum);
 } while(next_permutation(nodes.begin(), nodes.end()));

// Generate all set of n elements
unsigned next_set_n(unsigned x) {
  unsigned smallest, ripple, new_smallest, ones;
  if(x==0) return 0;
  smallest = (x & -x);
  ripple = x + smallest;
  new_smallest = (ripple & -ripple);
  ones = ((new_smallest/smallest) >> 1 ) - 1;
  return ripple | ones;
}
```

8.2 C++ input/output

```
#include <iostream>
#include <iomanip>

using namespace std;

#define db(x) cerr << #x << "=" << x << endl
#define db2(x, y) cerr << #x << "=" << x << "," << #y << "=" << y <<
    endl
#define db3(x, y, z) cerr << #x << "=" << x << "," << #y << "=" << y
    << "," << #z << "=" << z << endl

#define LSOne(S) (S & (-S))

// remove duplicated from vector
#define UNIQUE(x) x.erase(unique(x.begin(), x.end()), x.end())

// Generate all set of n elements
unsigned next_set_n(unsigned x) {
    unsigned smallest, ripple, new_smallest, ones;
    if(x==0) return 0;
    smallest = (x & -x);
    ripple = x + smallest;
    new_smallest = (ripple & -ripple);
    ones = ((new_smallest/smallest) >> 1 ) - 1;
    return ripple | ones;
}


int main()
{
    // Ouput a specific number of digits past the decimal point,
    // in this case 5
    cout.setf(ios::fixed); cout << setprecision(5);
    cout << 100.0/7.0 << endl;
    cout.unsetf(ios::fixed);

    // Output the decimal point and trailing zeros
    cout.setf(ios::showpoint);
    cout << 100.0 << endl;
    cout.unsetf(ios::showpoint);

    // Output a '+' before positive values
    cout.setf(ios::showpos);
    cout << 100 << " " << -100 << endl;
    cout.unsetf(ios::showpos);

    // Output numerical values in hexadecimal
    cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;

    // Convert int to binary string
    cout << bitset<32>(val).to_string() << endl;

}
```