Índice

1. Teoría de números

1.1. Big mod

```
//retorna (b^p)mod(m)
// 0 <= b,p <= 2147483647
// 1 <= m <= 46340
long f(long b, long p, long m){
  long mask = 1;
  long pow2 = b % m;
  long r = 1;

  while (mask){
    if (p & mask)
      r = (r * pow2) % m;
    pow2 = (pow2*pow2) % m;
    mask <<= 1;
  }
  return r;
}
```

1.2. Criba de Eratóstenes

Marca los números primos en un arreglo. Algunos tiempos de ejecución:

SIZE	Tiempo (s)
100000	0.004
1000000	0.078
10000000	1.550
100000000	14.319

```cpp
#include <iostream>

const int SIZE = 1000000;

//criba[i] = false si i es primo
bool criba[SIZE+1];

void buildCriba(){
  memset(criba, false, sizeof(criba));

  criba[0] = criba[1] = true;
  for (int i=2; i<=SIZE; i += 2){
    criba[i] = true;
  }

  for (int i=3; i<=SIZE; i += 2){
    if (!criba[i]){
      for (int j=i+i; j<=SIZE; j += i){
        criba[j] = true;
      }
    }
  }
}
```

1.3. Divisores de un número

Este algoritmo imprime todos los divisores de un número (en desorden) en $O(\sqrt{n})$. Hasta 4294967295 (máximo *unsigned long*) responde instantaneamente. Se puede forzar un poco más usando *unsigned long long* pero más allá de 10^{12} empieza a responder muy lento.

```cpp
 for (int i=1; i*i<=n; i++) {
  if (n%i == 0) {
    cout << i << endl;
    if (i*i<n) cout << (n/i) << endl;
  }
}
```

2. Grafos

2.1. Algoritmo de Dijkstra

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <cmath>
#include <sstream>
#include <functional>
```

```
using namespace std;

const double infinity = 1E20;

struct point{
  double x, y;
  point(double X, double Y){ x = X; y = Y;}
};

map< point, double > dist;

bool operator ==(const point &a, const point &b){ return (a.x ==
b.x && a.y == b.y);}
bool operator !=(const point &a, const point &b){ return !(a ==
b);}
bool operator <(const point &a, const point &b){ return (a.x <
b.x || (a.x == b.x && a.y < b.y));}
double distancia(point a, point b){return hypot(a.y-b.y,
a.x-b.x);}


struct heapCompare : public binary_function<point, point, bool>
{
  bool operator()(const point &x, const point &y) const
  { return dist[x] > dist[y]; }
};


struct grafo{
  //contiene todos los nodos sueltos
  vector<point> nodos;
  //contiene un vector con todos los vecinos para el punto point
  map< point, vector<point> > vecinos;

  void insert(point a){
    if (vecinos.count(a) == 1) return; //Ya insertamos este nodo
    nodos.push_back(a);
    vector<point> v;
    vecinos.insert(make_pair(a, v));
  }

  void make_vecinos(double maxPath){
    for (map< point, vector<point> >::iterator
it=vecinos.begin(); it!=vecinos.end(); ++it){
      if (distancia((*it).first, point(0.00, 0.00)) > maxPath){
        continue;
      }
      for (map< point, vector<point> >::iterator jt = it;
jt!=vecinos.end(); ++jt){
        if ((*it).first != (*jt).first){
```

```
            if ((*jt).first.x - (*it).first.x > 1.5){
              break;
            }
            vector<point> adj = vecinos[(*it).first];
            if (distancia((*jt).first, (*it).first) <= 1.5){
              vecinos[(*it).first].push_back((*jt).first);
              vecinos[(*jt).first].push_back((*it).first);
            }
          }
        }
      }
    }

    void initialize(){
      dist.clear();
      for (int i=0; i<nodos.size(); ++i){
        dist[nodos[i]] = infinity;
        if (nodos[i].x == 0.00 && nodos[i].y == 0.00){
          dist[nodos[i]] = 0.00;
        }
      }
    }

    void dijkstra(const double &maxPath, const point &finalPoint){
      initialize();

      priority_queue<point, vector<point>, heapCompare > q;
      q.push(point(0.0, 0.0));
      while (!q.empty()){
        point u = q.top();
        q.pop();
        if (distancia(point(0.00, 0.00), u) + distancia(u,
finalPoint) <= maxPath){
          for (int i=0; i<vecinos[u].size(); ++i){
            point v = vecinos[u][i];
            if (dist[vecinos[u][i]] > (dist[u] + distancia(u,v))){
              dist[vecinos[u][i]] = dist[u] + distancia(u, v);
              q.push(v);
            }
          }
        }
      }
    }

};



int main(){
  while (true){
```

```
    string s;
    for (s = ""; s == ""; getline(cin, s));
    if (s == "*") break;


    grafo g;

    stringstream line;
    line << s;

    int w,h;
    line >> w >> h;
    g.insert(point((double)w, (double)h));
    g.insert(point(0.00, 0.00));
    int noPuntos;
    cin >> noPuntos;
    for (int i=0; i<noPuntos; ++i){
      double x,y;
      cin >> x >> y;
      g.insert(point(x,y));
    }


    double maximoCamino;
    cin >> maximoCamino;

    g.make_vecinos(maximoCamino);

    g.dijkstra(maximoCamino, point((double)w, (double)h));
    if (dist[point((double)w, (double)h)] <= maximoCamino){
      printf("I am lucky!\n");
    }else{
      printf("Boom!\n");
    }

  }
  return 0;
}
```

2.2. Algoritmo de Prim

```
#include <iostream>
#include <cmath>
#include <map>
#include <queue>
#include <set>

using namespace std;
```

```
typedef pair<double, double> point;
//Gives a vector of adjacent nodes to a point
typedef map< point, vector<point> > graph;
//Edge of length "first" that arrives to point "second"
typedef pair<double, point> edge;

double euclidean(const point &a, const point &b){ return
hypot(a.first-b.first, a.second-b.second);}


int main(){
  int casos;
  cin >> casos;
  while (casos--){
    graph g;
    int n;
    cin >> n;
    while (n--){
      double x,y;
      cin >> x >> y;
      point p(x,y);
      if (g.count(p) == 0){ //Si no está todavía
        vector<point> v;
        g[p] = v;
        for (graph::iterator i = g.begin(); i != g.end(); ++i){
          if ((*i).first != p){
            (*i).second.push_back(p);
            g[p].push_back((*i).first);
          }
        }
      }
    }

    set<point> visited;
    priority_queue<edge, vector<edge>, greater<edge> > q;
    //Each edge in q has got a length "first" and a point
"second".
    //It means I can reach point "second" which is "first" meters
away.
    //q has the closest reachable node on top (I may have already
visited it!)
    q.push(edge(0.0, (*g.begin()).first));
    double totalDistance = 0.0;
    while (!q.empty()){
      edge nearest = q.top();
      q.pop();
      point actualNode = nearest.second;
      if (visited.count(actualNode) == 1) continue; //Ya habia
visitado este
      totalDistance += nearest.first;
```

```
      visited.insert(actualNode);
      vector<point> neighbors = g[actualNode];
      for (int i=0; i<neighbors.size(); ++i){
        point t = neighbors[i];
        double dist = euclidean(actualNode, t);
        q.push(edge(dist, t));
      }
    }
    printf("%.2f\n", totalDistance);
    if (casos > 0) cout << endl; //Endl between cases
  }
}
```

2.3. Algoritmo de Floyd

```
#include <iostream>
#include <climits>
#include <algorithm>

using namespace std;

unsigned long long g[101][101];

int main(){
  int casos;
  cin >> casos;
  bool first = true;
  while (casos--){
    if (!first) cout << endl;
    first = false;

    int n, e, t;
    cin >> n >> e >> t;
    for (int i=0; i<n; ++i){
      for (int j=0; j<n; ++j){
        g[i][j] = INT_MAX;
      }
      g[i][i] = 0;
    }

    int m;
    cin >> m;
    while (m--){
      int i, j, k;
      cin >> i >> j >> k;
      g[i-1][j-1] = k;
    }

    for (int k=0; k<n; ++k){
      for (int i=0; i<n; ++i){
```

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

    int r=0;
    e -= 1;
    for (int i=0; i<n; ++i){
      r += ((g[i][e] <= t) ? 1 : 0);
      /*      if (g[i][e] <= t){
        cout << "Adding " << i+1 << " - distancia: " <<
g[i][e] << endl;
        }*/
    }

    cout << r << endl;
  }
  return 0;
}
```

2.4. Puntos de articulación

```
#include <vector>
#include <set>
#include <map>
#include <algorithm>
#include <iostream>
#include <iterator>

using namespace std;

typedef string node;
typedef map<node, vector<node> > graph;
typedef char color;

const color WHITE = 0, GRAY = 1, BLACK = 2;

graph g;
map<node, color> colors;
map<node, int> d, low;

set<node> cameras;

int timeCount;

void dfs(node v, bool isRoot = true){
  colors[v] = GRAY;
  d[v] = low[v] = ++timeCount;
  vector<node> neighbors = g[v];
```

```
  int count = 0;
  for (int i=0; i<neighbors.size(); ++i){
    if (colors[neighbors[i]] == WHITE){ //  (v, neighbors[i]) is
a tree edge
      dfs(neighbors[i], false);
      if (!isRoot && low[neighbors[i]] >= d[v]){
        cameras.insert(v);
      }
      low[v] = min(low[v], low[neighbors[i]]);
      ++count;
    }else{ // (v, neighbors[i]) is a back edge
      low[v] = min(low[v], d[neighbors[i]]);
    }
  }
  if (isRoot && count > 1){ //Is root and has two neighbors in
the DFS-tree
    cameras.insert(v);
  }
  colors[v] = BLACK;
}

int main(){
  int n;
  int map = 1;
  while (cin >> n && n > 0){
    if (map > 1) cout << endl;
    g.clear();
    colors.clear();
    d.clear();
    low.clear();
    timeCount = 0;
    while (n--){
      node v;
      cin >> v;
      colors[v] = WHITE;
      g[v] = vector<node>();
    }

    cin >> n;
    while (n--){
      node v,u;
      cin >> v >> u;
      g[v].push_back(u);
      g[u].push_back(v);
    }

    cameras.clear();

    for (graph::iterator i = g.begin(); i != g.end(); ++i){
      if (colors[(*i).first] == WHITE){
```

```
        dfs((*i).first);
      }
    }

    cout << "City map #"<<map<<": " << cameras.size() << "
camera(s) found" << endl;
    copy(cameras.begin(), cameras.end(),
ostream_iterator<node>(cout,"\n"));
    ++map;
  }
  return 0;
}
```

3. Programación dinámica

3.1. Longest common subsequence

```
#define MAX(a,b) ((a>b)?(a):(b))
int dp[1001][1001];

int lcs(const string &s, const string &t){
  int m = s.size(), n = t.size();
  if (m == 0 || n == 0) return 0;
  for (int i=0; i<=m; ++i)
    dp[i][0] = 0;
  for (int j=1; j<=n; ++j)
    dp[0][j] = 0;
  for (int i=0; i<m; ++i)
    for (int j=0; j<n; ++j)
      if (s[i] == t[j])
        dp[i+1][j+1] = dp[i][j]+1;
      else
        dp[i+1][j+1] = MAX(dp[i+1][j], dp[i][j+1]);
  return dp[m][n];
}
```

4. Geometría

4.1. Área de un polígono

Si P es un polígono simple (no se intersecta a sí mismo) su área está dada por:

$$A(P) = \frac{1}{2}\sum_{i=0}^{n-1}(x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

4.2. Convex hull: Graham Scan

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

```cpp
#include <iterator>
#include <cmath>

using namespace std;

struct point{
  int x,y;
  point() {}
  point(int X, int Y) : x(X), y(Y) {}
};

point pivot;

ostream& operator<< (ostream& out, const point& c)
{
  out << "(" << c.x << "," << c.y << ")";
  return out;
}

//P es un polígono ordenado anticlockwise.
//Si es clockwise, retorna el area negativa.
//Si no esta ordenado retorna pura mierda
double area(const vector<point> &p){
  double r = 0.0;
  for (int i=0; i<p.size(); ++i){
    int j = (i+1) % p.size();
    r += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return r/2.0;
}

//retorna si c esta a la izquierda de el segmento AB
inline int cross(const point &a, const point &b, const point &c){
  return (b.x-a.x)*(c.y-a.y) - (c.x-a.x)*(b.y-a.y);
}

//Self < that si esta a la derecha del segmento Pivot-That
bool angleCmp(const point &self, const point &that){
  return( cross(pivot, that, self) < 0 );
}

inline int distsqr(const point &a, const point &b){
  return (a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y);
}

//vector p tiene los puntos ordenados anticlockwise
vector<point> graham(vector<point> p){
  pivot = p[0];
  sort(p.begin(), p.end(), angleCmp);
  //Ordenar por ángulo y eliminar repetidos.
```

```
  //Si varios puntos tienen el mismo angulo se borran todos
excepto el que esté más lejos
  for (int i=1; i<p.size()-1; ++i){
    if (cross(p[0], p[i], p[i+1]) == 0){ //Si son colineales...
      if (distsqr(p[0], p[i]) < distsqr(p[0], p[i+1])){ //Borrar
el mas cercano
        p.erase(p.begin() + i);
      }else{
        p.erase(p.begin() + i + 1);
      }
      i--;
    }
  }

  vector<point> chull(p.begin(), p.begin()+3);

  //Ahora sí!!!
  for (int i=3; i<p.size(); ++i){
    while ( chull.size() >= 2 && cross(chull[chull.size()-2],
chull[chull.size()-1], p[i]) < 0){
      chull.erase(chull.end() - 1);
    }
    chull.push_back(p[i]);
  }

  return chull;
}

int main(){
  int n;
  int tileNo = 1;
  while (cin >> n && n){
    vector<point> p;
    point min(10000, 10000);
    int minPos;
    for (int i=0; i<n; ++i){
      int x, y;
      cin >> x >> y;
      p.push_back(point(x,y));
      if (y < min.y || (y == min.y && x < min.x )){
        min = point(x,y);
        minPos = i;
      }
    }
    double tileArea = fabs(area(p));

    //Destruye el orden cw|ccw poligono, pero hay que hacerlo por
que Graham necesita el pivote en p[0]
    swap(p[0], p[minPos]);
    pivot = p[0];
```

```
    double chullArea = fabs(area(graham(p)));

    printf("Tile #%d\n", tileNo++);
    printf("Wasted Space =%.2f \%\n\n",  (chullArea - tileArea) *
100.0 / chullArea);

  }
  return 0;
}
```