Resumen de algoritmos para torneos de programación

Sebastián Arcila Valenzuela

22 de mayo de 2011

1. Plantilla

```
using namespace std;
#include <algorithm>
#include <iostream>
#include <iterator>
#include <sstream>
#include <fstream>
#include <cassert>
#include <climits>
#include <cstdlib>
#include <cstring>
#include <string>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <deque>
#include <stack>
#include <list>
#include <map>
#include <set>

template <class T> string toStr(const T &x)
{ stringstream s; s << x; return s.str(); }
template <class T> int toInt(const T &x)
{ stringstream s; s << x; int r; s >> r; return r; }

#define ALL(x) ((x).begin(),(x).end())
#define D(x) cout << #x " = " << (x) << endl

const double EPS = 1e-9;
int cmp(double x, double y = 0, double tol = EPS){
    return( x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}
```

.....

2. Teoría de números

2.1. Big mod

```
long bigmod(long b, long p, long m){
  if (p == 0) return 1;
  // square(x) = x * x
  else if (p%2 == 0) return square(bigmod (b,p/2,m)) % m;
  else return ((b % m) * bigmod( b,p-1,m)) % m;
}
```

.....

2.2. Phi de Euler

```
//Generate primes with Erathostenes
int fi(int n) {
  if(primes[n]) return n-1;
  int result = n;
  for(int i=2;i*i <= n;i++) {
    if (n % i == 0) result -= result / i;
    while (n % i == 0) n /= i;
  }
  if (n > 1) result -= result / n;
  return result;
}
```

.....

2.3. GCD Extendido

```
int egcd(int a, int b, int &x, int &y){
  x = 0, y = 1;
  int lastx = 1, lasty = 0;
  int quot, temp;
  while(b != 0){
    quot = a/b;
    temp = b;
    b = a%b;
    a = temp;
    temp = x;
    x = lastx - quot*temp;
    lastx = temp;
    temp = y;
    y = lasty - quot*temp;
    lasty = temp;
  }
  x = lastx, y = lasty;
  return a;
}
```

2.4. Fibo

```
typedef unsigned long long uulong;
uulong fib(int n){
  uulong i=1,j=0,k=0,h=1,t=0;
  while(n>0){
    if (n%2==1){ t=j*h;  j=i*h + j*k + t;  i = i*k + t; }
    t = h*h; h = 2*k*h + t; k = k*k + t;
    n = floor(n/2);
  }
  return j;
}
```

2.5. Criba de Eratóstenes

```
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=4; i<=SIZE; i += 2){
    criba[i] = true;
  }
  for (int i=3; i*i<=SIZE; i += 2){
    if (!criba[i]){
      for (int j=i*i; j<=SIZE; j += i){
        criba[j] = true;
      }
    }
  }
}
```

2.6. Factores Primos

```
typedef map<int,int> prime_map;
void sqeeze(prime_map &M, int &n, int p) {
  for ( ; n%p ==0 ;n/=p) M[p]++;
}
prime_map factor(int n){
  prime_map M;
  if(n<0) return factor(-n);
  if(n<2) return M;
  sqeeze(M, n, 2); sqeeze(M, n, 3);
  int maxP = sqrt(n) + 2;
  for(int p=5; p< maxP; p+=6){
    sqeeze(M , n, p); sqeeze(M, n, p+2);
  }
  if(n>1) M[n]++;
  return M;
}
```

3. Combinatoria

3.1. Cuadro resumen

Fórmulas para combinaciones y permutaciones:

Tipo	*¿Se permite la repetición?*	*Fórmula*
r-permutaciones	No	$\frac{n!}{(n-r)!}$
r-combinaciones	No	$\frac{n!}{r!(n-r)!}$
r-permutaciones	Sí	n^r
r-combinaciones	Sí	$\frac{(n+r-1)!}{r!(n-1)!}$

3.2. Combinaciones, coeficientes binomiales, triángulo de Pascal

Complejidad: $O(n^2)$

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \\ 1 & n = k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{en otro caso} \end{cases}$$

```
const int N = 30;
long long choose[N+1][N+1];
  /* Binomial coefficients */
  for (int i=0; i<=N; ++i) choose[i][0] = choose[i][i] = 1;
  for (int i=1; i<=N; ++i)
    for (int j=1; j<i; ++j)
      choose[i][j] = choose[i-1][j-1] + choose[i-1][j];
```

.....

Nota: $\binom{n}{k}$ está indefinido en el código anterior si $n > k$. ¡La tabla puede estar llena con cualquier basura del compilador!

3.3. Permutaciones con elementos indistinguibles

El número de permutaciones diferentes de n objetos, donde hay n_1 objetos indistinguibles de tipo 1, n_2 objetos indistinguibles de tipo 2, ..., y n_k objetos indistinguibles de tipo k, es

$$\frac{n!}{n_1!n_2!\cdots n_k!}$$

Ejemplo: Con las letras de la palabra `PROGRAMAR` se pueden formar $\frac{9!}{2!\cdot 3!} = 30240$ permutaciones diferentes.

3.4. Desordenes, desarreglos o permutaciones completas

Un desarreglo es una permutación donde ningún elemento i está en la posición i-ésima. Por ejemplo, *4213* es un desarreglo de 4 elementos pero *3241* no lo es porque el 2 aparece en la posición 2.

Sea D_n el número de desarreglos de n elementos, entonces:

$$D_n = \begin{cases} 1 & n = 0 \\ 0 & n = 1 \\ (n-1)(D_{n-1} + D_{n-2}) & n \geq 2 \end{cases}$$

Usando el principio de inclusión-exclusión, también se puede encontrar la fórmula

$$D_n = n!\left[1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + (-1)^n\frac{1}{n!}\right] = n!\sum_{i=0}^{n}\frac{(-1)^i}{i!}$$

4. Grafos

4.1. Topological Sort(BFS)

```
/** Creates an edge from u to v. This represents that task
u comes before task v **/
void add_edge(int u, int v){
  g[u].push_back(v);
  d[v]++;
}

int d[MAXN]; //d[i] is the number dependencies

vector<int> top_sort(graph &g, int *d){
  vector<int> order;
  int n = g.size();
  queue<int> q;
```

```
  set <int> inside;
  for(int i=0; i<n; ++i)
  if(d[i]==0){
    q.push(i);
    inside.insert(i);
    order.push_back(i);
  }
  while(q.size()){
    int actual = q.front();
    q.pop();
    inside.erase(actual);
    for(int i=0;i<g[actual].size();++i){
      int next = g[actual][i];
      d[next]--;
      if(d[next]==0){
        if(inside.count(next)) {
          return vector<int>(1,INT_MAX); // There's a cycle
        }
        q.push(next); inside.insert(next); order.push_back(next);
      }
    }
  }
  ir(order.size()!=n)return vector<int>(1,INT_MAX);
  return order;
}
```

4.2. Longest Path in DAG

```
struct node {
  int weight;
  int index;
};
bool visited[MAXNODES];
bool can_go(node n);//retorna true si se puede visitar ese nodo
node best;
int dfs(node root)
{
  memset(visited, false, sizeof visited);
  stack<node> s;
  s.push(root);
  int ans = 0;
  while(s.size())
  {
    node actual = s.top();
    visited[actual.index] = true;
    s.pop();
    int weight = actual.weight;
    if(weight > ans)
    {
      ans = weight;
      best = actual;
    }
      //for para cada vecino)
    if(can_go(vecino))
      s.push(vecino);
  }
  return ans;
}
int max_path_dag()
{
  node root;
  root.weight = 0;
  root.index = 0; // cualquier node del dag funciona
  int t = dfs(root);
  best.weight = 0;
  int ans = dfs(best);
  return ans;
}
```

4.3. Algoritmo de Dijkstra

El peso de todas las aristas debe ser no negativo.

```
// //Complejidad: O(E log V)
// ¡Si hay ciclos de peso negativo, el algoritmo se queda
// en un ciclo infinito!
// Usar Bellman-Ford en ese caso.
struct edge{
  int to, weight;
  edge() {}
  edge(int t, int w) : to(t), weight(w) {}
  bool operator < (const edge &that) const {
```

```
    return weight > that.weight;
  }
};

vector<edge> g[MAXNODES];
// g[i] es la lista de aristas salientes del nodo i.  Cada una
// indica hacia que nodo va (to) y su peso (weight).  Para
// aristas bidireccionales se deben crear 2 aristas dirigidas.

// encuentra el camino más corto entre s y todos los demás
// nodos.
int d[MAXNODES]; //d[i] = distancia más corta desde s hasta i
int p[MAXNODES]; //p[i] = predecesor de i en la ruta más corta
int dijkstra(int s, int n){
  //s = nodo inicial, n = número de nodos
  for (int i=0; i<n; ++i){
    d[i] = INT_MAX;
    p[i] = -1;
  }
  d[s] = 0;
  priority_queue<edge> q;
  q.push(edge(s, 0));
  while (!q.empty()){
    int node = q.top().to;
    int dist = q.top().weight;
    q.pop();

    if (dist > d[node]) continue;
    if (node == t){
      //dist es la distancia más corta hasta t.
      //Para reconstruir la ruta se pueden seguir
      //los p[i] hasta que sea -1.
      return dist;
    }

    for (int i=0; i<g[node].size(); ++i){
      int to = g[node][i].to;
      int w_extra = g[node][i].weight;

      if (dist + w_extra < d[to]){
        d[to] = dist + w_extra;
        p[to] = node;
        q.push(edge(to, d[to]));
      }
    }
  }
  return INT_MAX;
}
```

.....

4.4. Minimum spanning tree: Algoritmo de Kruskal + Union-Find

```
//Complejidad: O(E log V)
struct edge{
  int start, end, weight;
  bool operator < (const edge &that) const {
    //Si se desea encontrar el árbol de recubrimiento de
    //máxima suma, cambiar el < por un >
    return weight < that.weight;
  }
};


///////////////// Empieza Union find ///////////////////////
//Complejidad: O(m log n), donde m es el número de operaciones
//y n es el número de objetos. En la práctica la complejidad
//es casi que O(m).
int p[MAXNODES], rank[MAXNODES];
void make_set(int x){ p[x] = x, rank[x] = 0; }
void link(int x, int y){
  if (rank[x] > rank[y]) p[y] = x;
  else{ p[x] = y; if (rank[x] == rank[y]) rank[y]++; }
}
int find_set(int x){
  return x != p[x] ? p[x] = find_set(p[x]) : p[x];
}
void merge(int x, int y){ link(find_set(x), find_set(y)); }
///////////////// Termina Union find ///////////////////////

//e es un vector con todas las aristas del grafo ¡El grafo
//debe ser no digirido!
long long kruskal(const vector<edge> &e){
  long long total = 0;
  sort(e.begin(), e.end());
```

```
  for (int i=0; i<=n; ++i){
    make_set(i);
  }
  for (int i=0; i<e.size(); ++i){
    int u = e[i].start, v = e[i].end, w = e[i].weight;
    if (find_set(u) != find_set(v)){
      total += w;
      merge(u, v);
    }
  }
  return total;
}
```

4.5. Algoritmo de Floyd-Warshall

```
//Complejidad: O(V^3)
//No funciona si hay ciclos de peso negativo
// g[i][j] = Distancia entre el nodo i y el j.
unsigned long long g[MAXNODES][MAXNODES];
void floyd(int n){
  //Llenar g antes
  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]);
      }
    }
  }
  //Acá se cumple que g[i][j] = Longitud de la ruta más corta
  //de i a j.
}
```

4.6. Algoritmo de Bellman-Ford

Si no hay ciclos de coste negativo, encuentra la distancia más corta entre un nodo y todos los demás. Si sí hay, permite saberlo.
El coste de las aristas sí puede ser negativo (*Debería*, si no es así se puede usar Dijsktra o Floyd).

```
//Complejidad: O(V*E)
const int oo = 1000000000;
struct edge{
  int v, w; edge(){} edge(int v, int w) : v(v), w(w) {}
};
vector<edge> g[MAXNODES];

int d[MAXNODES];
int p[MAXNODES];
// Retorna falso si hay un ciclo de costo negativo alcanzable
// desde s.  Si retorna verdadero, entonces d[i] contiene la
// distancia más corta para ir de s a i.  Si se quiere
// determinar la existencia de un costo negativo que no
// necesariamente sea alcanzable desde s, se crea un nuevo
// nodo A y nuevo nodo B. Para todo nodo original u se crean
// las aristas dirigidas (A, u) con peso 1 y (u, B) con peso
// 1. Luego se corre el algoritmo de Bellman-Ford iniciando en
// A.
bool bellman(int s, int n){
  for (int i=0; i<n; ++i){
    d[i] = oo;
    p[i] = -1;
  }

  d[s] = 0;
  for (int i=0, changed = true; i<n-1 && changed; ++i){
    changed = false;
    for (int u=0; u<n; ++u){
      for (int k=0; k<g[u].size(); ++k){
        int v = g[u][k].v, w = g[u][k].w;
        if (d[u] + w < d[v]){
          d[v] = d[u] + w;
          p[v] = u;
          changed = true;
        }
      }
    }
  }

  for (int u=0; u<n; ++u){
    for (int k=0; k<g[u].size(); ++k){
      int v = g[u][k].v, w = g[u][k].w;
      if (d[u] + w < d[v]){
```

```
        //Negative weight cycle!

        //Finding the actual negative cycle. If not needed
        //return false immediately.
        vector<bool> seen(n, false);
        deque<int> cycle;
        int cur = v;
        for (; !seen[cur]; cur = p[cur]){
          seen[cur] = true;
          cycle.push_front(cur);
        }
        cycle.push_front(cur);
        //there's a negative cycle that goes from
        //cycle.front() until it reaches itself again
        printf("Negative weight cycle reachable from s:\n");
        int i = 0;
        do{
          printf("%d ", cycle[i]);
          i++;
        }while(cycle[i] != cycle[0]);
        printf("\n");
        // Negative weight cycle found

        return false;
      }
    }
  }
  return true;
}
```

4.7. Puntos de articulación

```
// Complejidad: O(E + V)

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; //contendrá los puntos de articulación
int timeCount;

// Uso: Para cada nodo u:
// colors[u] = WHITE, g[u] = Aristas salientes de u.
// Funciona para grafos no dirigidos.

void dfs(node v, bool isRoot = true){
  colors[v] = GRAY;
  d[v] = low[v] = ++timeCount;
  const 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]){
        //current node is an articulation point
        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;
}
```

4.8. Máximo flujo: Método de Ford-Fulkerson, algoritmo de Edmonds-Karp

El algoritmo de Edmonds-Karp es una modificación al método de Ford-Fulkerson. Este último utiliza DFS para hallar un camino de aumentación, pero la sugerencia de Edmonds-Karp es utilizar BFS que lo hace más eficiente en algunos grafos.

```
/*
  cap[i][j] = Capacidad de la arista (i, j).
  prev[i] = Predecesor del nodo i en un camino de aumentación.
 */
int cap[MAXN+1][MAXN+1], prev[MAXN+1];

vector<int> g[MAXN+1]; //Vecinos de cada nodo.
inline void link(int u, int v, int c)
{ cap[u][v] = c; g[u].push_back(v), g[v].push_back(u); }
/*
  Notar que link crea las aristas (u, v) && (v, u) en el grafo
  g. Esto es necesario porque el algoritmo de Edmonds-Karp
  necesita mirar el "back-edge" (j, i) que se crea al bombear
  flujo a través de (i, j). Sin embargo, no modifica
  cap[v][u], porque se asume que el grafo es dirigido. Si es
  no-dirigido, hacer cap[u][v] = cap[v][u] = c.
*/


/*
  Método 1:

  Mantener la red residual, donde residual[i][j] = cuánto
  flujo extra puedo inyectar a través de la arista (i, j).

  Si empujo k unidades de i a j, entonces residual[i][j] -= k
  y residual[j][i] += k (Puedo "desempujar" las k unidades de
  j a i).

  Se puede modificar para que no utilice extra memoria en la
  tabla residual, sino que modifique directamente la tabla
  cap.
*/

int residual[MAXN+1][MAXN+1];
int fordFulkerson(int n, int s, int t){
  memcpy(residual, cap, sizeof cap);

  int ans = 0;
  while (true){
    fill(prev, prev+n, -1);
    queue<int> q;
    q.push(s);
    while (q.size() && prev[t] == -1){
      int u = q.front();
      q.pop();
      vector<int> &out = g[u];
      for (int k = 0, m = out.size(); k<m; ++k){
        int v = out[k];
        if (v != s && prev[v] == -1 && residual[u][v] > 0)
          prev[v] = u, q.push(v);
      }
    }

    if (prev[t] == -1) break;

    int bottleneck = INT_MAX;
    for (int v = t, u = prev[v]; u != -1; v = u, u = prev[v]){
      bottleneck = min(bottleneck, residual[u][v]);
    }
    for (int v = t, u = prev[v]; u != -1; v = u, u = prev[v]){
      residual[u][v] -= bottleneck;
      residual[v][u] += bottleneck;
    }
    ans += bottleneck;
  }
  return ans;
}



/*
  Método 2:

  Mantener la red de flujos, donde flow[i][j] = Flujo que,
  err, fluye de i a j. Notar que flow[i][j] puede ser
  negativo. Si esto pasa, es lo equivalente a decir que i
  "absorbe" flujo de j, o lo que es lo mismo, que hay flujo
  positivo de j a i.

  En cualquier momento se cumple la propiedad de skew
  symmetry, es decir, flow[i][j] = -flow[j][i]. El flujo neto
  de i a j es entonces flow[i][j].
*/

int flow[MAXN+1][MAXN+1];
```

```
int fordFulkerson(int n, int s, int t){
  //memset(flow, 0, sizeof flow);
  for (int i=0; i<n; ++i) fill(flow[i], flow[i]+n, 0);
  int ans = 0;
  while (true){
    fill(prev, prev+n, -1);
    queue<int> q;
    q.push(s);
    while (q.size() && prev[t] == -1){
      int u = q.front();
      q.pop();
      vector<int> &out = g[u];
      for (int k = 0, m = out.size(); k<m; ++k){
        int v = out[k];
        if (v != s && prev[v] == -1 && cap[u][v] > flow[u][v])
          prev[v] = u, q.push(v);
      }
    }

    if (prev[t] == -1) break;

    int bottleneck = INT_MAX;
    for (int v = t, u = prev[v]; u != -1; v = u, u = prev[v]){
      bottleneck = min(bottleneck, cap[u][v] - flow[u][v]);
    }
    for (int v = t, u = prev[v]; u != -1; v = u, u = prev[v]){
      flow[u][v] += bottleneck;
      flow[v][u] = -flow[u][v];
    }
    ans += bottleneck;
  }
  return ans;
}
```

.....

4.9. Máximo flujo para grafos dispersos usando Ford-Fulkerson

```
/////////////// Maximum flow for sparse graphs ///////////////
///////////////    Complexity: O(V * E^2)      ///////////////
```

```
/*
  Usage:
  initialize_max_flow();
  Create graph using add_edge(u, v, c);
  max_flow(source, sink);

  WARNING: The algorithm writes on the cap array. The capacity
  is not the same after having run the algorithm.  If you need
  to run the algorithm several times on the same graph, backup
  the cap array.
 */

const int MAXE = 50000; //Maximum number of edges
const int oo = INT_MAX / 4;
int cap[MAXE];
int first[MAXE];
int next[MAXE];
int adj[MAXE];
int current_edge;

/*
  Builds a directed edge (u, v) with capacity c.
  Note that actually two edges are added, the edge
  and its complementary edge for the backflow.
 */
int add_edge(int u, int v, int c){
  adj[current_edge] = v;
  cap[current_edge] = c;
  next[current_edge] = first[u];
  first[u] = current_edge++;

  adj[current_edge] = u;
  cap[current_edge] = 0;
  next[current_edge] = first[v];
  first[v] = current_edge++;
}

void initialize_max_flow(){
  current_edge = 0;
  memset(next, -1, sizeof next);
  memset(first, -1, sizeof first);
}
```

```
int q[MAXE];
int incr[MAXE];
int arrived_by[MAXE];
//arrived_by[i] = The last edge used to reach node i
int find_augmenting_path(int src, int snk){
  /*
    Make a BFS to find an augmenting path from the source to
    the sink.  Then pump flow through this path, and return
    the amount that was pumped.
  */
  memset(arrived_by, -1, sizeof arrived_by);
  int h = 0, t = 0;
  q[t++] = src;
  arrived_by[src] = -2;
  incr[src] = oo;
  while (h < t && arrived_by[snk] == -1){ //BFS
    int u = q[h++];
    for (int e = first[u]; e != -1; e = next[e]){
      int v = adj[e];
      if (arrived_by[v] == -1 && cap[e] > 0){
        arrived_by[v] = e;
        incr[v] = min(incr[u], cap[e]);
        q[t++] = v;
      }
    }
  }

  if (arrived_by[snk] == -1) return 0;

  int cur = snk;
  int neck = incr[snk];
  while (cur != src){
    //Remove capacity from the edge used to reach node "cur"
    //Add capacity to the backedge
    cap[arrived_by[cur]] -= neck;
    cap[arrived_by[cur] ^ 1] += neck;
    //move backwards in the path
    cur = adj[arrived_by[cur] ^ 1];
  }
  return neck;
}

int max_flow(int src, int snk){
  int ans = 0, neck;
  while ((neck = find_augmenting_path(src, snk)) != 0){
    ans += neck;
  }
  return ans;
}
```

.....

4.10. Máximo flujo para grafos disperos usando algoritmo de Dinic

```
/*
  ACRush's Dinic algorithm for maximum flow
  Complexity: O(E V^2)

  Usage:
  init(number of nodes, source, sink);
  Create graph using add_edge(int u, int v, int c1, int c2):
  This adds two directed edges: u -> v with capacity c1
    and v -> u with capacity c2.
  c2 by default is 0.
  After creating the graph, nedge contains the number of
     total edges.
  dinic_flow();
  This doesn't modify the capacity of the original graph,
     so you can run the algorithm several times on the same
     graph.
  If you want to run the algorithm with different sources/sinks
     assign the correct value to src and dest before calling
     dinic_flow().
 */

const int maxnode=2*55 + 5; const int
maxedge=maxnode*(maxnode-1)/2; const int oo=1000000000;

int node,src,dest,nedge; int
head[maxnode],point[maxedge],next[maxedge],
flow[maxedge],capa[maxedge];
int dist[maxnode],Q[maxnode],work[maxnode];

void init(int _node,int _src,int _dest) { node=_node;
src=_src; dest=_dest; for (int i=0;i<node;i++) head[i]=-1;
```

```
nedge=0; } void add_edge(int u,int v,int c1,int c2 = 0) {
point[nedge]=v,capa[nedge]=c1,flow[nedge]=0,
next[nedge]=head[u],head[u]=(nedge++);
point[nedge]=u,capa[nedge]=c2,flow[nedge]=0,
next[nedge]=head[v],head[v]=(nedge++);
} bool dinic_bfs() { memset(dist,255,sizeof(dist));
dist[src]=0; int sizeQ=0; Q[sizeQ++]=src; for (int
cl=0;cl<sizeQ;cl++) for (int k=Q[cl],i=head[k];i>=0;i=next[i])
if (flow[i]<capa[i] && dist[point[i]]<0) {
dist[point[i]]=dist[k]+1; Q[sizeQ++]=point[i]; } return
dist[dest]>=0; } int dinic_dfs(int x,int exp) { if (x==dest)
return exp; for (int &i=work[x];i>=0;i=next[i]) { int
v=point[i],tmp; if (flow[i]<capa[i] && dist[v]==dist[x]+1 &&
(tmp=dinic_dfs(v,min(exp,capa[i]-flow[i])))>0) { flow[i]+=tmp;
flow[i^1]-=tmp; return tmp; } } return 0; } int dinic_flow() {
for (int i=0; i<nedge; ++i) flow[i] = 0; int result=0; while
(dinic_bfs()) { for (int i=0;i<node;i++) work[i]=head[i];
while (1) { int delta=dinic_dfs(src,oo); if (delta==0) break;
result+=delta; } } return result; }
```

.....

4.11. Máximo flujo mínimo costo

Implementación de Misof:

```
const int MAXN = 105, oo = INT_MAX / 2 - 1;
int cap[MAXN][MAXN];
int cost[MAXN][MAXN];
int flow[MAXN][MAXN];
/* Uso:
  Llenar cap y cost.
  Llenar flow con 0s.
  Invocar la función.
 */
pair<int, int> min_cost_max_flow(int N, int source, int sink)
  { int flowSize = 0; int flowCost = 0; int infinity = 1;
  while (2*infinity > infinity) infinity *= 2;
  // speed optimization #1: adjacency graph
  // speed optimization #2: cache the degrees
  vector<int> deg(N,0); vector<vector<int> > G(N); for (int
  i=0; i<N; i++) for (int j=0; j<i; j++) if (cap[i][j]>0 ||
  cap[j][i]>0) { deg[i]++; deg[j]++; G[i].push_back(j);
  G[j].push_back(i); } vector<int> potential(N,0); while (1) {
// use dijkstra to find an augmenting path
vector<int> from(N,-1); vector<int> dist(N,infinity);
priority_queue< pair<int,int>, vector<pair<int,int> >,
greater<pair<int,int> > > Q; Q.push(make_pair(0,source));
from[source]=-2; dist[source] = 0; while (!Q.empty()) {
int howFar = Q.top().first; int where = Q.top().second;
Q.pop(); if (dist[where] < howFar) continue; for (int i=0;
i<deg[where]; i++) { int dest = G[where][i];
  // now there are two possibilities: add flow in
  // the right direction, or remove in the other one
    if (flow[dest][where] > 0) if (dist[dest] >
      dist[where] + potential[where] - potential[dest] -
      cost[dest][where]) { dist[dest] = dist[where] +
      potential[where] - potential[dest] -
      cost[dest][where]; from[dest] = where;
      Q.push(make_pair(dist[dest],dest)); } if
      (flow[where][dest] < cap[where][dest]) if
      (dist[dest] > dist[where] + potential[where] -
      potential[dest] + cost[where][dest]) { dist[dest] =
      dist[where] + potential[where] - potential[dest] +
      cost[where][dest]; from[dest] = where;
      Q.push(make_pair(dist[dest],dest)); }
    // no breaking here, we need the whole graph
  } }
// update vertex potentials
for (int i=0; i<N; i++) potential[i] += dist[i];
// if there is no path, we are done
if (from[sink] == -1) return make_pair(flowSize,flowCost);
// construct an augmenting path
int canPush = infinity; int where = sink; while (1) { int
prev=from[where]; if (prev==-2) break; if
(flow[where][prev]) canPush = min( canPush,
flow[where][prev] ); else canPush = min( canPush,
cap[prev][where] - flow[prev][where] ); where=prev; }
// update along the path
where = sink; while (1) { int prev=from[where]; if
(prev==-2) break; if (flow[where][prev]) {
flow[where][prev] -= canPush; flowCost -= canPush *
cost[where][prev]; } else { flow[prev][where] += canPush;
flowCost += canPush * cost[prev][where]; } where=prev; }
flowSize += canPush; } return make_pair(0, oo); }
```

4.12. Componentes fuertemente conexas: Algoritmo de Tarjan

```
/* Complexity: O(E + V)
 Tarjan's algorithm for finding strongly connected
components.

 *d[i] = Discovery time of node i. (Initialize to -1)
 *low[i] = Lowest discovery time reachable from node
 i. (Doesn't need to be initialized)
 *scc[i] = Strongly connected component of node i. (Doesn't
 need to be initialized)
 *s = Stack used by the algorithm (Initialize to an empty
 stack)
 *stacked[i] = True if i was pushed into s. (Initialize to
 false)
 *ticks = Clock used for discovery times (Initialize to 0)
 *current_scc = ID of the current_scc being discovered
 (Initialize to 0)
*/
vector<int> g[MAXN];
int d[MAXN], low[MAXN], scc[MAXN];
bool stacked[MAXN];
stack<int> s;
int ticks, current_scc;
void tarjan(int u){
  d[u] = low[u] = ticks++;
  s.push(u);
  stacked[u] = true;
  const vector<int> &out = g[u];
  for (int k=0, m=out.size(); k<m; ++k){
    const int &v = out[k];
    if (d[v] == -1){
      tarjan(v);
      low[u] = min(low[u], low[v]);
    }else if (stacked[v]){
      low[u] = min(low[u], low[v]);
    }
  }
  if (d[u] == low[u]){
    int v;
    do{
      v = s.top();
      s.pop();
      stacked[v] = false;
      scc[v] = current_scc;
    }while (u != v);
    current_scc++;
  }
}
```

4.13. 2-Satisfiability

Dada una ecuación lógica de conjunciones de disyunciones de 2 términos, se pretente decidir si existen valores de verdad que puedan asignarse a las variables para hacer cierta la ecuación.
Por ejemplo, $(b_1 \vee \neg b_2) \wedge (b_2 \vee b_3) \wedge (\neg b_1 \vee \neg b_2)$ es verdadero cuando b_1 y b_3 son verdaderos y b_2 es falso.
Solución: Se sabe que $(p \to q) \leftrightarrow (\neg p \vee q)$. Entonces se traduce cada disyunción en una implicación y se crea un grafo donde los nodos son cada variable y su negación. Cada implicación es una arista en este grafo. Existe solución si nunca se cumple que una variable y su negación están en la misma componenete fuertemente conexa (Se usa el algoritmo de Tarjan, 4.12).

5. Programación dinámica

5.1. Kadane

```
const int MAXN = 22;

int cube[MAXN][MAXN][MAXN];
int mat[MAXN][MAXN];
int arr[MAXN];

int n;

// Returns the maximum sum inside an array
// The sum best = Sum i in [from, to]
int kadane(){
  int best=1<<31,current=0,from=0,to=0,aa=0;
  for(int i=0;i<MAXN;++i){
    current += arr[i];
```

```
    if ( current > best ){ best=current; from=aa; to=i;}
    if ( current < 0 ){ current = 0; aa = i+1;}
  }
  return best;
}
// Returns the submatrix with maximum sum
// The sum is inside the matrix (xi,y1) - (x2, y2)
// A is the matrix, N the size
int kadane2D () {
  vector<int>pr(102,0);
  int S = 1<<31, s=0, k,l,x1=0,x2=0,y1=0,y2=0,j,t;
  for(int z=0;z < N;++z){
    pr = vector<int>(MAXN,0);
    for(int x=z;x<N;++x){
      t=0;s = 1<<31;j=k=l=0;
      for(int i=0;i<N;++i) {
        pr[i]=pr[i]+a[x][i]; t=t+pr[i];
        if (t>s){ s = t; k = i; l = j;}
        if(t<0){ t=0; j=i+1;}
      }
      if (s > S) { S = s; x1 = x; y1 = k; x2 = z;  y2 = l;}
    }
  }
  return S;
}

// Easier to implement. Less information
int best2D(){
  int ans = 0;
  for(int i=0; i<n; ++i){
    memset(arr, 0, sizeof arr);
    for (int j=i; j<n; ++j){
      for (int k=0; k<n; ++k) arr[k] += mat[j][k];
      int sum = 0;
      for (int k=0; k<n; ++k){
        sum += arr[k];
        ans = max(ans, sum);
        if (sum < 0) sum = 0;
      }
    }
  }
  return ans;
}
```

```
// Cube has the actual input. If all numbers in cube are negative
// the maximum sum is the biggest of the numbers
int kadane3D(){
  int ans = 0;
  for (int i=0; i<n; ++i){
    memset(mat, 0, sizeof mat);
    for (int j=i; j<n; ++j){
      for (int ii=0; ii<n; ++ii){
        for (int jj=0; jj<n; ++jj){
          mat[ii][jj] += cube[j][ii][jj];
        }
      }
      ans = max(ans, ());
    }
  }
  return ans;
}
```

.....

5.2. LIS

```
#define INF 2<<30-1

int main(){
  int n;
  while(scanf("%d", &n)==1){
    vector<long>S(n);
    vector<long>M(n+1,INF);
    for(int i=0;i<n;++i) scanf("%ld", &S[i]);
    M[0]=0;
    int _m = 0;
    for(int i=0; i<S.size();++i){
      int d = upper_bound(M.begin(),M.begin()+n,S[i]) - M.begin();
      if(S[i]!=M[d-1]){
        M[d] = S[i];
        _m = max(_m,d);
        //parent[S[i]] = M[d-1];
      }
    }
```

```
    printf("%d\n",max(1,_m));
  }
  return 0;
}
```

.....

5.3. 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];
}
```

.....

5.4. Partición de troncos

Este problema es similar al problema de *Matrix Chain Multiplication*. Se tiene un tronco de longitud n, y m puntos de corte en el tronco. Se puede hacer un corte a la vez, cuyo costo es igual a la longitud del tronco. ¿Cuál es el mínimo costo para partir todo el tronco en pedacitos individuales?
Ejemplo: Se tiene un tronco de longitud 10. Los puntos de corte son 2, 4, y 7. El mínimo costo para partirlo es 20, y se obtiene así:

- Partir el tronco (0, 10) por 4. Vale 10 y quedan los troncos (0, 4) y (4, 10).
- Partir el tronco (0, 4) por 2. Vale 4 y quedan los troncos (0, 2), (2, 4) y (4, 10).
- No hay que partir el tronco (0, 2).
- No hay que partir el tronco (2, 4).
- Partir el tronco (4, 10) por 7. Vale 6 y quedan los troncos (4, 7) y (7, 10).
- No hay que partir el tronco (4, 7).
- No hay que partir el tronco (7, 10).
- El costo total es $10 + 4 + 6 = 20$.

El algoritmo es $O(n^3)$, pero optimizable a $O(n^2)$ con una tabla adicional:

```
/*
  O(n^3)
 dp[i][j] = Mínimo costo de partir la cadena entre las
 particiones i e j, ambas incluídas.
*/
int dp[1005][1005];
int p[1005];

int cubic(){
  int n, m;
  while (scanf("%d %d", &n, &m)==2){
    p[0] = 0;
    for (int i=1; i<=m; ++i){
      scanf("%d", &p[i]);
    }
    p[m+1] = n;
    m += 2;

    for (int i=0; i<m; ++i){
      dp[i][i+1] = 0;
    }

    for (int i=m-2; i>=0; --i){
      for (int j=i+2; j<m; ++j){
        dp[i][j] = p[j]-p[i];
        int t = INT_MAX;
        for (int k=i+1; k<j; ++k){
          t = min(t, dp[i][k] + dp[k][j]);
        }
        dp[i][j] += t;
      }
    }
```

```
    printf("%d\n", dp[0][m-1]);
  }
  return 0;
}


/*
  O(n^2)

  dp[i][j] = Mínimo costo de partir la cadena entre las
  particiones i e j, ambas incluídas.  pivot[i][j] = Índice de
  la partición que usé para lograr dp[i][j].
 */
int dp[1005][1005], pivot[1005][1005];
int p[1005];

int quadratic(){
  int n, m;
  while (scanf("%d %d", &n, &m)==2){
    p[0] = 0;
    for (int i=1; i<=m; ++i){
      scanf("%d", &p[i]);
    }
    p[m+1] = n;
    m += 2;

    for (int i=0; i<m-1; ++i){
      dp[i][i+1] = 0;
    }
    for (int i=0; i<m-2; ++i){
      dp[i][i+2] = p[i+2] - p[i];
      pivot[i][i+2] = i+1;
    }

    for (int d=3; d<m; ++d){ //d = longitud
      for (int j, i=0; (j = i + d) < m; ++i){
        dp[i][j] = p[j] - p[i];
        int t = INT_MAX, s;
        for (int k=pivot[i][j-1]; k<=pivot[i+1][j]; ++k){
          int x = dp[i][k] + dp[k][j];
          if (x < t) t = x, s = k;
        }
        dp[i][j] += t, pivot[i][j] = s;
      }
    }

    printf("%d\n", dp[0][m-1]);
  }
  return 0;
}
```

.....

6. Geometría

6.1. Cabe, Area y Centroide

```
// Returns true if pXq is inside aXb
bool cabe(long p, long q, long a, long b){
  long x,y,z,q; if(p<q) swap(p,q); if(a<b) swap(a,b);
  if(p<=a && q<=b) return true;
  if(p==q) return b>=q;
  x = 2*p*q*a; y=p*p-q*q; z=p*p+q*q; w=z-a*a;
  return p>a && 1.0*b*z >= x+y*sqrt(w) - 1e-10;
}

// Centroide (centro de masa) de un polno
// pt[i][0] = pt[i].x  |  pt[i][1] = pt[i].y
// Area will return positive or negative
double area(vector<vector<double> > &pt){
  double r = 0.0; int t = pt.size();
  for(int i = 0, j = 1; i<t; i++, j = j+1 == t? 0 : j+1){
    r+= (pt[i][0] * pt[j][1] - pt[i][1] * pt[j][0]);
  }
  return r/2.0;
}

pair<double, double> centroide(vector<vector<double> > &pt){
  double d = area(pt) * 6.0;
  double p[2];
  p[0] = p[1] = 0.0;
  for(int i = 0, j = 1, t = pt.size(); i<t; i++,
    j = j+1 ==t ? 0 : j+1)
    for(int k = 0; k<2;k++)
    p[k] += (pt[i][k] + pt[j][k]) *
```

```
      (pt[i][0] * pt[j][1] - pt[j][0] * pt[i][1]);
  return pair<double, double>(pt[0]/d, pt[1]/d);
}
```

6.2. Convex hull: Graham Scan

Complejidad: $O(n \log_2 n)$

```
//Graham scan: Complexity: O(n log n)
struct point{
  int x,y;
  point() {}
  point(int X, int Y) : x(X), y(Y) {}
};

point pivot;

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);
}

inline double dist(const point &a, const point &b){
  return sqrt(distsqr(a, b));
}

//retorna > 0 si c esta a la izquierda del segmento AB
//retorna < 0 si c esta a la derecha del segmento AB
//retorna == 0 si c es colineal con 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){
  int t = cross(pivot, that, self);
  if (t < 0) return true;
  if (t == 0){
    //Self < that si está más cerquita
    return (distsqr(pivot, self) < distsqr(pivot, that));
  }
  return false;
}

vector<point> graham(vector<point> p){
  //Metemos el más abajo más a la izquierda en la posición 0
  for (int i=1; i<p.size(); ++i){
    if (p[i].y < p[0].y ||
        (p[i].y == p[0].y && p[i].x < p[0].x))
      swap(p[0], p[i]);
  }

  pivot = p[0];
  sort(p.begin(), p.end(), angleCmp);

  //Ordenar por ángulo y eliminar repetidos.
  //Si varios puntos tienen el mismo angulo el más lejano
  //queda después en la lista
  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]);
  }
  //chull contiene los puntos del convex hull ordenados
  //anti-clockwise.  No contiene ningún punto repetido.  El
  //primer punto no es el mismo que el último, i.e, la última
  //arista va de chull[chull.size()-1] a chull[0]
  return chull;
}
```

6.3. Mínima distancia entre un punto y un segmento

```
/*
  Returns the closest distance between point pnt and the segment
  that goes from point a to b
  Idea by:
```

```
  http://local.wasp.uwa.edu.au/~pbourke/geometry/pointline/
 */
double distance_point_to_segment(const point &a,const point &b,
                                 const point &pnt){
  double u =
    ((pnt.x - a.x)*(b.x - a.x) + (pnt.y - a.y)*(b.y - a.y))
    /distsqr(a, b);
  point intersection;
  intersection.x = a.x + u*(b.x - a.x);
  intersection.y = a.y + u*(b.y - a.y);
  if (u < 0.0 || u > 1.0){
    return min(dist(a, pnt), dist(b, pnt));
  }
  return dist(pnt, intersection);
}
```

6.4. Mínima distancia entre un punto y una recta

```
/*
  Returns the closest distance between point pnt and the line
  that passes through points a and b
  Idea by:
  http://local.wasp.uwa.edu.au/~pbourke/geometry/pointline/
*/
double distance_point_to_line(const point &a, const point &b,
                              const point &pnt){
  double u =
    ((pnt.x - a.x)*(b.x - a.x)+(pnt.y - a.y)*(b.y - a.y))
    /distsqr(a, b);
  point intersection;
  intersection.x = a.x + u*(b.x - a.x);
  intersection.y = a.y + u*(b.y - a.y);
  return dist(pnt, intersection);
}
```

6.5. Determinar si un polígono es convexo

```
/*
  Returns positive if a-b-c make a left turn.
  Returns negative if a-b-c make a right turn.
  Returns 0.0 if a-b-c are colineal.
 */
double turn(const point &a, const point &b, const point &c){
  double z = (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
  if (fabs(z) < 1e-9) return 0.0;
  return z;
}

/*
  Returns true if polygon p is convex.
  False if it's concave or it can't be determined
  (For example, if all points are colineal we can't
  make a choice).
 */
bool isConvexPolygon(const vector<point> &p){
  int mask = 0;
  int n = p.size();
  for (int i=0; i<n; ++i){
    int j=(i+1)%n;
    int k=(i+2)%n;
    double z = turn(p[i], p[j], p[k]);
    if (z < 0.0){
      mask |= 1;
    }else if (z > 0.0){
      mask |= 2;
    }
    if (mask == 3) return false;
  }
  return mask != 0;
}
```

6.6. Determinar si un punto está dentro de un polígono convexo

```
/*
  Returns true if point a is inside convex polygon p.  Note
  that if point a lies on the border of p it is considered
  outside.

  We assume p is convex! The result is useless if p is
```

```
  concave.
 */
bool insideConvexPolygon(const vector<point> &p,
                         const point &a){
  int mask = 0;
  int n = p.size();
  for (int i=0; i<n; ++i){
    int j = (i+1)%n;
    double z = turn(p[i], p[j], a);
    if (z < 0.0){
      mask |= 1;
    }else if (z > 0.0){
      mask |= 2;
    }else if (z == 0.0) return false;
    if (mask == 3) return false;
  }
  return mask != 0;
}
```

6.7. Determinar si un punto está dentro de un polígono cualquiera

Field-testing:

- *TopCoder* - SRM 187 - Division 2 Hard - PointInPolygon

```
//Point
//Choose one of these two:
struct P {
  double x, y; P(){}; P(double q, double w) : x(q), y(w){}
};
struct P {
  int x, y; P(){}; P(int q, int w) : x(q), y(w){}
};

// Polar angle
// Returns an angle in the range [0, 2*Pi) of a given Cartesian point.
// If the point is (0,0), -1.0 is returned.
// REQUIRES:
// include math.h
// define EPS 0.000000001, or your choice
// P has members x and y.
double polarAngle( P p )
{
  if(fabs(p.x) <= EPS && fabs(p.y) <= EPS) return -1.0;
  if(fabs(p.x) <= EPS) return (p.y > EPS ? 1.0 : 3.0) * acos(0);
  double theta = atan(1.0 * p.y / p.x);
  if(p.x > EPS) return(p.y >= -EPS ? theta : (4*acos(0) + theta));
  return(2 * acos( 0 ) + theta);
}

//Point inside polygon
// Returns true iff p is inside poly.
// PRE: The vertices of poly are ordered (either clockwise or
//      counter-clockwise.
// POST: Modify code inside to handle the special case of "on
// an edge".
// REQUIRES:
// polarAngle()
// include math.h
// include vector
// define EPS 0.000000001, or your choice
bool pointInPoly( P p, vector< P > &poly )
{
  int n = poly.size();
  double ang = 0.0;
  for(int i = n - 1, j = 0; j < n; i = j++){
    P v( poly[i].x - p.x, poly[i].y - p.y );
    P w( poly[j].x - p.x, poly[j].y - p.y );
    double va = polarAngle(v);
    double wa = polarAngle(w);
    double xx = wa - va;
    if(va < -0.5 || wa < -0.5 || fabs(fabs(xx)-2*acos(0)) < EPS){
        // POINT IS ON THE EDGE
        assert( false );
        ang += 2 * acos( 0 );
        continue;
      }
    if( xx < -2 * acos( 0 ) ) ang += xx + 4 * acos( 0 );
    else if( xx > 2 * acos( 0 ) ) ang += xx - 4 * acos( 0 );
    else ang += xx;
  }
  return( ang * ang > 1.0 );
}
```

6.8. Intersección de dos rectas

```
/*
  Finds the intersection between two lines (Not segments!
```

```
  Infinite lines)
  Line 1 passes through points (x0, y0) and (x1, y1).
  Line 2 passes through points (x2, y2) and (x3, y3).

  Handles the case when the 2 lines are the same (infinite
  intersections),
  parallel (no intersection) or only one intersection.
*/
void line_line_intersection(double x0, double y0,
                            double x1, double y1,

                            double x2, double y2,
                            double x3, double y3){
#ifndef EPS
#define EPS 1e-9
#endif

  double t0 = (y3-y2)*(x0-x2)-(x3-x2)*(y0-y2);
  double t1 = (x1-x0)*(y2-y0)-(y1-y0)*(x2-x0);
  double det = (y1-y0)*(x3-x2)-(y3-y2)*(x1-x0);
  if (fabs(det) < EPS){
    //parallel
    if (fabs(t0) < EPS || fabs(t1) < EPS){
      //same line
      printf("LINE\n");
    }else{
      //just parallel
      printf("NONE\n");
    }
  }else{
    t0 /= det;
    t1 /= det;
    double x = x0 + t0*(x1-x0);
    double y = y0 + t0*(y1-y0);
    //intersection is point (x, y)
    printf("POINT %.2lf %.2lf\n", x, y);
  }
}
```

6.9. Intersección de dos segmentos de recta

```
/*
  Returns true if point (x, y) lies inside (or in the border)
  of box defined by points (x0, y0) and (x1, y1).
*/
bool point_in_box(double x, double y,
                  double x0, double y0,
                  double x1, double y1){
  return
    min(x0, x1) <= x && x <= max(x0, x1) &&
    min(y0, y1) <= y && y <= max(y0, y1);
}

/*
  Finds the intersection between two segments (Not infinite
  lines!)
  Segment 1 goes from point (x0, y0) to (x1, y1).
  Segment 2 goes from point (x2, y2) to (x3, y3).

  (Can be modified to find the intersection between a segment
  and a line)

  Handles the case when the 2 segments are:
  *Parallel but don't lie on the same line (No intersection)
  *Parallel and both lie on the same line (Infinite
  *intersections or no intersections)
  *Not parallel (One intersection or no intersections)

  Returns true if the segments do intersect in any case.
*/
bool segment_segment_intersection(double x0, double y0,
                                  double x1, double y1,

                                  double x2, double y2,
                                  double x3, double y3){
#ifndef EPS
#define EPS 1e-9
#endif

  double t0 = (y3-y2)*(x0-x2)-(x3-x2)*(y0-y2);
  double t1 = (x1-x0)*(y2-y0)-(y1-y0)*(x2-x0);
  double det = (y1-y0)*(x3-x2)-(y3-y2)*(x1-x0);
  if (fabs(det) < EPS){
    //parallel
    if (fabs(t0) < EPS || fabs(t1) < EPS){
      //they lie on same line, but they may or may not intersect.
      return (point_in_box(x0, y0,   x2, y2, x3, y3) ||
              point_in_box(x1, y1,   x2, y2, x3, y3) ||
              point_in_box(x2, y2,   x0, y0, x1, y1) ||
              point_in_box(x3, y3,   x0, y0, x1, y1));
    }else{
```

```
      //just parallel, no intersection
      return false;
    }
  }else{
    t0 /= det;
    t1 /= det;
    /*
      0 <= t0 <= 1 iff the intersection point lies in segment 1.
      0 <= t1 <= 1 iff the intersection point lies in segment 2.
    */
    if (0.0 <= t0 && t0 <= 1.0 && 0.0 <= t1 && t1 <= 1.0){
      double x = x0 + t0*(x1-x0);
      double y = y0 + t0*(y1-y0);
      //intersection is point (x, y)
      return true;
    }
    //the intersection points doesn't lie on both segments.
    return false;
  }
}
```

.....

7. Estructuras de datos

7.1. RMQ

```
#define sz(A) (int)(A).size()
#define FOR(A,B) for(int A=0; A < (int) (B);A++)
#define pb push_back
#define N 100000
#define inf 10000001
int dist[N], pai[N], pp[N][20], mini[N][20];
int maxi[N][20], n;
struct X {
  int v,c;
};

X no(int a, int b) {
  X x;
  x.v = a;
  x.c = b;
  return x;
}
vector< X > adj[N];
void dfs(int v, int p, int f, int c) {
  pai[v] = f;
  dist[v] = p;
  mini[v][0] = c;
  maxi[v][0] = c;
  FOR(i,sz(adj[v])) {
    if(adj[v][i].v == f) continue;
    dfs(adj[v][i].v, p+1, v, adj[v][i].c);
  }
}

void pre() {
  memset(pai,-1,sizeof(pai));

  FOR(i,n) for(int j=0; (1<<j) < n; j++) {
    pp[i][j] = -1;
    mini[i][j] = inf;
    maxi[i][j] = -inf;
  }

  dfs(0,0,-1,0);
  pai[0] = 0;
  maxi[0][0] = -inf;
  mini[0][0] = inf;

  FOR(i,n) pp[i][0] = pai[i];

  for(int j=1; (1<<j) < n; j++) {
    for(int i=0; i < n; i++) {
      if(pp[i][j-1] == -1) continue;
      pp[i][j] = pp[ pp[i][j-1] ] [j-1];
      mini[i][j] = min(mini[pp[i][j-1]][j-1], mini[i][j-1] );
      maxi[i][j] = max(maxi[pp[i][j-1]][j-1], maxi[i][j-1] );
    }
  }

}

int resolve(int a, int b) {
  int rmini = inf;
  int rmaxi = -inf;

  if(dist[a] < dist[b]) swap(a,b);
  int log;
  for(log = 1; (1<<log) <= dist[a]; log++);
  log--;
  for(int i=log; i >= 0; i--)
```

```
    if(dist[a] - (1<<i) >= dist[b]) {
      rmini = min(rmini, mini[a][i]);
      rmaxi = max(rmaxi, maxi[a][i]);
      a = pp[a][i];
    }

  if(a == b) {
    printf("%d %d\n", rmini, rmaxi);
    return a;
  }

  for(int i=log; i>=0; i--) {
    if(pp[a][i] != -1 && pp[a][i] != pp[b][i]) {
      rmini = min(rmini, mini[a][i]);
      rmini = min(rmini, mini[b][i]);
      rmaxi = max(rmaxi, maxi[a][i]);
      rmaxi = max(rmaxi, maxi[b][i]);
      a = pp[a][i]; b = pp[b][i];
    }
  }

  rmini = min(rmini, mini[a][0]);
  rmini = min(rmini, mini[b][0]);
  rmaxi = max(rmaxi, maxi[a][0]);
  rmaxi = max(rmaxi, maxi[b][0]);
  printf("%d %d\n", rmini, rmaxi);
  return pai[a];
}

void res(int a, int b) {
  int px = resolve(a,b); // LCA
}

int main() {
  int a , b ,c,k;
  scanf("%d", &n);
  FOR(i,n) adj[i].clear();

  FOR(i,n-1) {
    scanf("%d %d %d", &a,&b,&c);
    a--;b--;
    adj[a].pb( no(b,c));
    adj[b].pb(no(a,c));
  }

  pre();
  scanf("%d", &k);
  FOR(i,k) {
    scanf("%d %d", &a,&b);
    a--;b--;
    res(a,b);
  }
  return 0;
}
```

.....

7.2. Árboles de Fenwick ó Binary indexed trees

Se tiene un arreglo $\{a_0, a_1, \cdots, a_{n-1}\}$. Los árboles de Fenwick permiten encontrar $\sum_{k=i}^{j} a_k$ en orden $O(\log_2 n)$ para parejas de (i, j) con $i \leq j$. De la misma manera, permiten sumarle una cantidad a un a_i también en tiempo $O(log_2 n)$.

```
// In this implementation, the tree is represented by a vector<int>.
// Elements are numbered by 0, 1, ..., n-1.
// tree[i] is sum of elements with indexes i&(i+1)..i, inclusive.
// (Note: this is a bit different from what is proposed
// in Fenwick's article.
// To see why it makes sense, think about the trailing 1's in binary
// representation of indexes.)

// Creates a zero-initialized Fenwick tree for n elements.
vector<int> create(int n) { return vector<int>(n, 0); }
// Returns sum of elements with indexes a..b, inclusive
int query(const vector<int> &tree, int a, int b) {
  if (a == 0) {
    int sum = 0;
    for (; b >= 0; b = (b & (b + 1)) - 1)
      sum += tree[b];
    return sum;
  } else {
    return query(tree, 0, b) - query(tree, 0, a-1);
  }
}
// Increases value of k-th element by inc.
void increase(vector<int> &tree, int k, int inc) {
  for (; k < (int)tree.size(); k |= k + 1)
    tree[k] += inc;
}
```

7.3. Segment tree

```
/*
  como cuestion util vale la pena mencionar que todo
  arbol binario se puede representar en un arreglo de
  la siguiente manera:
  la raiz es el elemento en posicion 0
  el hijo izquierdo de un nodo en posicion x es x * 2 + 1
  el hijo derecho de un nodo en posicion x es x * 2 + 2
  de esta manera no necesitamos usar mas que un arreglo para
  almacenar nuestro segment tree, que es un arbol binario
  este segment tree tiene dos operaciones
  update(pos, by): actualiza el arbol de manera que el elemento
  de la lista de numeros (no del arbol) en posicio pos
  aumente en un valor igual a by
  sum(from, to): determina la suma de los elementos en la
  lista de numeros en el intervalo [from, to]
*/
//numero maximo de elementos en la lista de numeros
#define N 200000
int tree[N * 4];
int n; //numero de elementos en el input
//actualiza en el arbol un valor de la lista de numeros
void update(int pos, int by){
  int node = 0, left = 0, right = n - 1;
  //intervalo inicial [0, n -1]
  while(left != right)
    //mientras tengamos que dividir el intervalo actual
    {
      tree[node] += by;
      int mid = (left + right) / 2;
      if(pos <= mid)
        {//hijo de la izquierda [left, mid]
          node = node * 2 + 1;
          right = mid;
        }else
        {//hijo de la derecha [mid + 1, right]
          node = node * 2 + 2;
          left = mid + 1;
        }
    }
  tree[node] += by;
}

/*
  determina la suma de los elementos del intervalo [from, to]
  la funcion es tal que node representa el intervalo [left, right]
  y [from, to] siempre es subconjunto de [left, right]
*/
int sum(int from, int to, int node = 0,
        int left = 0, int right = n - 1){
  if(from == left && to == right)
    // si el segmento [left, right] es parte de lo que queremos sumar
    return tree[node];
  int mid = (left + right) / 2;
  int res = 0;
  if(from <= mid) //si necesitamos ir por la izquierda
    res += sum(from, min(to, mid), node * 2 + 1, left, mid);
  if(to > mid) //si necesitamos ir por la derecha
    res += sum(max(mid + 1, from), to, node * 2 + 2, mid + 1, right);
  return res;
}
//inicializa todo el arbol con cero
int init(){
  for(int i = 0; i < 4 * n; i++) tree[i] = 0;
}
```

8. String Matching

8.1. KMP

```
// Computes the jumping function
vector<int> kmp_table(string &P){
  int i = 0, j = -1;
  int m = P.size();
  vector<int> f(m+1);
  f[0] = -1;
  while( i < m ){
    while(j>=0 and P[i] != P[j]) j = f[j];
    f[++i] = ++j;
  }
  return f;
}

void kmp(string &T, string &P){
  vector<int> pi = kmp_table(P);
  int n = T.size(), m = P.size();
  int q = 0;
```

```
  for(int i = 0; i<n; ){
    while(q > -1 and P[q] != T[i])  q = pi[q];
    i++; q++;
    if(q >= m ){
      any = true;
      printf("%d\n", i - q);
      q = pi[q];
    }
  }
}

int main(){
  int T, C=1;
  string s,t;
  while(scanf("%d", &T)){
    if(T==0) break;
    printf("Test case #%d\n", C++);
    cin >> s;
    vector<int>pi = kmp_table(s);
    for(int i=1;i<=T;++i)
      if(pi[i] > 0)
        if(i % (i-pi[i]) == 0)
          printf("%d %d\n", i, i/(i-pi[i]));
    puts("");
  }
  return 0;
}
```

.....

8.2. Suffix Arrays

```
const int N = 1000001;
// Begins Suffix Arrays implementation
// O(n log n) - Manber and Myers algorithm
//Usage:
// Fill str with the characters of the string.
// Call SuffixSort(n) where n = str.size()
// That's it!
//Output:
// pos = The suffix array. It has n suffixes
//       Contains the suffixes sorted in lexicographical order.
//       Each suffix is represented as a single integer
//       (the position of str where it starts).
// rank = The inverse of the suffix array.
//        rank[i] = the index of the suffix str[i..n)
//           in the pos array.
//        (In other words, pos[i] = k <==> rank[k] = i)
//        With this array, you can compare two suffixes in O(1):
//        Suffix str[i..n) is smaller
//        than str[j..n) if and only if rank[i] < rank[j]
// Use this like this!
// int n = s.size();
// for(int i=0;i<n;++i) str[i] = s[i];
// SuffixSort(n);
// getHeight(n);
int str[N]; //input
int rank[N], pos[N]; //output
int cnt[N], next[N]; //internal
bool bh[N], b2h[N];
bool smaller_first_char(int a, int b){
  return str[a] < str[b];
}
void SuffixSort(int n){
  for (int i=0; i<n; ++i){
    pos[i] = i;
  }
  sort(pos, pos + n, smaller_first_char);
  for (int i=0; i<n; ++i){
    bh[i] = i == 0 || str[pos[i]] != str[pos[i-1]];
    b2h[i] = false;
  }
  for (int h = 1; h < n; h <<= 1){
    int buckets = 0;
    for (int i=0, j; i < n; i = j){
      j = i + 1;
      while (j < n && !bh[j]) j++;
      next[i] = j;
      buckets++;
    }
    if (buckets == n) break;
    for (int i = 0; i < n; i = next[i]){
      cnt[i] = 0;
      for (int j = i; j < next[i]; ++j){
        rank[pos[j]] = i;
      }
    }
    cnt[rank[n - h]]++;
    b2h[rank[n - h]] = true;
    for (int i = 0; i < n; i = next[i]){
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0){
```

```
          int head = rank[s];
          rank[s] = head + cnt[head]++;
          b2h[rank[s]] = true;
        }
      }
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0 && b2h[rank[s]]){
          for (int k = rank[s]+1; !bh[k] && b2h[k]; k++)
            b2h[k] = false;
        }
      }
    }
    for (int i=0; i<n; ++i){
      pos[rank[i]] = i;
      bh[i] |= b2h[i];
    }
  }
  for (int i=0; i<n; ++i){
    rank[pos[i]] = i;
  }
}
// End of suffix array algorithm
int height[N];
// height[i] =
//  length of the LCP of suffix pos[i] and pos[i-1]
// height[0] = 0
void getHeight(int n){
  for (int i=0; i<n; ++i) rank[pos[i]] = i;
  height[0] = 0;
  for (int i=0, h=0; i<n; ++i){
    if (rank[i] > 0){
      int j = pos[rank[i]-1];
      while (i + h < n && j + h < n && str[i+h] == str[j+h]) h++;
      height[rank[i]] = h;
      if (h > 0) h--;
    }
  }
}
// lcp(x,y) = min(lcp(x,x+1), lcp(x+1, x+2), ... , lcp(y-1, y))
string s;
// You need a string W that represents the pattern
// Not really tested. Pseudo-tested
int match_prefix(int n){
  string W; // Fill this outside
  if(W[0] < s[pos[0]]) return -1; // Is not here!
  if(W[0] > s[pos[n-1]]) return -1; // Not here too!
  if(W == s.substr(pos[0])) return pos[0];
  // Binary search for the W pattern
  int l = 0, r = n-1, m;
  while(r-l > 1){
    m = (l+r)/2;
    if(W >= s.substr(pos[m]))
      l = m;
    else
      r = m;
  }
  // r is the i-sth smallest suffix
  // that means that pos[r] is the actual index
  if(W != s.substr(pos[r], W.size())) return -1; // not here at all!
  printf("Matched at %d\n", r);
  return pos[r];
}
void get_the_biggest_repeated_substring(){
  // Get the biggest repeated substring and how many times it appears
  // First, get the biggest repeated string (biggest height[i])
  // Then count it's repetitions
  // GATTACA
  for(int i=1;i<n;++i){
    if(longest < height[i]){longest = height[i]; position = i - 1;}
  }
}
void number_of_different_substrings(){
  // If you have the i-th smaller suffix, Si,
  //    it's length will be |Si| = n - pos[i]
  // Now, height[i] stores the number of
  //     common letters between Si and Si
  //     (s.substr(pos[i]) and s.substr(pos[i-1]))
  // so, you have |Si| - height[i] different strings
  //    from these two suffixes => n - pos[i] - height[i]
  for(int i=0;i<n;++i) ans += n-pos[i]-height[i];
}
void number_of_repeated_substrings(){
  // Number of substrings that appear at least twice in the text.
  // The trick is that all 'spare' substrings that can give us
  // Lcp(i - 1, i) can be obtained by Lcp(i - 2, i - 1)
  // due to the ordered nature of our array.
  // And the overall answer is
  // Lcp(0, 1) +
  //     Sum(max[0, Lcp(i, i - 1) - Lcp(i - 2, i - 1)])
  //     for 2 <= i < n
  // File Recover
```

```
  int cnt = height[1];
  for(int i=2;i<n;++i){
    cnt += max(0, height[i] - height[i-1]);
  }
}
void repeated_m_times(int m){
  // Given a string s and an int m, find the size
  // of the biggest substring repeated m times (find the rightmost pos)
  // if a string is repeated m+1 times, then it's repeated m times too
  // The answer is the maximum, over i, of the longest common prefix
  // between suffix i+m-1 in the sorted array.
  int length = 0, position = -1, t;
  for(int i=0;i<=n-m;++i){
    if((t=lcp(i,i+m-1,n)) > length){
      length = t;
      position = pos[i];
    }else if(t == length) { position = max(position, pos[i]); }
  }
  // Here you'll get the rightmost position
  // (that means, the last time the substring appears)
  for(int i = 0; i < n; ){
    if(pos[i] + length > n) {++i; continue;}
    int ps = 0, j = i+1;
    while(j<n && height[j] >= length){
      ps = max(ps,pos[j]);
      j++;
    }
    if(j - i >= m) position = max(position, ps);
    i = j;
  }
  if(length != 0)
    printf("%d %d\n", length, position);
  else
    puts("none");
}
void smallest_rotation(){
  // Reads a string of lenght k. Then just double it (s = s+s)
  // and find the suffix arrays.
  // The answer is the smallest i for which s.size() - pos[i] >= k
  // If you want the first appearence (and not the string)
  //   you'll need the second cycle
  int best = 0;
  for(int i=0;i<n;++i){
    if(n - pos[i] >= k){
      //Find the first appearence of the string
      while( n - pos[i] >= k){
        if(pos[i] < pos[best] && pos[i]!=0) best = i;
        i++;
      }
      break;
    }
  }
  if(pos[best] == k) puts("0");
  else printf("%d\n", pos[best]);
}
```

.....