Guía de Maratones
Universidad Simón Bolívar

Sánchez, Piñero, De Abreu, Colmenares

Mayo 27, 2013

UNION FIND

Implementación de Conjuntos Disjuntos

```
#define MAX 10005
int uf[MAX];
int FIND(int x){ return uf[x]==x? x:uf[x] = FIND(uf[x]); }
void UNION(int x,int y){ uf[FIND(x)] = FIND(y); }
void init(int n){
  for(int i = 0; i < n; i++){
    uf[i] = i;
  }
}
```

2SAT

1) Construct the implication graph of the instance, and find its strongly connected components using any of the known linear-time algorithms for strong connectivity analysis.

2) Check whether any strongly connected component contains both a variable and its negation. If so, report that the instance is not satisfiable and halt.

3) Construct the condensation of the implication graph, a smaller graph that has one vertex for each strongly connected component, and an edge from component i to component j whenever the implication graph contains an edge uv such that u belongs to component i and v belongs to component j. The condensation is automatically a directed acyclic graph and, like the implication graph from which it was formed, it is skew-symmetric.

4) Topologically order the vertices of the condensation; the order in which the components are generated by Kosaraju's algorithm is automatically a topological ordering.

5) For each component in this order, if its variables do not already have truth assignments, set all the terms in the component to be false. This also causes all of the terms in the complementary component to be set to true.

CICLO EULERIANO

Para saber si un grafo tiene un ciclo euleriano, basta con probar que todo nodo tiene grado par. Este algoritmo te obtiene el ciclo euleriano resultante.

```
void reorder2(vector<int> &v, int elem){
  int k;
  int tam = v.size();
  for(int i = 0; i < tam; i+=2){
    if(v[i] == elem){
      k = i;
      break;
    }
  }
  vector<int> aux(tam);
  for(int i = 0; i < tam; i++){
    aux[i] = v[i];
  }
  for(int i = 0; i < tam; i++){
    v[i] = aux[(i+k)%tam];
  }
}

bool confirm(){
  int inicial;
  inicial = g1[0];
  for(int i = 0; i < ncolor; i++){
    if(salida[i].size() %2 == 1){
      return false;
    }
  }
  vector<int> C, C2;
  vector<int> A;
  A.push_back(inicial);
  color[inicial] = 1;
  int u, w;
  while(A.size() > 0){
    u = A.back();
    if(color[u] != 1){
      A.pop_back();
      continue;
    }

    if(C.size() > 0) reorder2(C, u);

    C2.clear();
    while(tam[u] > 0){
      w = salida[u].back();
      salida[u].pop_back();
      if(v[u][w] <= 0){
        continue;
      }
      v[u][w] --;
      v[w][u]--;
      tam[u]--;
      tam[w]--;
      C2.push_back(u);
      C2.push_back(w);
      if(tam[w] < 2){
        color[w] = -1;
      }else{
        color[w] = 1;
        A.push_back(w);
      }
      u = w;
    }
    for(int i = 0; i < C2.size(); i++){
      C.push_back(C2[i]);
    }
  }

  if(C.size() < g1.size()*2){
    return false;
  }
  //circuit.push_back(circuit[0]);
  for(int i = 0; i < C.size(); i+=2){
    printf("%d %d\n",C[i], C[i+1]);
  }

  return true;
}
```

TARJAN

Tarjan for undirected graphs (Cacol)

```
/**
 * Tarjan's algorithm for finding biconnected componets in UNDIRECTED GRAPHS.
 *
 * Finds all bridges (isthmus, cut-edges), articulation points (cut-vertex),
 * and biconnected components in a provided undirected graph.
 *
 * @param MAXN          The maximum number of nodes in the graph.
 * @param graph         Adjecency-list description of the graph.
 * @param n             Number of nodes in the graph.
 * @return bridges      All bridges in the graph.
 * @return b_comp       All biconnected components in the graph, the
 *                      components are given as a set of non-repeated
 *                      edges. The direction of the edges follow
 *                      a DFS in the biconneced component.
 * @return a_points     All articulation points in the provided graph.
 *
 * @time: O(|E|+|V|)
 * @test: (uva)796, 610, 315
 */

//........Params begin..................
#define MAXN 10000
vector< vector<int> > graph;
int n;
//........Params end..................

vector< vector< pair<int,int> > > b_comp;
vector< pair<int,int> > bridges;
vector< int > a_points;
stack< pair<int,int> > stk;
int depth[MAXN], low[MAXN];

inline void new_b_comp(int v, int w){
    int x = b_comp.size();
    b_comp.push_back(vector< pair<int,int> >());
    bool stop = false;
    while(!stop){
        b_comp[x].push_back(stk.top());
        stop = stk.top().first == v && stk.top().second == w;
        stk.pop();
    }
}

void b_comp_dfs(int p, int v){
    low[v] = depth[v];
    bool is_a_point = false;
    for(int i=0;i<graph[v].size();i++){
        int w = graph[v][i];
        if(w==p || depth[w] > depth[v]) continue;
        stk.push(make_pair(v,w));
        if(depth[w]==-1){
            depth[w] = depth[v]+1;
            b_comp_dfs(v,w);
            low[v] = min(low[v],low[w]);
            if(low[w] >= depth[v]) //v is an articulation point
                is_a_point = true, new_b_comp(v,w);
            if(low[w]==depth[w]) //v - w is a bridge
                bridges.push_back(make_pair(min(v,w),max(v,w)));
        }
        else low[v] = min(low[v],depth[w]);
    }
    //root-nodes need a diffetent analysis
    if(depth[v]!=0 && is_a_point) a_points.push_back(v);
}

//Main function. The for cycle can be erased if it is
//warrantied that the graph is connected.
inline void find_b_comp(){
    b_comp.clear(), bridges.clear(), a_points.clear();
    memset(depth,-1,sizeof(int)*n);
    stk = stack< pair<int,int> >();
    for(int i=0;i<n;i++) if(depth[i]==-1) {
        depth[i] = 0, b_comp_dfs(-1,i);
        //Analyze if i is an a_point
        int count = 0;
        for(int j=0;j<graph[i].size() && count<2;j++)
            if(depth[graph[i][j]]==1) count++;
        if(count > 1) a_points.push_back(i);
    }
}
```

TARJAN2*

Tarjan for directed graphs (Cacol)

```
/**
 * Tarjan's algorithm for finding strongly connected components in DIRECTED GRAPHS.
 *
 * Calculates a partition of the graph's nodes in strongly connected components.
 *
 * @param MAXN          The maximum number of nodes in the graph.
 * @param graph         Adjecency-list description of the graph.
 * @param n             Number of nodes in the graph.
 * @return sc_comp_cnt  Contains the number of different strongly connected
 *                      components found.
 * @return sc_comp      Contains the str.conn.comp. to which it vertex belongs to,
 *                      i.e. sc_comp[i] = #of the str.conn.comp. of vertex i.
 *
 * @time: O(|E|+|V|)
 * @test: (spoj)BOTTOM
 */

//........Params begin..................
#define MAXN 5010

vector< vector<int> > graph;
int n;
//........Params end..................

int sc_comp[MAXN], sc_comp_cnt;
int visit[MAXN], low[MAXN];
bool active[MAXN];
stack<int> stk;

inline void new_sc_comp(int v){
    bool stop = false;
    while(!stop){
        sc_comp[stk.top()] = sc_comp_cnt;
        active[stk.top()] = false;
        stop = stk.top() == v;
        stk.pop();
    }
    sc_comp_cnt++;
}

void scc_dfs(int v, int &step){
    low[v] = visit[v];
    active[v] = true, stk.push(v);
    for(int i=0;i<graph[v].size();i++){
        int w = graph[v][i];
        if(visit[w] == -1){
            visit[w] = step++;
            scc_dfs(w,step);
            low[v] = min(low[v],low[w]);
        }
        else if(active[w]) low[v] = min(low[v],visit[w]);
    }
    if(low[v]==visit[v]) new_sc_comp(v);
}

inline void find_sc_comps(){
    memset(visit,-1,sizeof(int)*n);
    //memset(active,false,sizeof(bool)*n);
    sc_comp_cnt = 0;
    stk = stack<int>();
    for(int i=0,step=0;i<n;i++) if(visit[i]==-1)
        visit[i] = step++, scc_dfs(i,step);
}
```

FLOYD

O(n*n*n) n - numero de nodos cost - matriz de costos(infinito si no hay arco)

```
for(int k = 0; k < n; k++){
  for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
      cost[i][j] = min(cost[i][j], cost[i][k] + cost[k][j]);
    }
  }
}
```

BELLMAN-FORD

O(n*m) n - numero de nodos cost - matriz de costos(infinito si no hay arco)

```
for v pertenece V[G] do
  distancia[v]=INFINITO
  predecesor[v]=NIL
distancia[s]=0
for i=0 to n-1 do
  for (u, v) pertenece E[G] do
    if distancia[v] > distancia[u] + peso(u, v) then
      distancia[v] = distancia[u] + peso (u, v)
      predecesor[v] = u
```

LOWEST COMMON ANCESTOR

```
#define M 10009
#define LM 100
#define lint long long
int t, n;
char s[100];

vector<pair<int,int> > salida[M];
int L[M]; // L is the level of the node
int T[M]; // Father of node
lint C[M]; // Cost to father
lint PATH[M]; // Cost to root
int P[M][LM]; // DP path 2**i

int bfs(int ini){
  int u, v;
  pair<int,int> p1,p2;
  queue< pair<int, int> > Q;
  Q.push(make_pair(ini,0));

  for(int i = 0; i < n; i++){
    T[i] = -1;
    PATH[i] = 0;
  }

  while(!Q.empty()){
    p1 = Q.front();
    Q.pop();

    u = p1.second;

    L[u] = p1.first;

    for(int i = 0; i < salida[u].size(); i++){
      v = salida[u][i].second;
      if(T[v] == -1 && v != ini){
        T[v] = u;
        C[v] = salida[u][i].first;
        PATH[v] = PATH[u] + C[v];
        p2.first = p1.first+1;
        p2.second = v;
        Q.push(p2);
      }
    }
  }
}

void process3(int N){
  int i, j;
  //we initialize every element in P with -1
  for(i = 0; i < N; i++){
    for(j = 0; 1 << j < N; j++){
      P[i][j] = -1;
    }
  }
  //the first ancestor of every node i is T[i]
  for (i = 0; i < N; i++){
    P[i][0] = T[i];
  }
  //bottom up dynamic programing
  for(j = 1; 1 << j < N; j++){
    for(i = 0; i < N; i++){
      if(P[i][j - 1] != -1){
        P[i][j] = P[P[i][j - 1]][j - 1];
      }
    }
  }
}

int LCA(int N, int p, int q){
  int tmp, log, i;
  //if p is situated on a higher level than q then we swap them
  if(L[p] < L[q]){
    tmp = p, p = q, q = tmp;
  }
  //we compute the value of [log(L[p)]
  for(log = 1; 1 << log <= L[p]; log++);
  log--;
  //we find the ancestor of node p situated on the same level
  //with q using the values in P
  for (i = log; i >= 0; i--){
    if (L[p] - (1 << i) >= L[q]){
      p = P[p][i];
    }
  }
  if(p == q){
    return p;
  }
  //we compute LCA(p, q) using the values in P
  for (i = log; i >= 0; i--){
    if (P[p][i] != -1 && P[p][i] != P[q][i]){
      p = P[p][i], q = P[q][i];
    }
  }
  return T[p];
}

int init(){
  bfs(0);
  process3(n);
}
```

Heavy Light Decomposition*

```
#define root 0
#define N 10100
#define LN 14
#define T int
#define neutro -1
vector <int> adj[N], costs[N], indexx[N];
int baseArray[N], ptr;
int chainNo, chainInd[N], chainHead[N], posInBase[N];
int depth[N], pa[LN][N], otherEnd[N], subsize[N];
T st[N*6];
T f(T a, T b) { return a>b?a:b; }
/*
* make_tree:
* Used to construct the segment tree. It uses the baseArray for construction
*/
void make_tree(int cur, int s, int e) {
  if(s == e-1) {
    st[cur] = baseArray[s];
    return;
  }
  int c1 = (cur<<1), c2 = c1 | 1, m = (s+e)>>1;
  make_tree(c1, s, m);
  make_tree(c2, m, e);
  st[cur] = f(st[c1], st[c2]);
}
/*
* update_tree:
* Point update. Update a single element of the segment tree.
*/
void update_tree(int cur, int s, int e, int x, int val) {
  if(s > x || e <= x) return;
  if(s == x && s == e-1) {
    st[cur] = val;
    return;
  }
  int c1 = (cur<<1), c2 = c1 | 1, m = (s+e)>>1;
  update_tree(c1, s, m, x, val);
  update_tree(c2, m, e, x, val);
  st[cur] = f(st[c1], st[c2]);
}
/*
* query_tree:
* Given S and E, it will return the maximum value in the range [S,E)
*/
T query_tree(int cur, int s, int e, int S, int E) {
  if(s >= E || e <= S) { return neutro; }
  if(s >= S && e <= E) { return st[cur]; }
  int c1 = (cur<<1), c2 = c1 | 1, m = (s+e)>>1;
  return f(query_tree(c1, s, m, S, E), query_tree(c2, m, e, S, E));
```

```
}
/*
* query_up:
* It takes two nodes u and v, condition is that v is an ancestor of u
* We query the chain in which u is present till chain head, then move to next chain up
* We do that way till u and v are in the same chain, we query for that part of chain and break
*/
int query_up(int u, int v) {
  if(u == v) return 0; // Trivial
  int uchain, vchain = chainInd[v], ans = -1, q;
  // uchain and vchain are chain numbers of u and v
  while(1) {
    uchain = chainInd[u];
    if(uchain == vchain) {
      // Both u and v are in the same chain, so we need to query from u to v, update answer and break.
      // We break because we came from u up till v, we are done
      if(u==v) break;
      q = query_tree(1, 0, ptr, posInBase[v]+1, posInBase[u]+1);
      // Above is call to segment tree query function
      if(q > ans) ans = q; // Update answer
      break;
    }
    q = query_tree(1, 0, ptr, posInBase[chainHead[uchain]], posInBase[u]+1);
    // Above is call to segment tree query function. We do from chainHead of u till u. That is the whol
    // start till head. We then update the answer
    if(q > ans) ans = q;
    u = chainHead[uchain]; // move u to u's chainHead
    u = pa[0][u]; //Then move to its parent, that means we changed chains
  }
  return ans;
}
/*
* LCA:
* Takes two nodes u, v and returns Lowest Common Ancestor of u, v
*/
int LCA(int u, int v) {
  if(depth[u] < depth[v]) swap(u,v);
  int diff = depth[u] - depth[v];
  for(int i=0; i<LN; i++) if( (diff>>i)&1 ) u = pa[i][u];
  if(u == v) return u;
  for(int i=LN-1; i>=0; i--) if(pa[i][u] != pa[i][v]) {
    u = pa[i][u];
    v = pa[i][v];
  }
  return pa[0][u];
}
void query(int u, int v) {
  /*
  * We have a query from u to v, we break it into two queries, u to LCA(u,v) and LCA(u,v) to v
  */
  int lca = LCA(u, v);
  int ans = query_up(u, lca); // One part of path
  int temp = query_up(v, lca); // another part of path
  if(temp > ans) ans = temp; // take the maximum of both paths
  printf("%d\n", ans);
}
/*
* change:
* We just need to find its position in segment tree and update it
*/
void change(int i, int val) {
  int u = otherEnd[i];
  update_tree(1, 0, ptr, posInBase[u], val);
}
/*
* Actual HL-Decomposition part
* Initially all entries of chainHead[] are set to -1.
* So when ever a new chain is started, chain head is correctly assigned.
* As we add a new node to chain, we will note its position in the baseArray.
* In the first for loop we find the child node which has maximum sub-tree size.
* The following if condition is failed for leaf nodes.
* When the if condition passes, we expand the chain to special child.
* In the second for loop we recursively call the function on all normal nodes.
* chainNo++ ensures that we are creating a new chain for each normal child.
*/
void HLD(int curNode, int cost, int prev) {
  if(chainHead[chainNo] == -1) {
    chainHead[chainNo] = curNode; // Assign chain head
  }
  chainInd[curNode] = chainNo;
  posInBase[curNode] = ptr; // Position of this node in baseArray which we will use in Segtree
  baseArray[ptr++] = cost;
  int sc = -1, ncost;
  // Loop to find special child
  for(int i=0; i<adj[curNode].size(); i++) if(adj[curNode][i] != prev) {
    if(sc == -1 || subsize[sc] < subsize[adj[curNode][i]]) {
      sc = adj[curNode][i];
      ncost = costs[curNode][i];
    }
  }
  if(sc != -1) {
    // Expand the chain
    HLD(sc, ncost, curNode);
  }
  for(int i=0; i<adj[curNode].size(); i++) if(adj[curNode][i] != prev) {
    if(sc != adj[curNode][i]) {
      // New chains at each normal node
      chainNo++;
      HLD(adj[curNode][i], costs[curNode][i], curNode);
    }
  }
}
```

```
/*
* dfs used to set parent of a node, depth of a node, subtree size of a node
*/
void dfs(int cur, int prev, int _depth=0) {
  pa[0][cur] = prev;
  depth[cur] = _depth;
  subsize[cur] = 1;
  for(int i=0; i<adj[cur].size(); i++)
    if(adj[cur][i] != prev) {
      otherEnd[indexx[cur][i]] = adj[cur][i];
      dfs(adj[cur][i], cur, _depth+1);
      subsize[cur] += subsize[adj[cur][i]];
    }
}
int main() {
  int t;
  scanf("%d ", &t);
  while(t--) {
    ptr = 0;
    int n;
    scanf("%d", &n);
    // Cleaning step, new test case
    for(int i=0; i<n; i++) {
      adj[i].clear();
      costs[i].clear();
      indexx[i].clear();
      chainHead[i] = -1;
      for(int j=0; j<LN; j++) pa[j][i] = -1;
    }
    for(int i=1; i<n; i++) {
      int u, v, c;
      scanf("%d %d %d", &u, &v, &c);
      u--; v--;
      adj[u].push_back(v);
      costs[u].push_back(c);
      indexx[u].push_back(i-1);
      adj[v].push_back(u);
      costs[v].push_back(c);
      indexx[v].push_back(i-1);
    }
    chainNo = 0;
    dfs(root, -1); // We set up subsize, depth and parent for each node
    HLD(root, -1, -1); // We decomposed the tree and created baseArray
    make_tree(1, 0, ptr); // We use baseArray and construct the needed segment tree
    // Below Dynamic programming code is for LCA.
    for(int i=1; i<LN; i++)
      for(int j=0; j<n; j++)
        if(pa[i-1][j] != -1)
          pa[i][j] = pa[i-1][pa[i-1][j]];

    while(1) {
      char s[100];
      scanf("%s", s);
      if(s[0]=='D') {
        break;
      }
      int a, b;
      scanf("%d %d", &a, &b);
      if(s[0]=='Q') {
        query(a-1, b-1);
      } else {
        change(a-1, b);
      }
    }
  }
}
```

MAX FLOW - DINIC

```
#define MAXN 5009
#define MAXM 60009
#define INF 1000000009
#define lint long long
class Dinic {
        int n, m, head[MAXN], level[MAXN], s, t, work[MAXN];
        struct edge {
                int v, c, f, nxt;
                edge() {}
                edge(int v, int c, int f, int nxt): v(v), c(c), f(f), nxt(nxt) {}
        } e[MAXM];
        bool _bfs() {
                static int q[MAXN];
                memset(level, -1, sizeof(int) * n);
                int le = 0, ri = 0;
                q[ri++] = s;
                level[s] = 0;
                while(le < ri) {
                        for(int k = q[le++], i = head[k]; i != -1; i = e[i].nxt) {
                                if(e[i].f < e[i].c && level[e[i].v] == -1) {
                                        level[e[i].v] = level[k] + 1;
                                        q[ri++] = e[i].v;
                                }
                        }
                }
                return (level[t] != -1);
        }
        int _dfs(int u, int f) {
                if(u == t)
                        return f;
                for(int& i = work[u]; i != -1; i = e[i].nxt) {
                        if(e[i].f < e[i].c && level[u] + 1 == level[e[i].v]) {
                                int minf = _dfs(e[i].v, min(f, e[i].c - e[i].f));
                                if(minf > 0) {
                                        e[i].f += minf;
                                        e[i ^ 1].f -= minf;
                                        return minf;
                                }
                        }
                }
                return 0;
        }
public:
        void init(int nn, int src, int dst) {
                n = nn;
                s = src;
                t = dst;
                m = 0;
                memset(head, -1, sizeof(int) * n);
        }
        void addEdge(int u, int v, int c, int rc) {
                assert(u < n);
                assert(v < n);
                e[m] = edge(v, c, 0, head[u]);
                head[u] = m++;
                e[m] = edge(u, rc, 0, head[v]);
                head[v] = m++;
                assert(m < MAXM);
        }
        lint maxFlow() {
                lint ret = 0;
                while(_bfs()) {
                        memcpy(work, head, sizeof(int) * n);
                        while(true) {
                                int delta = _dfs(s, INF);
                                if(delta == 0)
                                        break;
                                ret = ret + delta;
                        }
                }
                return ret;
        }
};

Dinic d;

int dinic(){

  d.init(n+1, source, sink);
  while( m-- ){
      int u, v, c; scanf("%d %d %d", &u, &v, &c );
      u;
      v;
      d.addEdge(u,v,c,c);
  }
  return d.maxFlow();
}
```

MAX FLOW - MIN COST

(Standford)

```
// 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.

using namespace std;

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

const L INF = 999999999;

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);
    }
  };
```

MAX FLOW - MIN CUT

(Standford)

```
// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
//
// Running time:
//     O(|V|^3)
// INPUT:
//     - graph, constructed using AddEdge()
// OUTPUT:
//     - (min cut value, nodes in half of min cut)
#include <cmath>
#include <vector>
#include <iostream>
using namespace std;
typedef vector<int> VI;
typedef vector<VI> VVI;
const int INF = 1000000000;
pair<int, VI> GetMinCut(VVI &weights) {
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;

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

MAX BIPARTITE MATCHING - FORD FUKERSON

Se debe colocar M suficientemente grande para que quepa el sumidero. Se debe construir salida(Lista de adyacencias). Y se deben colocar las capacidades. q es un separador entre los conjuntos

```
#define M 2009
#define q 1002
int r, c, n;

int orden[M];
int sumidero = M-1;
vector<int> salida[M];
int capacidad[M][M];
const int maxnodos = M;

int from[M];
bool visitados[M];
int find_path(int source, int n) {
  stack<int> Q;
  bool terminar;
  int where, next, prev, path_cap;
  Q.push(source);

  visitados[source] = true;
  terminar = false;
  while(!Q.empty() && !terminar){
    where = Q.top();
    Q.pop();
    for(int i = 0; i < salida[where].size(); i++){
      next = salida[where][i];
      if(!visitados[next] && capacidad[where][next] > 0){
        Q.push(next);
        visitados[next]= true;
        from[next] = where;
        if(next == sumidero){
          terminar = true;
          break;
        }
      }
    }
  }
  where = sumidero;

  if(terminar == false) return 0;

  path_cap = 1;

  where = sumidero;
  while(from[where] > -1 && where != source){
    prev = from[where];
    if(prev < q){
      orden[prev] = where;
    }
    capacidad[prev][where] -= path_cap;
    capacidad[where][prev] += path_cap;
    where = prev;
  }
  return path_cap;
}

int max_flow(int n) {
  int result, path_capacity;
  result = 0;

  for(int i = 0; i < n; i++){
    memset(from, -1, sizeof(from));
    memset(visitados, 0, sizeof(visitados));
    path_capacity = find_path(i, n);
    result += path_capacity;
  }
  return result;
}
```

TEOREMA DE KONING - FORD FUKERSON

Se debe usar el algoritmo de Max Bipartite Matching - ford fukerson.
solx son las rectas verticales y soly son las rectas horizontales

```
bool solx[M];
bool soly[M];
bool visit[M];

int BFS(int first){
  visit[first] = true;
  int actual, next;
  queue<int> cola;
  cola.push(first);

  while(!cola.empty()){
    actual = cola.front();
    cola.pop();
    visit[actual] = true;
    for(int i = 0; i < salida[actual].size(); i++){
      next = salida[actual][i];
      if(visit[next] || next == sumidero) continue;
      if(actual < q && orden[actual] != next){
        cola.push(next);
      }
      if(actual >= q && ordeninv[actual] == next){
        cola.push(next);
      }
    }
  }
}

int result(){
  for(int i = 0; i < M; i++){
    solx[i] = false;
    soly[i] = false;
  }
  int num = max_flow(r);
  memset(visit, 0, sizeof(visit));
  for(int i = 0; i < r; i++){
    if(salida[i].size() > 0 && orden[i] < 0){
      BFS(i);
    }
  }
  for(int i = 0; i < r; i++){
    if(salida[i].size() > 0 && !visit[i]) soly[i] = true;
  }
  for(int i = 0; i < c; i++){
    if(salida[i+q].size() > 0 && visit[i+q]) solx[i] = true;
  }
  return num;
}
```

MAXBIPARTITE MATCHING - HOPCROFT KARP

n: número de nodos a la izquierda, numerados de 1 a n m: número de nodos a la derecha, numerados de n+1 a n+m G = NIL[0] âĹł G1[G[1—n]] âĹł G2[G[n+1—n+m]]

```
// n: number of nodes on left side, nodes are numbered 1 to n
// m: number of nodes on right side, nodes are numbered n+1 to n+m
// G = NIL[0] U G1[G[1---n]] U G2[G[n+1---n+m]]

#define MAX 100001
#define NIL 0
#define INF (1<<28)

vector<int> G[MAX];
int n, m, match[MAX], dist[MAX];

bool bfs() {
    int i, u, v, len;
    queue< int > Q;
    for(i=1; i<=n; i++) {
        if(match[i]==NIL) {
            dist[i] = 0;
            Q.push(i);
        }
        else dist[i] = INF;
    }
    dist[NIL] = INF;
    while(!Q.empty()) {
        u = Q.front(); Q.pop();
        if(u!=NIL) {
            len = G[u].size();
            for(i=0; i<len; i++) {
                v = G[u][i];
                if(dist[match[v]]==INF) {
                    dist[match[v]] = dist[u] + 1;
                    Q.push(match[v]);
                }
            }
        }
    }
    return (dist[NIL]!=INF);
}

bool dfs(int u) {
    int i, v, len;
    if(u!=NIL) {
        len = G[u].size();
        for(i=0; i<len; i++) {
            v = G[u][i];
            if(dist[match[v]]==dist[u]+1) {
                if(dfs(match[v])) {
                    match[v] = u;
                    match[u] = v;
                    return true;
                }
            }
        }
        dist[u] = INF;
        return false;
    }
    return true;
}

int hopcroft_karp() {
  int matching = 0, i;
  // match[] is assumed NIL for all vertex in G
  while(bfs())
    for(i=1; i<=n; i++)
      if(match[i]==NIL && dfs(i))
        matching++;
  return matching;
}
```

MAX MATCHING - GABOW

```
/**
 * Gabow's algorithm for finding a maximum match in a general (undirected) graph.
 *
 * Calculates a matching of the graphs in the array match[].
 * IMPORTANT: the provided graph must have three properties:
 * 1) The nodes are numbered from 1 to n. Node #0 is reserved for internal use.
 * 2) The provided graph must be an edge adjacency-list, which means that
 *      graph[v][i] must return the number of an edge. For knowing the nodes
 *      incident in an edge the array edge[][] is used.
 * 3) The edges are numbered from 0 to m-1.
 *
 * @param MAXN          The maximum number of nodes in the graph.
 * @param MAXM          The maximum number of edges in the graph.
 * @param graph         Edge adjecency-list description of the graph.
 * @param edge          Description of incident nodes in each edge. The edges
 *                      are numbered from 0 to m-1.
 * @param n             Number of nodes in the graph.
 * @param m             Number of edges in the graph.
 *
 * @return match        For every vertex i (between 1 and n), if i is a
 *                      matched vertex, then the edge (i,match[i]) is matched.
 *                      If i is not matched, then match[i] == 0.
 *
 * @time: O(|V|^3)
 * @test: (livearchive)4130, (timus)1099
 */

//........Params begin..................
#define MAXN 100
#define MAXM 100*100

vector< vector<int> > graph;
int edge[MAXM+1][2];
int n,m;
//........Params end..................

int match[MAXN+1];
int label[MAXN+1],first[MAXN+1];
queue<int> q;

void rematch(int v, int w){
    int t = match[v];
    match[v] = w;
    if(match[t] != v) return;
    if(label[v] <= n){
        match[t] = label[v];
        return rematch(label[v],t);
    }
    int e = label[v]-1-n,
        x = edge[e][0], y = edge[e][1];
    rematch(x,y);
    rematch(y,x);
}

inline void edge_label(int e){
    int x = edge[e][0], y = edge[e][1],
        r = first[x], s = first[y],
        flag = n+1+e, join;
    if(r==s) return;
    label[r] = label[s] = -flag;
    while(true){
        if(s!=0){int aux = s; s = r, r = aux;}
        r = first[label[match[r]]];
        if(label[r]==-flag){
            join = r;
            break;
        }
        label[r] = -flag;
    }
    for(int i=0;i<2;i++){
        int v = first[edge[e][i]];
        while(v!=join){
            label[v] = flag, first[v] = join;
            q.push(v);
            v = first[label[match[v]]];
        }
    }
    for(int i=1;i<=n;i++)
        if(label[i]>=0 && label[first[i]] >= 0)
            first[i] = join;
}

inline bool augpath(int u){
    memset(label,-1,sizeof(int)*(n+1));
    label[u] = first[u] = 0;
    q = queue<int>();
    q.push(u);
    while(!q.empty()){
        int x = q.front(); q.pop();
        for(int i=0;i<graph[x].size();i++){
            int e = graph[x][i],
                y = (edge[e][0] != x ? edge[e][0] : edge[e][1]);
            if(match[y] == 0 && y != u) {
                match[y] = x;
                rematch(x,y);
                return true;
            }
            if(label[y]>=0){
                edge_label(e);
                continue;
            }
            int v = match[y];
            if(label[v]<0){
                label[v] = x, first[v] = y;
                q.push(v);
            }
        }
    }
    return false;
}

//Main function, returns the number of matched edges found
inline int gabow(){
    memset(match,0,sizeof(int)*(n+1));
    int tot = 0;
    for(int i=1;i<=n;i++) {
        if(match[i]!=0) continue;
        if(augpath(i)) tot++;
    }
    return tot;
}
```

MAX BIPARTITE MATCHING MIN COST

Algoritmo Húngaro. (STANDFORD)

```
///////////////////////////////////////////////////////////////////////////
// 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.
///////////////////////////////////////////////////////////////////////////

#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;

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;
}
```

STABLE MARRIAGE PROBLEM

Problema del matrimonio estable. Realiza matchings de hombres y mujeres tal que nunca nadie va a engañar a su pareja, porque prefiere a su pareja que al otro.

```
int gm[509][509]; //gm[i][j] mujer: que prefiere el hombre i en la posiciÃşn j
int gw[509][509]; //gw[i][j] hombre: que prefiere la mujer i en la posiciÃşn j

int pm[509][509]; //pm[i][j] posiciÃşn, de la mujer j preferido por i
int pw[509][509]; //pw[i][j] posiciÃşn, del hombre j preferido por i

int nm[509];

int engmw[509]; // engaged man to woman
int engwm[509]; // engaged woman to man

bool fm[509]; // FREE MAN
bool fw[509]; // FREE WOMAN

int n, t, e;

int stableMatching(){
  queue<int> mans;
  int m, m2, w;
  for(int i = 0; i < n; i++){
    fm[i] = true;
    fw[i] = true;
    mans.push(i);
    nm[i] = 0;
  }

  while(!mans.empty()){
    m = mans.front();
    mans.pop();

    for(; nm[m] < n; nm[m]++){
      w = gm[m][nm[m]];

      if(fw[w]){
        fw[w] = false;
        fm[m] = false;
        engmw[m] = w;
        engwm[w] = m;

        break;
      }else{
        m2 = engwm[w];
        if(pw[w][m] < pw[w][m2]){
          fm[m2] = true;
          fw[w] = false;
          fm[m] = false;
          engmw[m] = w;
          engwm[w] = m;
          mans.push(m2);
          break;
        }
      }

    }

    if(fm[m]){

      nm[m] = 0;
      mans.push(m);
    }
  }
}


int format(){

  for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
      scanf("%d", &gw[e][j]); // WOMAN i: el preferido j
      gw[j][j]--;
      pw[j][gw[e][j]] = j; // WOMAN i: prefiere a gm[e][j] de lugar j

    }
  }
  for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
      scanf("%d", &gm[e][j]); // MAN i: la preferida j
      gm[e][j]--;
      pm[e][gm[e][j]] = j; // MAN i: prefiere a gw[e][j] de lugar j
    }
  }
  stableMatching();
  for(int i = 0; i < n; i++){
    printf("%d %d\n", i+1, engmw[i]+1); // print matches
  }

  return 0;

}
```

MEDIAN OF TWO ARRAYS

```c
#include<stdio.h>

int max(int, int); /* to get maximum of two integers */
int min(int, int); /* to get minimum of two integeres */
int median(int [], int); /* to get median of a sorted array */

/* This function returns median of ar1[] and ar2[].
   Assumptions in this function:
   Both ar1[] and ar2[] are sorted arrays
   Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    int m1; /* For median of ar1 */
    int m2; /* For median of ar2 */

    /* return -1  for invalid input */
    if (n <= 0)
        return -1;

    if (n == 1)
        return (ar1[0] + ar2[0])/2;

    if (n == 2)
        return (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1])) / 2;

    m1 = median(ar1, n); /* get the median of the first array */
    m2 = median(ar2, n); /* get the median of the second array */

    /* If medians are equal then return either m1 or m2 */
    if (m1 == m2)
        return m1;

     /* if m1 < m2 then median must exist in ar1[m1....] and ar2[....m2] */
    if (m1 < m2)
    {
        if (n % 2 == 0)
            return getMedian(ar1 + n/2 - 1, ar2, n - n/2 +1);
        else
            return getMedian(ar1 + n/2, ar2, n - n/2);
    }

    /* if m1 > m2 then median must exist in ar1[....m1] and ar2[m2...] */
    else
    {
        if (n % 2 == 0)
            return getMedian(ar2 + n/2 - 1, ar1, n - n/2 + 1);
        else
            return getMedian(ar2 + n/2, ar1, n - n/2);
    }
}
```

```c
/* Function to get median of a sorted array */
int median(int arr[], int n)
{
    if (n%2 == 0)
        return (arr[n/2] + arr[n/2-1])/2;
    else
        return arr[n/2];
}


/* Utility functions */
int max(int x, int y)
{
    return x > y? x : y;
}

int min(int x, int y)
{
    return x > y? y : x;
}

/* Driver program to test above function */
int main()
{
    int ar1[] = {1, 2, 3, 6};
    int ar2[] = {4, 6, 8, 10};
    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
      printf("Median is %d", getMedian(ar1, ar2, n1));
    else
     printf("Doesn't work for arrays of unequal size");
    return 0;
}
```

AVL TREE*

Answer query number of nodes greater than k.

```
namespace AVL{

  struct node{
    int key;
    int val;
    int num;
    unsigned char height;
    node* left;
    node* right;
    node(int k) { key = k; left = right = 0; height = 1; val = 1; num = 1;}
  };

  unsigned char height(node* p){
    return p?p->height:0;
  }

  int bfactor(node* p){
    return height(p->right)-height(p->left);
  }

  void fixnum(node * p){
    p->num = p->val;
    if(p->left != 0) p->num = p->num+p->left->num;
    if(p->right != 0) p->num = p->num+p->right->num;
  }

  void fixheight(node* p){
    unsigned char hl = height(p->left);
    unsigned char hr = height(p->right);
    p->height = (hl>hr?hl:hr)+1;
    fixnum(p);
  }

  node* rotateright(node* p){
    node* q = p->left;
    p->left = q->right;
    q->right = p;
    fixheight(p);
    fixheight(q);
    return q;
  }

  node* rotateleft(node* q){
    node* p = q->right;
    q->right = p->left;
    p->left = q;
    fixheight(q);
    fixheight(p);
    return p;
  }

  node* balance(node* p){ // balancing the p node
    fixheight(p);
    if( bfactor(p)==2 )
    {
        if( bfactor(p->right) < 0 )
            p->right = rotateright(p->right);
        return rotateleft(p);
    }
    if( bfactor(p)==-2 )
    {
        if( bfactor(p->left) > 0  )
            p->left = rotateleft(p->left);
        return rotateright(p);
    }
    return p; // balancing is not required
  }


  // p = insert(p,k);
  node* insert(node* p, int k){ // insert k key in a tree with p root
    if( !p ) return new node(k);
    if( k<p->key )
        p->left = insert(p->left,k);
    else
        p->right = insert(p->right,k);
    return balance(p);
  }

  node* findmin(node* p){ // find a node with minimal key in a p tree
    return p->left?findmin(p->left):p;
  }

  node* removemin(node* p){ // deleting a node with minimal key from a p tree
    if( p->left==0 )
        return p->right;
    p->left = removemin(p->left);
    return balance(p);
  }
  // p = remove(p,k);
  node* remove(node* p, int k){ // deleting k key from p tree
    if( !p ) return 0;
    if( k < p->key )
        p->left = remove(p->left,k);
    else if( k > p->key )
        p->right = remove( p->right,k);
    else{ //  k == p->key
        node* q = p->left;
        node* r = p->right;
```

```
        delete p;
        if( !r ) return q;
        node* min = findmin(r);
        min->right = removemin(r);
        min->left = q;
        return balance(min);
    }
    return balance(p);
  }
  node* find(node*p, int k){
    if(!p) return 0;
    if( k < p->key )
        return find(p->left,k);
    else if( k > p->key )
        return find(p->right,k);
    else {
        return p;
    }
  }

  void bal(node *p, int k){ // restore the balance
    if( k < p->key )
      bal(p->left,k);
    else if( k > p->key )
      bal(p->right,k);
    fixnum(p);
  }

  // Num of nodes >= to k
  int query(node* p, int k){
    if( !p ) return 0;
    if( k < p->key ){
      return p->val + (p->right==0?0:p->right->num) + query(p->left,k) ;
    }else if( k > p->key ){
      return query(p->right,k);
    }else {
      return p->val + (p->right==0?0:p->right->num);
    }
  }

}
```

Binary Indexed Tree

Binary Index Tree de una dimensión. (Cacol)

```
#define MAXN 1000

//Tree has the BIT, n is the num of elements in the array
//and bitmask = 2^floor(lg2(n))
int tree[MAXN+1], n, bitmask;

//Returns the accumulated sum up to
//idx. However the first element of the array
//is 1 (not 0, careful witht that).
int read(int idx){
    int tot = 0;
    while(idx > 0){
        tot += tree[idx];
        idx ^= (idx&-idx);
    }
    return tot;
}

//Adds c to the array cell idx (1-indexed)
//and updates the structure
int update(int idx,int c){
    while(idx <= n){
        tree[idx] += c;
        idx += (idx&-idx);
    }
}

//Finds the right-most element which accumulated sum
//is equal to s. The array is 1-indexed and
//only works if it does not have non-negative numbers.
int find(int s){
    int mask = bitmask, idx = 0;
    while(mask > 0){
        int tidx = idx+mask;
        if(tidx > n){mask>>=1;continue;}
        if(s >= tree[tidx]){
            s -= tree[tidx];
            idx = tidx;
        }
        mask >>= 1;
    }
    if(s!=0) return 0;
    return idx;
}
```

Binary Indexed Tree 2D

Binary Index Tree de dos dimensiones.

```
int max_x;
int max_y;
int tree[1030][1030];

int read2(int idx, int idy){
        int sum = 0;
        while (idy > 0){
                sum += tree[idx][idy];
                idy -= (idy & -idy);
        }
        return sum;
}
int read(int idx, int idy){
        int sum = 0;
        while (idx > 0){
                sum += read2(idx, idy);
                idx -= (idx & -idx);
        }
        return sum;
}
void update(int x , int y , int val){
        int y1;
        while (x <= max_x){
                y1 = y;
                while (y1 <= max_y){
                        tree[x][y1] += val;
                        y1 += (y1 & -y1);
                }
                x += (x & -x);
        }
}

result = read(x2+1,y2+1);
if(x1 > 0 && y1 == 0){
  result += -read(x1,y2+1);
}else if(x1 == 0 && y1 > 0){
  result += - read(x2+1,y1);
}else{
  result += -read(x1,y2+1) - read(x2+1,y1) + read(x1,y1);
}
```

RANGE TREE*

Range tree general.

```
#define MAXN 100000
#define T int
#define neutro 0//Neutro es el neutro de la funcion del segment tree
//El indice 0 se desperdicia, se guardan los datos desde el indice 1
T input[MAXN];
//El tamano del arbol debe ser N*logN, en este caso el log 100000 es 17 aprox
T tree[MAXN*17];

//Esta es la funcion que define lo que guarda el segment tree
//En este caso es suma pero puede ser maximo o minimo
T f(T a, T b) {
    return a + b;
}

// init(1,1,n)
void init(int node, int b, int e) {
        if(b == e) tree[node] = input[b];
        else {
                int m = (b + e) >> 1, lt = node << 1, rt = lt | 1;
                init(lt, b, m);
                init(rt, m+1, e);
                tree[node] = f(tree[lt], tree[rt]);
        }
}

// query(1,1,N,i,j)
T query(int node, int b, int e, int i, int j) {
        if(i > e || j < b) return neutro;
        if (i <= b && e <= j) return tree[node];
        else {
                int m = (b + e) >> 1, lt = node << 1, rt = lt | 1;
                return f(query(lt, b, m, i, j), query(rt, m+1, e, i, j));
        }
}

void modify(int node, int b, int e, int i, int v) {
  if(i > e || i < b) return;
  if (i <= b && e <= i) {
    tree[node] = v;
    return;
  }
  int m = (b + e) >> 1, lt = node << 1, rt = lt | 1;
  modify(lt, b, m, i, v);
  modify(rt, m+1, e, i, v);
  tree[node] = f(tree[lt], tree[rt]);
}
```

RANGE TREE LAZY PROPAGATION*

Range tree general con lazy propagation.

```
#define MAXN 100000
#define T int
#define neutro 0
// entrada, arbol, acumulado(propagacion) elemento 0 se desperdicia
T input[MAXN]; T tree[MAXN*17]; T acum[MAXN*17];
T f(T a, T b) { return a + b; } // Funcion que mantiene el segment tree
// init(1,1,n)
void init(int node, int b, int e) {
  acum[node] = 0;
        if(b == e) tree[node] = input[b];
        else {
                int m = (b + e) >> 1, lt = node << 1, rt = lt | 1;
                init(lt, b, m);
                init(rt, m+1, e);
                tree[node] = f(tree[lt], tree[rt]);
        }
}
// query(1,1,N,i,j)
T query(int node, int b, int e, int i, int j) {
  int m = (b + e) >> 1, lt = node << 1, rt = lt | 1;

  tree[node] += acum[node] * (e - b + 1);
  acum[lt] += acum[node];
  acum[rt] += acum[node];
  acum[node] = 0;

        if(i > e || j < b) return neutro;
        if (i <= b && e <= j) return tree[node];
        else return f(query(lt, b, m, i, j), query(rt, m+1, e, i, j));
}
// modify(1,1,N,i,j,val)
void modify(int node, int b, int e, int i, int j, int v) {
  int m = (b + e) >> 1, lt = node << 1, rt = lt | 1;
  tree[node] += acum[node] * (e - b + 1);
  acum[lt] += acum[node];
  acum[rt] += acum[node];
  acum[node] = 0;
  if(i > e || j < b) return;
        if (i <= b && e <= j) {
          tree[node] += v * (e - b + 1);
          acum[lt] += v;
    acum[rt] += v;
          return;
        }
        modify(lt, b, m, i, j, v);
        modify(rt, m+1, e, i, j, v);
        tree[node] = f(tree[lt], tree[rt]);
}
```

KMP

Retorna las posiciones donde ocurren los matches. Cortesía de Chococontest.

```
vector<int> KMP(string S, string K){
  vector<int> T(K.size() + 1, -1);
  for(int i = 1; i <= K.size(); i++){
    int pos = T[i - 1];
    while(pos != -1 && K[pos] != K[i - 1]) pos = T[pos];
    T[i] = pos + 1;
  }

  vector<int> matches;
  for(int sp = 0, kp = 0; sp < s.size(); sp++){
    while(kp != -1 && (kp == K.size() || K[kp] != S[sp]))
         kp = T[kp];
    kp++;
    if(kp == K.size()) matches.push_back(sp + 1 - K.size());
  }

  return matches;
}
```

RABIN KARP*

Chequea si k substring de s.

```
#define B 261
const int L = 2000000;
unsigned long long H[L],powB[L];

string sw; // Sum of strings.
unsigned long long get_hash(int l, int r){
    return H[r + 1] - H[l] * powB[r - l + 1];
}
void init(){
  powB[0] = 1;
  for(int i = 1;i < L;++i){
    powB[i] = powB[i - 1] * B;
  }
  H[0] = 0;
  for(int i = 0;i < sw.size();++i){
    H[i + 1] = sw[i] + H[i] * B;
  }
}
// px, py pointers, lx, ly lengths
bool check(int px, int lx, int py, int ly){
    unsigned long long hx = get_hash(px,px + lx - 1);
    for(int i = 0;i + lx <= ly;++i)
        if(get_hash(py + i,py + i + lx - 1) == hx)
            return true;
    return false;
}
```

AHO CORASICK*

```
namespace aho { // aho-Corasick's algorithm
    const int MAXN = 3000005;
    map<char, int> g[MAXN];
    int f[MAXN]; // failure
    vector<int> output[MAXN];
    int n; // state count
    vector<string> plants;


    // n should be the sum of the lenghts of all substrings
    void reset() {
        n = 1;
        g[0].clear();
        output[0].clear();
        f[0] = 0;
        plants.clear();
    }
    // add s
    void add(const char *s) {
        int state = 0;
        int id = plants.size();
        plants.push_back(string(s));
        for (int i = 0; s[i]; i++) {
            char c = s[i];
            if (g[state].count(c) == 0) {
                g[state][c] = n;
                g[n].clear();
                output[n].clear();
                f[n] = -1;
                n++;
            }
            state = g[state][c];
        }
        output[state].push_back(id);
    }

    void prepare() { // the BFS step
        queue<int> q;
        f[0] = 0;
        for (char c = 'A'; c <= 'z'; ++c) {
            if (g[0].count(c) == 0) {
                g[0][c] = 0;
            } else {
                int s = g[0][c];
                f[s] = 0;
                q.push(s);
            }
        }
        while (q.size() > 0) {
            int u = q.front(); q.pop();
            for (map<char, int>::iterator i = g[u].begin(); i != g[u].end(); ++i) {
                char label = i->first;
                int node = i->second;
                f[node] = f[u];
                while (g[f[node]].count(label) == 0) {
                    f[node] = f[f[node]];
                }
                f[node] = g[f[node]][label];
                output[node].insert(output[node].end(), output[f[node]].begin(), output[f[node]].end())
                q.push(node);
            }
        }
    }

    int next_state(int state, char label) {
        while (g[state].count(label) == 0) {
            state = f[state];
        }
        return g[state][label];
    }
    // Todos los substring de id
    vector<int> subs(int id){
      string &s = plants[id];
      vector<int> res;
      int state = 0;
      for (int k = 0; k < s.size(); ++k) {
          char next = s[k];
          state = next_state(state, next);
          for (int e = 0; e < output[state].size(); ++e) {
              int to = output[state][e];
              if (plants[to].size() < s.size()) { // found a proper substring
                  res.push_back(to);
              }
          }
      }
      return res;
    }

}
```

SUFFIX ARRAY

Construye el arreglo de sufijos.

```
#include<iostream>
#include<string>
#include<algorithm>
using namespace std;
#define MAXN 1000005
int n,t;  //n es el tamaÃśo de la cadena
int p[MAXN],r[MAXN],h[MAXN];
//p es el inverso del suffix array, no usa indices del suffix array ordenado
//h el el tamaÃśo del lcp entre el i-esimo y el i+1-esimo elemento de suffix array ordenado
string s;
void fix_index(int *b, int *e) {
   int pkm1, pk, np, i, d, m;
   pkm1 = p[*b + t];
   m = e - b; d = 0;
   np = b - r;
   for(i = 0; i < m; i++) {
      if (((pk = p[*b+t]) != pkm1) && !(np <= pkm1 && pk < np+m)) {
         pkm1 = pk;
         d = i;
      }
      p[*(b++)] = np + d;
   }
}

bool comp(int i, int j) {
   return p[i + t] < p[j + t];
}
void suff_arr() {
   int i, j, bc[256];
   t = 1;
   for(i = 0; i < 256; i++) bc[i] = 0;  //alfabeto
   for(i = 0; i < n; i++) ++bc[int(s[i])]; //counting sort inicial del alfabeto
   for(i = 1; i < 256; i++) bc[i] += bc[i - 1];
   for(i = 0; i < n; i++) r[--bc[int(s[i])]] = i;
   for(i = n - 1; i >= 0; i--) p[i] = bc[int(s[i])];
   for(t = 1; t < n; t *= 2) {
      for(i = 0, j = 1; i < n; i = j++) {
         while(j < n && p[r[j]] == p[r[i]]) ++j;
         if (j - i > 1) {
            sort(r + i, r + j, comp);
            fix_index(r + i, r + j);
         }
      }
   }
}

void lcp() {
   int tam = 0, i, j;
   for(i = 0; i < n; i++)if (p[i] > 0) {
      j = r[p[i] - 1];
      while(s[i + tam] == s[j + tam]) ++tam;
      h[p[i] - 1] = tam;
      if (tam > 0) --tam;
   }
   h[n - 1] = 0;
}

int main(){
   s="banana";
   n=s.size();
   suff_arr();
   lcp();
   for(int i=0;i<n;i++)cout<<r[i]<<" ";cout<<endl;
   for(int i=0;i<n;i++)cout<<h[i]<<" ";cout<<endl;
   return 0;
}
```

SUFFIX ARRAY

Construye el arreglo de sufijos. (cacol)

```
/***************************************************************************
 * Suffix Array builder.
 * @input
 *  s[MAXN] The string to process
 *  n       The size of the string, including the terminator!
 *
 * @output
 *  sa[MAXN]  The suffix array
 *  lcp[MAXN] The LCP array.
 *
 * @time O(n*log(n))
 ***************************************************************************/

#define MAXN 1000
#define ALPH_SIZE 256

int sa_[MAXN],*sa = &sa_[0], lcp[MAXN], n;
int sa2_[MAXN], grp_[MAXN], grp2_[MAXN], gs[MAXN], g,
    *sa2=&sa2_[0], *grp=&grp_[0], *grp2=&grp2_[0];

void suffix_array(char *s){
    memset(gs,0,sizeof(int)*ALPH_SIZE);
    for(int i=0;i<n;i++) gs[s[i]]++;
    for(int i=1;i<ALPH_SIZE;i++) gs[i] += gs[i-1];
    for(int i=n-1;i>=0;i--) sa[--gs[s[i]]] = i;

    grp[sa[0]] = gs[0] = g = 0;
    for(int i=1;i<n;i++){
        if(s[sa[i]] != s[sa[i-1]]) gs[++g] = i;
        grp[sa[i]] = g;
    }
    g++;
    for(int d=1;d<n&&g<n;d<<=1){
        for(int i=0;i<d;i++) sa2[gs[grp[n-i-1]]++] = n-i-1;
        for(int i=0;i<n;i++) if(sa[i]>=d) sa2[gs[grp[sa[i]-d]]++] = sa[i]-d;
        int *aux = sa; sa = sa2, sa2 = aux;

        grp2[sa[0]] = gs[0] = g = 0;
        for(int i=1;i<n;i++){
            if(grp[sa[i]] != grp[sa[i-1]] || sa[i]+d>=n || sa[i-1]+d>=n ||
               grp[sa[i]+d] != grp[sa[i-1]+d]) gs[++g] = i;
            grp2[sa[i]] = g;
        }
        g++;
        aux = grp; grp = grp2, grp2 = aux;
    }
    lcp[0] = 0;
    for(int i=0,l=0;i<n-1;i++,l--){
        if(l<0) l = 0;
        while(s[i+l] == s[sa[grp[i]-1]+l]) l++;
        lcp[grp[i]] = l;
    }
}

/*************************************************************************
 * Suffix Tree builder.
 * @input
 *   sa[MAXN]   The suffix array
 *   lcp[MAXN] The LCP array.
 *   n          The size of the string
 *
 * @output
 *   sft    A pointer to the root of the suffix array
 *
 * @time O(n)
 *************************************************************************/

struct sufft {
  int b,l;
  sufft *f;
  vector<sufft*> *sons;

  sufft(){sons = new vector<sufft*>();f = NULL;}

  sufft(int b,int l, sufft *f){
    this->b = b;
    this->l = l;
    this->f = f;
    sons = new vector<sufft*>();
  }
};

sufft *sft;

void suffix_tree(){
  stack<int> st = stack<int>();
  sufft *root = new sufft(),
        *it = new sufft(sa[0],n-sa[0],root);
  root->b = root->l = 0;
  root->sons->push_back(it);

  st.push(0);
  for(int i=1;i<n;i++){
    if(lcp[i] == lcp[st.top()]){
      int j = sa[i]+lcp[i];
      sufft *x = new sufft(j, n-j, it->f);
      it->f->sons->push_back(x);
      it = x;
      st.push(i);
    }
    else if(lcp[i] > lcp[st.top()]){
      int ld = lcp[i] - lcp[st.top()],
          j = sa[i]+lcp[i];
      sufft *nit = new sufft(it->b+ld,it->l-ld,it),
            *x = new sufft(j, n-j, it);

      it->l = ld;
      vector<sufft*> *aux = nit->sons;
      nit->sons = it->sons;
      it->sons = aux;

      aux->push_back(nit); aux->push_back(x);

      it = x;
      st.push(i);
    }
    else {
      while(lcp[i] < lcp[st.top()]){
        int sz = it->f->sons->size();
        for(int j=0;j<sz-1;j++) st.pop();
        it = it->f;
      }
      i--;
    }
  }
  sft = root;
}

/*************************************************************************
 * Suffix Array builder.
 * @input
 *   s[MAXN] The string to process
 *   n       The size of the string
 *
 * @output
 *   sa[MAXN]  The suffix array
 *   lcp[MAXN] The LCP array.
 *
 * @time O(n)
 *************************************************************************/

inline bool leq(int a1, int a2,   int b1, int b2) {
  return(a1 < b1 || a1 == b1 && a2 <= b2);
}
inline bool leq(int a1, int a2, int a3,   int b1, int b2, int b3) {
  return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3));
}
static void radixPass(int* a, int* b, int* r, int n, int K)
{
  int* c = new int[K + 1];
  for (int i = 0;  i <= K;  i++) c[i] = 0;
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;
  for (int i = 0, sum = 0;  i <= K;  i++) {
     int t = c[i];  c[i] = sum;  sum += t;
  }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];
  delete [] c;
}

void suffixArray(int *s, int* SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  int* s12  = new int[n02 + 3];  s12[n02]= s12[n02+1]= s12[n02+2]=0;
  int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int* s0   = new int[n0];
  int* SA0  = new int[n0];

  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;

  radixPass(s12 , SA12, s+2, n02, K);
  radixPass(SA12, s12 , s+1, n02, K);
  radixPass(s12 , SA12, s  , n02, K);

  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0;  i < n02;  i++) {
    if (s[SA12[i]] != c0 || s[SA12[i]+1] != c1 || s[SA12[i]+2] != c2) {
      name++;  c0 = s[SA12[i]];  c1 = s[SA12[i]+1];  c2 = s[SA12[i]+2];
    }
    if (SA12[i] % 3 == 1) { s12[SA12[i]/3]      = name; }
    else                  { s12[SA12[i]/3 + n0] = name; }
  }

  if (name < n02) {
    suffixArray(s12, SA12, n02, name);
    for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
  } else
    for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;

  for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) s0[j++] = 3*SA12[i];
  radixPass(s0, SA0, s, n0, K);

  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI();
    int j = SA0[p];
    if (SA12[t] < n0 ?
        leq(s[i],       s12[SA12[t] + n0], s[j],       s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
    {
      SA[k] = i;  t++;
      if (t == n02) {
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
      }
    } else {
      SA[k] = j;  p++;
      if (p == n0)  {
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
      }
    }
  }
  delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;
}

int s[MAXN], sa[MAXN], lcp[MAXN], n;
inline void suffix_array(int alph_size){
  s[n] = s[n+1] = s[n+2] = 0;
  suffixArray(s,sa,n,alph_size);

  int *pos = new int[n];
  for(int i=0;i<n;i++) pos[sa[i]] = i;

  for(int i=0,l=0;i<n;i++,l--){
    if(l<0) l = 0;
    int j = (pos[i] == 0 ? n : sa[pos[i]-1]);
    while(s[i+l] == s[j+l]) l++;
    lcp[pos[i]] = l;
  }
  lcp[0] = 0;

  delete [] pos;
}
```

SUFFIX ARRAY - MAX SUBSTRING

```
#include<iostream>
#include<string>
#include<algorithm>
#include<cstdio>
#include<cstring>

using namespace std;

string words[35];
int len[35], N;
bool esta[35];

#define MAXN 1000005
int n,t,n1,n2; //n es el tamao de la cadena
int p[MAXN],r[MAXN],h[MAXN];
//p es el inverso del suffix array, no usa indices del suffix array ordenado
//h el el tamao del lcp entre el i-esimo y el i+1-esimo elemento de suffix array ordenado
string s, s1;
void fix_index(int *b, int *e) {
    int pkm1, pk, np, i, d, m;
    pkm1 = p[*b + t];
    m = e - b; d = 0;
    np = b - r;
    for(i = 0; i < m; i++) {
        if (((pk = p[*b+t]) != pkm1) && !(np <= pkm1 && pk < np+m)) {
            pkm1 = pk;
            d = i;
        }
        p[*(b++)] = np + d;
    }
}
bool comp(int i, int j) {
    return p[i + t] < p[j + t];
}
void suff_arr() {
    int i, j, bc[256];
    t = 1;
    for(i = 0; i < 256; i++) bc[i] = 0; //alfabeto
    for(i = 0; i < n; i++) ++bc[int(s[i])]; //counting sort inicial del alfabeto
    for(i = 1; i < 256; i++) bc[i] += bc[i - 1];
    for(i = 0; i < n; i++) r[--bc[int(s[i])]] = i;
    for(i = n - 1; i >= 0; i--) p[i] = bc[int(s[i])];
    for(t = 1; t < n; t *= 2) {
        for(i = 0, j = 1; i < n; i = j++) {
            while(j < n && p[r[j]] == p[r[i]]) ++j;
            if (j - i > 1) {
                sort(r + i, r + j, comp);
                fix_index(r + i, r + j);
            }
        }
    }
}
void lcp() {
    int tam = 0, i, j;
    for(i = 0; i < n; i++)if (p[i] > 0) {
        j = r[p[i] - 1];
        while(s[i + tam] == s[j + tam]) ++tam;
        h[p[i] - 1] = tam;
        if (tam > 0) --tam;
    }
    h[n - 1] = 0;
}

int indexToWord(int n) {
    int i = 0;
    while (n+1 > len[i]) i++;
    //cout << "Tengo " << n+1 << " y len " << len[i] << " con i " << i << endl;
    if (n+1 == len[i]) return -1;
    return i;
}

int calcularMin(int ini, int fin) {
    if (ini == fin) return n - (r[ini]+1);
    int r = h[ini];
    for (int i = ini+1; i < fin; i++) {
        r = min(r, h[i]);
    }
    return r;
}

int main(){
    //s="atatat$";
    //n=s.size();

    while (true) {
        cin >> N;

        if (feof(stdin)) break;

        cin >> words[0];
        len[0] = words[0].size() + 1;
        s = words[0];
        s += "$";
        for (int w = 1; w < N; w++) {
            cin >> words[w];
            len[w] = len[w-1] + words[w].size() + 1;
            s += words[w];
            s += "$";
        }

        n=s.size();
        suff_arr();
        lcp();

        memset(esta, 0, sizeof(esta));
        int start = 0, end = 1, b = 0;

        while (indexToWord(r[start]) == -1) start++;
        end = start+1;

        esta[indexToWord(r[start])] = true;
        while (start < end && end < n) {
            int ind = indexToWord(r[end]);
            //cout << r[end] << " -> " << ind << endl;
            while (esta[ind]) {
                //cout << "Recorte " << start << endl;
                esta[indexToWord(r[start])] = false;
                start++;
            }
            esta[ind] = true;
            if (end - start + 1 == N) b = max(b, calcularMin(start, end));
            //cout << "Agregue "<< end << " con size " << end-start+1 << endl;
            end++;
        }

        cout << b << endl;
    }


    /*for(int i=0;i<n;i++) {
        for (int j = r[i]; j < n; j++) {
            cout << s[j];
        }
        cout << endl;
    }
    for(int i=0;i<n;i++)cout<<h[i]<<" ";cout<<endl;*/

    return 0;
}
```

PI

```
double PI = acos(-1.0);
double PI = 4*atan(1.0);
```

DETERMINANTE

Codigo para el calculo del determinante de una matriz NxN

```
//Argumentos: 1) N - Dimension de la matriz cuadrada
//            2) A[i][j] para 0 <= i,j <= N

#include<stdio.h>
double matrix[10][10];
int determinant(int n){
  double ratio, det;
  int i, j, k;
  /* Conversion of matrix to upper triangular */
  for(i = 0; i < n; i++){
    for(j = 0; j < n; j++){
      if(j>i){
        ratio = matrix[j][i]/matrix[i][i];
        for(k = 0; k < n; k++){
          matrix[j][k] -= ratio * matrix[i][k];
        }
      }
    }
  }
  det = 1; //storage for determinant
  for(i = 0; i < n; i++)
    det *= matrix[i][i];

  return det;
}
```

MÉTODO DE SIMPSON

Fórmula: Integral de a hasta b de $f(x)dx = ((b-a)/6) * [f(a) + 4f((a+b)/2) + f(b)]$

En el caso de que el intervalo [a,b] no sea lo suficientemente pequeño, el error al calcular la integral puede ser muy grande. Para ello, se recurre a la fórmula compuesta de Simpson. Se divide el intervalo [a,b] en n subintervalos iguales (con n par), de manera que $x_i = a + ih$, donde $h = (b-a)/n$ para i = 0, 1, ..., n.

Aplicando la Regla de Simpson a cada subintervalo [x_{j-1},x_{j+1}], : j=1,3,5, ..., n-1, tenemos:

Integral de x_{j-1} hasta x_{j+1} de

$f(x)dx = (x_{j+1} - x_{j-1})/3 * [f(x_{j-1}) + 4f(x_j) + f(x_{j+1})]$.

Sumando las integrales de todos los subintervalos, llegamos a que:

Integral de a hasta b de

$f(x)dx = (h/3)[f(x_0) + 2 * sum_{j=1}^{n/2-1}(f(x_{2j})) + 4 * sum_{j=1}^{n/2}(f(x_{2j-1}) + f(x_n))]$

NÚMEROS DE MOTSKIN

Formula: - Recursiva:

$M_(n+1) = M_n + sum[i=0; n-1](M_i * M_(n-1-i)) = (2n+3/n+3)M_n + (3n/n+3)M_(n-1)$ - Coeficientes binomiales y Números de Catalan:

$sum[i=0; floor(n/2)](n, 2k)C_k$

Aplicaciones: - Número de maneras diferentes de dibujar cuerdas que no se intersecten enntre n puntos en un circulo. - El número de secuencias de enteros positivos de longitud n-1 en donde el elemento inicial y final sean 1 o 2or 2, y que la diferencia entre dos elementos consecutivos sea -1, 0 o 1. - Teniendo el cuadrante superior derecho de un plano, el número Motzkin de n da el número de rutas desde la coordena (0,0) hasta la coordenada(n, 0) en n pasos si se permite sólo moverse a la derecha (arriba,abajo, en frente) pero se prohibe que se baje del eje y=0.

COMBINATORIO

Calcula el combinatorio de n en k

```
long long comb[2009][1009];
long long c = 1000000007;

long long combdp(int n, int k){
  if(k == 0 || n == k) return 1;
  if(comb[n][k] != -1) return comb[n][k];
  if(comb[n][n-k] != -1) return comb[n][n-k];
  if(k == 1 || n-1 == k) return n;
  return comb[n][k] = ((combdp(n-1, k-1)%c + combdp(n-1, k)%c)%c+c) % c;
}
```

NÚMEROS DE CATALÁN

Cn = (2n, n) - (2n, n-1) Aplicaciones:

Numero de palabras de Dyck de longitud 2n. Una palabra de Dyck es una cadena de caracteres que consiste en n Xs y n Ys de forma que no haya ningún segmento inicial que tenga más Ys que Xs

Reinterpretando el símbolo X como un paréntesis abierto y la Y como un paréntesis cerrado, Cn cuenta el número de expresiones que contienen n pares de paréntesis correctamente colocados

MAXIMO COMUN DIVISOR

Te calcula el máximo común divisor entre dos enteros.

```
long long gcd(long long a, long long b){
  if(b == 0) return a;
  return gcd(b, a%b);
}
```

MAXIMO COMUN DIVISOR EXTENDIDO

Resuelve la siguiente ecuación: ax + by = gcd(a,b) donde: r1 = x, r2 = y

```
long long extended_gcd(long long a, long long b){ //, long long &r1, long long &r2
  long long x = 0, lastx = 1;
  long long y = 1, lasty = 0;
  while (b != 0){
    long long quotient = a / b;
    c(b, a % b, a, b);
    c(lastx - quotient*x, x, x, lastx);
    c(lasty - quotient*y, y, y, lasty);
  }
  // r1 = lastx; r2 = lasty;
  return lastx;
}
```

INVERSO MULTIPLICATIVO

Calcula el inverso multiplicativo de a módulo m. Cuidado, gcd(a,m) debe ser 1.

```
typedef long long lint;
inline lint mod_mult_inverse(lint a, lint m){

    lint x = 0, lastx = 1;
    lint y = 1, lasty = 0;

    lint b = m;
    lint aux;
    while( b!=0 ){
        lint quotient = a/b;
        lint aux;
        //{a, b} = {b, a bod b}
        aux = b;
        b = a%b;
        a = aux;
        //{x, lastx} = {lastx - quotient*x, x}
        aux = x;
        x = lastx - quotient*x;
        lastx = aux;
        //{y, lasty} = {lasty - quotient*y, y}
        aux = y;
        y = lasty - quotient*y;
        lasty = aux;
    }
    if(a!=1)
        return 0;
    x = lastx;
    if( x < 0 )
        return x + m;
    return x;
}
```

FERMAT LITTLE THEOREM

```
x^(p - 1) = 1 (mod p)
x^(p - 1) = x * x^(p - 2) = 1(mod p)
Conclusion:
a / b (mod p) = a * fastpow(b, p - 2, p) % p
```

GAUSS JORDAN

(standford)

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

#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
const double EPS = 1e-10;
typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

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

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

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

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

  return det;
}

int main() {
  const int n = 4;
  const int m = 2;
  double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
  double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
  VVT a(n), b(n);
  for (int i = 0; i < n; i++) {
    a[i] = VT(A[i], A[i] + n);
    b[i] = VT(B[i], B[i] + m);
  }
  double det = GaussJordan(a, b);
  // expected: 60
  cout << "Determinant: " << det << endl;
  // expected: -0.233333 0.166667 0.133333 0.0666667
  //           0.166667 0.166667 0.333333 -0.333333
  //           0.233333 0.833333 -0.133333 -0.0666667
  //           0.05 -0.75 -0.1 0.2
  cout << "Inverse: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }
  // expected: 1.63333 1.3
  //           -0.166667 0.5
  //           2.36667 1.7
  //           -1.85 -1.35
  cout << "Solution: " << endl;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++)
      cout << b[i][j] << ' ';
    cout << endl;
  }
}
```

TEOREMA CHINO DEL RESTO

Dado a[i] enteros y b[i] modulos resuelve el mínimo común múltiplo entre ecuaciones modulares. Se deben crear y[i] y z[i] arreglos auxiliares. n es la cantidad de ecuaciones y m es la multiplicación de todos los b[i]. Para usar este algoritmo se debe cumplir que gcd(b[i], b[j]) es 1.

```
void c(long long a, long long b, long long &r1, long long &r2){
  r1 = a;
  r2 = b;
}

long long CRT(){
  for(int i = 0; i < n; i++){
    z[i] = m / b[i];
    y[i] = ((extended_gcd(z[i], b[i]) % b[i]) + b[i] )%b[i];
  }

  long long res = 0;
  for(int i = 0; i < n; i++){
    res = (res + (a[i]*y[i]*z[i]))%m;
  }

  res = (((res)%m) +m) %m;
  return res;
}
```

TEOREMA CHINO DEL RESTO EXTENDIDO

Dados a1 n1 a2 y n2 retorna True si existe una solución a las ecuaciones dando como resultado a0 y n0. Retorna False en caso contrario.

```
bool CRT(long long a1, long long n1, long long a2, long long n2, long long &a0, long long &n0){
  long long gcdn1n2 = gcd(n1, n2);
  long long lcmn1n2 = n1*n2/gcdn1n2;
  if(a1 % gcdn1n2 != a2 % gcdn1n2) return false;
  long long x, y;
  extended_gcd(n1, n2, x, y);
  long long q1, q2, r;
  r = a1 %  gcdn1n2;
  q1 = a1 / gcdn1n2;
  q2 = a2 / gcdn1n2;
  long long res = q2*x*n1 + q1*y*n2 + r;
  res = ((res % lcmn1n2) + lcmn1n2)%lcmn1n2;
  a0 = res;
  n0 = lcmn1n2;
  return true;
}
```

Transformada rápida de Fourier*

```
struct cpx{
  cpx(){}
  cpx(double aa):a(aa){ b = 0;}
  cpx(double aa, double bb):a(aa),b(bb){}
  double a;
  double b;
  double modsq(void) const {
    return a * a + b * b;
  }
  cpx bar(void) const{
    return cpx(a, -b);
  }
};

cpx operator +(cpx a, cpx b){
  return cpx(a.a + b.a, a.b + b.b);
}

cpx operator *(cpx a, cpx b){
  return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}

cpx operator /(cpx a, cpx b){
  cpx r = a * b.bar();
  return cpx(r.a / b.modsq(), r.b / b.modsq());
}

cpx EXP(double theta){
  return cpx(cos(theta),sin(theta));
}

const double two_pi = 4 * acos(0);

// in:     input array
// out:    output array
// step:   {SET TO 1} (used internally)
// size:   length of the input/output {MUST BE A POWER OF 2}
// dir:    either plus or minus one (direction of the FFT)
// RESULT: out[k] = \sum_{j=0}^{size - 1} in[j] * exp(dir * 2pi * i * j * k / size)
void FFT(cpx *in, cpx *out, int step, int size, int dir){
  if(size < 1) return;
  if(size == 1)
  {
    out[0] = in[0];
    return;
  }
  FFT(in, out, step * 2, size / 2, dir);
  FFT(in + step, out + size / 2, step * 2, size / 2, dir);
  for(int i = 0 ; i < size / 2 ; i++)
  {
    cpx even = out[i];
    cpx odd = out[i + size / 2];
    out[i] = even + EXP(dir * two_pi * i / size) * odd;
    out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) / size) * odd;
  }
}

// Usage:
// f[0...N-1] and g[0..N-1] are numbers
// Want to compute the convolution h, defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly, define G and H.
// The convolution theorem says H[n] = F[n]G[n] (element-wise product).
// To compute h[] in O(N log N) time, do the following:
//   1. Compute F and G (pass dir = 1 as the argument).
//   2. Get H by element-wise multiplying F and G.
//   3. Get h by taking the inverse FFT (use dir = -1 as the argument)
//      and *dividing by N*. DO NOT FORGET THIS SCALING FACTOR.

cpx a[8] = {4, 3, 2 ,1,0,0,0,0};
cpx A[8];
void use(){
  FFT(a, A, 1, 8, 1);
}
```

Suma de digitos base m*

```
lint digitsum(lint r, lint m) {
  // sum from 0 to r - digits base m
  lint ret = 0;
  for (lint x = 1; x <= r; x *= m) {
    lint dig = (r / x) % m;
    lint le = (r / x) / m;
    lint ri = r % x;
    lint cur = 0;
    for (int i = 1; i < dig; ++i) cur += i;
    ret += cur * (le + 1) * x;
    ret += dig * (le * x + ri + 1);
    cur = 0;
    for (int i = dig + 1; i < m; ++i) cur += i;
    ret += cur * le * x;
  }
  return ret;
}
```

EXPONENCIACIÓN RÁPIDA

Resuelve (a ** b)

```
// sum a**i = (a**(n+1) -1) / (a-1)
long long exp(long long a, long long b, long long m){
  long long result = 1;
  long long power = a;
  while(b>0) {
    if (b%2==1) result = (result*power)%m;
    power = (power*power)%m;
    b = b/2;
  }
  return result;
}
```

TEST DE MILLER

Verifica si un número es primo. Colocarle 20 iteraciones.

```
long long mul(long long a,long long b,long long c){
    long long 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;
}

long long mod(long long a,long long b,long long c){
    long long x=1,y=a;
    while(b > 0){
        if(b%2 == 1){
            x=mul(x,y,c);
        }
        y = mul(y,y,c);
        b /= 2;
    }
    return x%c;
}

bool Miller(long long p,long long it){
    if(p<2){
        return false;
    }
    if(p!=2 && p%2==0){
        return false;
    }
    long long s=p-1;
    while(s%2==0){
        s/=2;
    }
    for(long long i=0;i<it;i++){
        long long a=rand()%(p-1)+1,tt=s;
        long long md=mod(a,tt,p);
        while(tt!=p-1 && md!=1 && md!=p-1){
            md=mul(md,md,p);
            tt *= 2;
        }
        if(md!=p-1 && tt%2==0){
            return false;
        }
    }
    return true;
}
```

SIEVE ERATOSTELES

```
#define SIZE 10009
int sieve[SIZE], primes[SIZE], prime_size;
vector<int> primos;
int initsieve(){
  int a, b;
  prime_size = 0;
  memset(sieve, 0, sizeof(sieve));
  sieve[0] = sieve[1] = 1;
  for(a = 2 ; a < SIZE ; a++){
    if(!sieve[a]){
      primos.push_back(a);
      primes[prime_size++] = a;
      for (b = a+a ; b < SIZE ; b += a){
        sieve[b] = 1;
      }
    }
  }
}
```

Point 2D

```
#include <iostream>
#include <cstdio>
#include <cmath>

/* Point 2d {{{ */
typedef double pdata;
struct p2d {
        pdata x, y;
        p2d operator+(p2d p){ return p2d(x + p.x, y + p.y); } //Suma vectores
        p2d operator-(p2d p){ return p2d(x - p.x, y - p.y); } //Resta vectores
        p2d operator*(pdata k){ return p2d(k * x, k * y); } //Escalar un vector, recibe un double
        p2d operator/(pdata k){ return p2d(x / k, y / k); } //Reducir un vector
        pdata  operator*(p2d p){ return (x * p.y - y * p.x);} //Producto cruz, recibe un point
        pdata operator^(p2d p){ return (x * p.x + y * p.y);} //Producto punto
        bool operator==(p2d p){ return (x == p.x && y == p.y);} //Igualdad puntos
        bool operator<(p2d p) const { return (x < p.x || (x == p.x && y < p.y));} //Operador <
        bool operator>(p2d p) const { return (x > p.x || (x == p.x && y > p.y));} //Operador >
        bool point_on_line(p2d p0, p2d p1){ return ((p1 - p0) * (*this - p0)) == 0;} //Pto en la linea,
        pdata sqr(pdata x){ return (x * x);} //Cuadrado
        double dist(p2d p){ return hypot(x - p.x, y - p.y); } //Distancia Entre dos puntos
        double dist2(p2d p){ return sqr(x - p.x) + sqr(y - p.y);}
        double mod(){ return sqrt(x * x + y * y); } //Modulo Vector
        double mod2(){ return (x * x + y * y); } //Modulo al cuadrado
        double ang(p2d p){ return acos((*this ^ p)/((*this).mod() * p.mod())); } //Angulo entre dos vec

        /*Funcion que calcula la distancia de un punto a una linea
                Argumentos: p0,p1 - Puntos que definen la linea
                  this - Punto a calcular la distancia */
        double point_line_distance(p2d p0, p2d p1){ //Distancia de un punto a una linea
                p2d v1 = *this - p0, v2 = p1 - p0;
                double u = (v1 ^ v2)/v2.mod2();
                p2d p = p0 + v2 * u;
                return (*this).dist(p);
        }

        /*Funcion que calcula la distancia de un punto a un segmento
                Argumentos: p0,p1 - Puntos que definen el segmento
                  this - Punto a calcular la distancia */
        double point_line_segment_distance(p2d p0, p2d p1){
                p2d v1 = *this - p0, v2 = p1 - p0;
                double u = (v1 ^ v2)/v2.mod2();
                if(u < 0) return (*this).dist(p0);
                if(u > 1) return (*this).dist(p1);
                return (*this).dist(p0 + v2 * u);
        }
        p2d(pdata _x = 0, pdata _y = 0): x(_x), y(_y) {};
};
/* }}} */
```

Point 2D extensiones

```
#include <iostream>
#include <cstdio>
#include <cmath>

/* Point 2d {{{ */
typedef double pdata;
struct p2d {
        pdata x, y;
        p2d operator-(p2d p){ return p2d(x - p.x, y - p.y); } //Resta vectores
        pdata  operator*(p2d p){ return (x * p.y - y * p.x);} //Producto cruz, recibe un point
        bool operator==(p2d p){ return (x == p.x && y == p.y);} //Igualdad puntos
        bool point_on_line(p2d p0, p2d p1){ return ((p1 - p0) * (*this - p0)) == 0;} //Pto en la linea,
        double dist(p2d p){ return hypot(x - p.x, y - p.y); } //Distancia Entre dos puntos

        p2d(pdata _x = 0, pdata _y = 0): x(_x), y(_y) {};
};
/* }}} */

/* Line 2d {{{ */
struct line {
        pdata A, B, C;
        line(p2d X, p2d Y): A(Y.y - X.y), B(X.x - Y.x), C((this->A)*X.x + (this->B)*X.y) {};
};
/* }}} */

//Retorna la interseccion, en casos de ser linea paralelas retornara un punto con (inf,inf)
p2d line_line_intersect(line L1, line L2) {
        p2d R;

        double det = L1.A*L2.B - L2.A*L1.B;
    if(det == 0){
        //Lineas paralelas
        R.x = 0x3f3f3f;
        R.y = 0x3f3f3f;
    }else{
        R.x = (L2.B*L1.C - L1.B*L2.C)/det; //X
        R.y = (L1.A*L2.C - L2.A*L1.C)/det; //Y
    }

    return R;
}

//Observacion: Para el caso de inters. de segmentos, basta ver si el pto de la interseccion basta con q


// Recibe 3 puntos y retorna el centro del circulo, el radio se saca con la distacia del centro a cualq
// de los 3 ptos.
p2d circle_3_points( p2d X, p2d Y, p2d Z ) {
        line L1 = line(X,Y);
        line L2 = line(Y,Z);
        p2d MedioXY = p2d((X.x + Y.x)/2, (X.y + Y.y)/2);
        p2d MedioYZ = p2d((Y.x + Z.x)/2, (Y.y + Z.y)/2);
        double D1 = (-L1.B)*MedioXY.x + L1.A*MedioXY.y;
        double D2 = (-L2.B)*MedioYZ.x + L2.A*MedioYZ.y;
        double A;
        A = L1.A;
        L1.A = -L1.B;
        L1.B = A;
        L1.C = D1;
        A = L2.A;
        L2.A = -L2.B;
        L2.B = A;
        L2.C = D2;
        return line_line_intersect(L1,L2);
}
```

Point in Polygon (Winding Number)

```
struct Point {
  public:
        int x, y;
  Point (int _x, int _y){
    x = _x;
    y = _y;
  }
  Point(){
    x = 0;
    y = 0;
  }
        bool operator <(const Point &p) const {
                return x < p.x || (x == p.x && y < p.y);
        }
        bool operator ==(const Point &p) const {
                return (x == p.x && y == p.y);
        }

};

// isLeft(): tests if a point is Left|On|Right of an infinite line.
//    Return: >0 for P2 left of the line through P0 and P1
//            =0 for P2  on the line
//            <0 for P2  right of the line

inline int isLeft( Point P0, Point P1, Point P2 )
{
    return ( (P1.x - P0.x) * (P2.y - P0.y)
            - (P2.x -  P0.x) * (P1.y - P0.y) );
}


// cn_PnPoly(): crossing number test
//      Return:  0 = outside, 1 = inside

int cn_PnPoly( Point P, Point* V, int n )
{
    int    cn = 0;    // the  crossing number counter

    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {    // edge from V[i]  to V[i+1]
       if (((V[i].y <= P.y) && (V[i+1].y > P.y))     // an upward crossing
        || ((V[i].y > P.y) && (V[i+1].y <=  P.y))) { // a downward crossing
            // compute  the actual edge-ray intersect x-coordinate
            float vt = (float)(P.y  - V[i].y) / (V[i+1].y - V[i].y);
            if (P.x <  V[i].x + vt * (V[i+1].x - V[i].x)) // P.x < intersect
                 ++cn;   // a valid crossing of y=P.y right of P.x
        }
    }
    return (cn&1);    // 0 if even (out), and 1 if  odd (in)

}


// wn_PnPoly(): winding number test
//      Return:  wn = the winding number (=0 only when P is outside)
int wn_PnPoly( Point P, Point* V, int n )
{
    int    wn = 0;    // the  winding number counter

    // loop through all edges of the polygon
    for (int i=0; i<n; i++) {   // edge from V[i] to  V[i+1]
        if (V[i].y <= P.y) {          // start y <= P.y
            if (V[i+1].y  > P.y)      // an upward crossing
                 if (isLeft( V[i], V[i+1], P) > 0)  // P left of  edge
                     ++wn;            // have  a valid up intersect
        }
        else {                        // start y > P.y (no test needed)
            if (V[i+1].y  <= P.y)     // a downward crossing
                 if (isLeft( V[i], V[i+1], P) < 0)  // P right of  edge
                     --wn;            // have  a valid down intersect
        }
    }
    return wn;
}

//Si el winding number es 0 -> no esta en el poligono
```

Centroide

```
#include<cstdio>
#include<cmath>
#include<iostream>

struct point {
        double x, y;
};


int N; //Tamano arreglo de puntos
point P[N]; //Arreglo de puntos, que le sobre un espacio!!!!

using namespace std;

point centroid(point P[])
{
        point result;
        int T, N, i;
        double d, A, resX, resY;
        P[N] = P[0], resX = resY = A = 0.0;
        for(i=0; i<N; i++)
        {
                d = P[i].x * P[i+1].y - P[i+1].x * P[i].y;
                A += d;
                resX += (P[i].x + P[i+1].x) * d;
                resY += (P[i].y + P[i+1].y) * d;
        }
        resX /= (3.0 * A);
        resY /= (3.0 * A);
        result.x = resX;
        result.y = resY;
        return result;
}
```

Área Polígono

Retorna el área del polígono con puntos p.

```
struct point {
        double x, y;
};


int N; //Tamano arreglo de puntos
point p[N]; //Arreglo de puntos, que le sobre un espacio!!!!


double calc_area(point p[]) {

        int area = 0;
        int cross = 0;
        //We will triangulate the polygon
        //into triangles with points p[0],p[i],p[i+1]

        for(int i = 1; i+1<N; i++){
                   int x1 = p[i].x - p[0].x;
            int y1 = p[i].y - p[0].y;
            int x2 = p[i+1].x - p[0].x;
            int y2 = p[i+1].y - p[0].y;
            cross = x1*y2 - x2*y1;
            area += cross;
        }

        return abs(area/2.0);
}
```

GEOMETRIA

FUNCIONES GEOMETRICAS (STANDFORD)

```
// C++ routines for computational geometry.
#include <iostream> <vector> <cmath> <cassert>
using namespace std;
double INF = 1e100;
double EPS = 1e-12;

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

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

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

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

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

// compute distance from c to segment between a and b
```

```
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

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

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

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

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

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

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

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

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

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

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

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

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

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

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```

CONVEX HULL MONOTONE CHAIN

Para que trabaje con puntos colineales basta con colocar que el producto cruz es menor a cero. Cuidado: Repite el primer punto al final.

```
typedef int coord_t;         // coordinate type
typedef long long coord2_t;  // must be big enough to hold 2*max(|coordinate|)^2
struct Point {
  public:
        int x, y;
  Point (int _x, int _y){
    x = _x;
    y = _y;
  }
  Point(){
    x = 0;
    y = 0;
  }
        bool operator <(const Point &p) const {
                return x < p.x || (x == p.x && y < p.y);
        }
        bool operator ==(const Point &p) const {
                return (x == p.x && y == p.y);
        }

};

// 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
// Returns a positive value, if OAB makes a counter-clockwise turn,
// negative for clockwise turn, and zero if the points are collinear.
coord2_t cross(const Point &O, const Point &A, const Point &B){
        return (A.x - O.x) * (coord2_t)(B.y - O.y) - (A.y - O.y) * (coord2_t)(B.x - O.x);
}

vector<Point> convex_hull(vector<Point> P){
        int n = P.size(), k = 0;
        vector<Point> H(2*n);
        // Sort points lexicographically
        sort(P.begin(), P.end());
        // Build lower hull
        for (int i = 0; i < n; i++) {
                while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
                H[k++] = P[i];
        }
        // Build upper hull
        for (int i = n-2, t = k+1; i >= 0; i--) {
                while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
                H[k++] = P[i];
        }
        H.resize(k);
        return H;
}
```

MIN MAX AREA OF TRIANGLES*

Dado un vector de puntos, devuelve la mínima y máxima area de los triángulos.

```
struct Point {
  public:
        int x, y;
  Point (int _x, int _y){
    x = _x;
    y = _y;
  }
  Point(){
    x = 0;
    y = 0;
  }
};

int n, idx[MAXN], inv[MAXN];
vector<Point> p;

class Event {
  public:
  int i, j, dx, dy, right, left;

  Event(int _i, int _j) {
    i = _i;
    j = _j;
    dx = p[i].x - p[j].x;
    dy = p[i].y - p[j].y;
    if (dy == 0 && dx < 0)
      dx = -dx;
    else if (dy < 0) {
      dx = -dx;
      dy = -dy;
    }
    left = min(p[i].x, p[j].x);
    right = max(p[i].x, p[j].x);
  }

  bool operator<(const Event &o) const {
    int temp = dy * o.dx - dx * o.dy;
    if (temp == 0) temp = left - o.left;
    if (temp == 0) temp = right - o.right;
    return (temp < 0);
  }
};

vector<Event> e;

int twiceArea(Point a, Point b, Point c) {
  return fabs((a.x - c.x) * (b.y - c.y) - (a.y - c.y) * (b.x - c.x));
}
```

```
bool compare(int f, int g) {
  int dy = p[f].y - p[g].y;
  int r = (dy == 0 ? (p[f].x - p[g].x) : dy);
  return (r < 0);
}

// Return the min area and the max area
pair<double,double> solve() {
  e.clear();
  for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++)
      e.pb(Event(i, j));

  sort(e.begin(), e.end());

  for (int i = 0; i < n; i++)
    idx[i] = i;

  sort(idx, idx+n, compare);

  for (int i = 0; i < n; i++)
    inv[idx[i]] = i;

  int bestmin = inf, bestmax = -1, k;

  for (int i = 0; i < e.size(); i++) {
    k = min(inv[e[i].i], inv[e[i].j]) - 1;
    if (k > -1) {
      bestmin = min(twiceArea(p[e[i].i], p[e[i].j], p[idx[k]]), bestmin);
      bestmax = max(twiceArea(p[e[i].i], p[e[i].j], p[idx[0]]), bestmax);
    }
    k = max(inv[e[i].i], inv[e[i].j]) + 1;
    if (k < n) {
      bestmin = min(twiceArea(p[e[i].i], p[e[i].j], p[idx[k]]), bestmin);
      bestmax = max(twiceArea(p[e[i].i], p[e[i].j], p[idx[n - 1]]), bestmax);
    }
    k = idx[inv[e[i].i]];
    idx[inv[e[i].i]] = idx[inv[e[i].j]];
    idx[inv[e[i].j]] = k;
    k = inv[e[i].i];
    inv[e[i].i] = inv[e[i].j];
    inv[e[i].j] = k;
  }
  return mp(bestmin/2.0, bestmax/2.0);
}

void init(vector<Point> ve){
  p = ve;
}
```