Manual de algoritmos para maratones de programación

Just Code It.

6 de septiembre de 2014

Índice

1. Plantilla

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <numeric>
#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>
#include <bitset>

#define D(x) cout << "DEBUG: " << #x " = " << x << endl

using namespace std;

const double EPS = 1e-9;
const double PI = acos(-1.0);

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; }

int
main() {

  return 0;
}
```

.....

2. Cosas para tener en cuenta

- Si la respuesta de un problema es un número de punto flotante redondeado y está dando $WrongAnswer$, ensayar sumarle un épsilon a la respuesta, es decir, sumarle EPS (siendo EPS por lo general 1e-9).
- Recordar que para redondear un número de punto flotante se debe usar el parámetro $\%.xf$ (donde x es la cantidad de cifras) en la función $printf$.
- En problemas que se trabajen números de punto flotante y enteros a la vez, es recomendable mutiplicar por 1.0 cuando se estén haciendo operaciones con ambos tipos de datos. Con esto se evitarán problemas de conversión.

3. Grafos

3.1. BFS

Algoritmo de recorrido de grafos en anchura que empieza desde una fuente s y visita todos los nodos alcanzables desde s.
El BFS también halla la distancia más corta entre s y los demás nodos si las aristas tienen todas peso 1.
Complejidad: $O(n + m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN]; // La lista de adyacencia
int d[MAXN];          // Distancia de la fuente a cada nodo

void bfs(int s, int n){ // s = fuente, n = número de nodos
   for (int i = 0; i <= n; ++i) d[i] = -1;

   queue <int> q;
   q.push(s);
   d[s] = 0;
   while (q.size() > 0){
      int cur = q.front();
      q.pop();
      for (int i = 0; i < g[cur].size(); ++i){
         int next = g[cur][i];
         if (d[next] == -1){
            d[next] = d[cur] + 1;
            q.push(next);
         }
      }
   }
}
```

3.2. DFS

Algoritmo de recorrido de grafos en profundidad que empieza visita todos los nodos del grafo.
El algoritmo puede ser modificado para que retorne información de los nodos según la necesidad del problema.
El grafo tiene un ciclo $\leftrightarrow$ si en algún momento se llega a un nodo marcado como gris.
Complejidad: $O(n + m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN];      // La lista de adyacencia
int color[MAXN];           // El arreglo de visitados
enum {WHITE, GRAY, BLACK}; // WHITE = 1, GRAY = 2, BLACK = 3

// Visita el nodo u y todos sus vecinos empezando por
// los más profundos
void dfs(int u){
   color[u] = GRAY;   // Marcar el nodo como semi-visitado
   for (int i = 0; i < g[u].size(); ++i){
     int v = g[u][i];
     if (color[v] == WHITE) dfs(v);  // Visitar los vecinos
   }
   color[u] = BLACK;  // Marcar el nodo como visitado
}

// Llama la función dfs para los nodos 0 a n-1
void call_dfs(int n){
   for (int u = 0; u < n; ++u) color[u] = WHITE;
   for (int u = 0; u < n; ++u)
     if (color[u] == WHITE) dfs(u);
}
```

3.3. Ordenamiento topológico

Dado un grafo no cíclico y dirigido (DAG), ordena los nodos linealmente de tal forma que si existe una arista entre los nodos u y v entonces u aparece antes que v en el ordenamiento.
Este ordenamiento se puede ver como una forma de poner todos los nodos en una línea recta y que las aristas vayan todas de izquierda a derecha.
Complejidad: $O(n + m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN];    // La lista de adyacencia
bool seen[MAXN];         // El arreglo de visitados para el dfs
vector <int> topo_sort;  // El vector del ordemamiento

void dfs(int u){
   seen[u] = true;
```

```
   for (int i = 0; i < g[u].size(); ++i){
      int v = g[u][i];
      if (!seen[v]) dfs(v);
   }
   topo_sort.push_back(u); // Agregar el nodo al ordenamiento
}
void topological(int n){   // n = número de nodos
   topo_sort.clear();
   for (int i = 0; i < n; ++i) seen[i] = false;
   for (int i = 0; i < n; ++i) if (!seen[i]) dfs(i);
   reverse(topo_sort.begin(), topo_sort.end());
}
```

3.4. Componentes fuertemente conexas

Dado un grafo dirigido, calcula la componente fuertemente conexa (SCC) a la que pertenece cada nodo.
Para cada pareja de nodos u, v que pertenecen a una misma SCC se cumple que hay un camino de u a v y de v a u.
Si se comprime el grafo dejando como nodos cada una de las componentes se quedará con un DAG.

3.4.1. Kosaraju's algorithm

Complejidad: $O(n+m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN];    // El grafo
vector <int> grev[MAXN]; // El grafo con las aristas reversadas
vector <int> topo_sort;  // El "ordenamiento topologico" del grafo
int scc[MAXN];   // La componente a la que pertenece cada nodo
bool seen[MAXN]; // El arreglo de visitado para el primer DFS

// DFS donde se halla el ordenamiento topológico
void dfs1(int u){
   seen[u] = true;
   for (int i = 0; i < g[u].size(); ++i){
      int v = g[u][i];
      if (!seen[v]) dfs1(v);
   }
   topo_sort.push_back(u);
}
// DFS donde se hallan las componentes
void dfs2(int u, int comp){
   scc[u] = comp;
   for (int i = 0; i < grev[u].size(); ++i){
      int v = grev[u][i];
      if (scc[v] == -1) dfs2(v, comp);
   }
}

// Halla las componentes fuertemente conexas del grafo usando
// el algoritmo de Kosaraju. Retorna la cantidad de componentes
int find_scc(int n){ // n = número de nodos
   // Crear el grafo reversado
   for (int u = 0; u < n; ++u){
      for (int i = 0; i < g[u].size(); ++i){
         int v = g[u][i];
         grev[v].push_back(u);
      }
   }

   // Llamar el primer dfs
   for (int i = 0; i < n; ++i){
      if (!seen[i]) dfs1(i);
   }
   reverse(topo_sort.begin(), topo_sort.end());

   // Llamar el segundo dfs
   int comp = 0;
   for (int i = 0; i < n; ++i){
      int u = topo_sort[i];
      if (scc[u] == -1) dfs2(u, comp++);
   }
   return comp;
}
```

3.4.2. Tarjan's algorithm

- $d[i]$ = Tiempo de descubrimiento del nodo i. (Inicializar con -1)
- $low[i]$ = Menor tiempo de descubrimiento alcanzable desde el nodo i. (No

inicializar)

- $scc[i]$ = Componente a la que pertenece el nodo i. (No inicializar)
- s = Pila usada por el algoritmo (Inicializar vacía)
- $stacked[i]$ = true si el nodo i está en la pila. (Inicializar con falso)
- $ticks$ = Reloj usado para los tiempos de descubrimiento (Inicializar en 0)
- $current_scc$ = id de la componente actual siendo descubierta. (Inicializar en 0)

```
vector <int> g[MAXN];
int d[MAXN], low[MAXN], scc[MAXN];
bool stacked[MAXN];
stack <int> s;
int ticks, current_scc;

// Check initializations.
void tarjan(int u) {
  d[u] = low[u] = ticks++;
  s.push(u);
  stacked[u] = true;
  for (int k = 0; k < g[u].size(); ++k) {
    int v = g[u][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++;
  }
}
```

3.5. Algoritmo de Dijkstra

Dado un grafo con pesos **no negativos** en las aristas, halla la mínima distancia entre una fuente s y los demás nodos.
Al heap se inserta primero la distancia y luego en nodo al que se llega. Si se quieren modificar los pesos por `long long` o por `double` se debe cambiar en los tipos de dato `dist_node` y `edge`.
Complejidad: $O((n+m)\log n)$ donde n es el número de nodos y m es el número de aristas.

```
const int MAXN = 100005;
const int INF = 1 << 30;      // Usar 1LL << 60 para long long
typedef pair <int, int> dist_node; // Datos del heap (dist, nodo)
typedef pair <int, int> edge; // Dato de las arista (nodo, peso)
vector <edge> g[MAXN];        // g[u] = (v = nodo, w = peso)
int d[MAXN];    // d[u] La distancia más corta de s a u
int p[MAXN];    // p[u] El predecesor de u en el camino más corto

// La función recibe la fuente s y el número total de nodos n
void dijkstra(int s, int n){
   for (int i = 0; i <= n; ++i){
      d[i] = INF;  p[i] = -1;
   }
   priority_queue < dist_node, vector <dist_node>,
                    greater<dist_node> > q;
   d[s] = 0;
   q.push(dist_node(0, s));
   while (!q.empty()){
      int dist = q.top().first;
      int cur = q.top().second;
      q.pop();
      if (dist > d[cur]) continue;
      for (int i = 0; i < g[cur].size(); ++i){
         int next = g[cur][i].first;
         int w_extra = g[cur][i].second;
         if (d[cur] + w_extra < d[next]){
            d[next] = d[cur] + w_extra;
            p[next] = cur;
            q.push(dist_node(d[next], next));
         }
      }
   }
}
```

```
// La función que retorna los nodos del camino más corto de s a t
// Primero hay que correr dijktra desde s.
// Eliminar si no se necesita hallar el camino.
vector <int> find_path (int t){
   vector <int> path;
   int cur = t;
   while(cur != -1){
      path.push_back(cur);
      cur = p[cur];
   }
   reverse(path.begin(), path.end());
   return path;
}
```

3.6. Algoritmo de Bellman-Ford

Dado un grafo con pesos cualquiera, halla la mínima distancia entre una fuente s y los demás nodos.
Si hay un ciclo de peso negativo en el grafo, el algoritmo lo indica.
Complejidad: $O(n \times m)$ donde n es el número de nodos y m es el número de aristas.
Tener en cuenta que si el nodo es inalcanzable la distancia que resulta en dicho nodo siempre será infinito.
Si el problema es como *Haunted Graveyard*, donde los niños querian salir del cementerio lo más rápido posible (no querian quedarse dando vueltas así fueran ciclos negativos) entonces no deberia poner aristas en el nodo de salida.

```
const int MAXN = 105;
const int INF = 1 << 30;  // Para long long INF = 1LL << 60
typedef pair <int, int> edge; // Modificar según el problema
vector <edge> g[MAXN]; // g[u] = (v = nodo, w = peso)
int d[MAXN];           // d[u] = distancia más corta de s a u

// Retorna verdadero si el grafo tiene un ciclo de peso negativo
// alcanzable desde s y falso si no es así.
// Al finalizar el algoritmo, si no hubo ciclo de peso negativo,
// la distancia más corta entre s y u está almacenada en d[u]
bool bellman_ford(int s, int n){ // s = fuente, n = número nodos
   for (int u = 0; u <= n; ++u) d[u] = INF;
   d[s] = 0;
   for (int i = 1; i <= n - 1; ++i){
      for (int u = 0; u < n; ++u){
         for (int k = 0; k < g[u].size(); ++k){
            int v = g[u][k].first;
            int w = g[u][k].second;
            d[v] = min(d[v], d[u] + w);
         }
      }
   }

   for (int u = 0; u < n; ++u){
      for (int k = 0; k < g[u].size(); ++k){
         int v = g[u][k].first;
         int w = g[u][k].second;
         if (d[v] > d[u] + w) return true;
      }
   }
   return false;
}
```

3.7. Algoritmo de Floyd-Warshall

Dado un grafo con pesos cualquiera, halla la mínima distancia entre cualquier para de nodos.
Si este algoritmo es muy lento para el problema ejecutar n veces el algoritmo de Dijkstra o de Bellman-Ford según el caso.
Complejidad: $O(n^3)$ donde n es el número de nodos.

Casos base: `d[i][j]` $= \begin{cases} 0 & \text{si } i = j \\ w_{i,j} & \text{si existe una arista entre } i \text{ y } j \\ +\infty & \text{en otro caso} \end{cases}$

Nota: Utilizar el tipo de dato apropiado (`int`, `long long`, `double`) para `d` y para $+\infty$ según el problema.

```
// Los nodos están numerados de 0 a n-1
for (int k = 0; k < n; ++k){
   for (int i = 0; i < n; ++i){
      for (int j = 0; j < n; ++j){
         d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
```

```
      }
   }
}
// Acá d[i][j] es la mínima distancia entre el nodo i y el j
```

.....

3.7.1. Clausura transitiva

Dado un grafo cualquiera, hallar si existe un camino desde i hasta j para cualquier pareja de nodos i, j

Casos base: `d[i][j]` $= \begin{cases} \texttt{true} & \text{si } i = j \\ \texttt{true} & \text{si existe una arista entre } i \text{ y } j \\ \texttt{false} & \text{en otro caso} \end{cases}$

Caso recursivo: `d[i][j] = d[i][j] or (d[i][k] and d[k][j]);`

3.7.2. Minimax

Dado un grafo con pesos, hallar el camino de i hasta j donde la arista más grande del camino sea lo más pequeña posible.
Ejemplos: Que el peaje más caro sea lo más barato posible, que la autopista más larga sea lo más corta posible.

Casos base: `d[i][j]` $= \begin{cases} 0 & \text{si } i = j \\ w_{i,j} & \text{si existe una arista entre } i \text{ y } j \\ +\infty & \text{en otro caso} \end{cases}$

Caso recursivo: `d[i][j] = min( d[i][j], max (d[i][k], d[k][j]) );`

3.7.3. Maximin

Dado un grafo con pesos, hallar el camino de i hasta j donde la arista más pequeña del camino sea lo más grande posible.
Ejemplos: Que el trayecto menos seguro sea lo más seguro posible, que la autopista de menos carriles tenga la mayor cantidad de carriles.

Casos base: `d[i][j]` $= \begin{cases} +\infty & \text{si } i = j \\ w_{i,j} & \text{si existe una arista entre } i \text{ y } j \\ -\infty & \text{en otro caso} \end{cases}$

Caso recursivo: `d[i][j] = max( d[i][j] , min(d[i][k], d[k][j]) )`

3.8. Algoritmo de Prim

Dado un grafo no dirigido y conexo, retorna el costo del árbol de mínima expansión de ese grafo.
El costo del árbol de mínima expansión también se puede ver como el mínimo costo de las aristas de manera que haya un camino entre cualquier par de nodos.
Complejidad: $O(m \log n)$ donde n es el número de nodos y m es el número de aristas.

```
const int MAXN = 10005;
typedef pair <int, int> edge;        // Pareja (nodo, peso)
typedef pair <int, int> weight_node; // Pareja (peso, nodo)
vector <edge> g[MAXN];               // Lista de adyacencia
bool visited[MAXN];

// Retorna el costo total del MST
int prim(int n){ // n = número de nodos
    for (int i = 0; i <= n; ++i) visited[i] = false;
    int total = 0;

    priority_queue<weight_node, vector <weight_node>,
                   greater<weight_node> > q;

    // Empezar el MST desde 0 (cambiar si el nodo 0 no existe)
    q.push(weight_node(0, 0));
    while (!q.empty()){
        int u = q.top().second;
        int w = q.top().first;
        q.pop();
        if (visited[u]) continue;

        visited[u] = true;
        total += w;
        for (int i = 0; i < g[u].size(); ++i){
            int v = g[u][i].first;
            int next_w = g[u][i].second;
            if (!visited[v]){
                q.push(weight_node(next_w, v));
            }
        }
    }
    return total;
```

```
}
```

3.9. Algoritmo de Kruskal

3.9.1. Union-Find

Union-Find es una estructura de datos para almacenar una colección conjuntos disjuntos (no tienen elementos en común) que cambian dinámicamente. Identifica en cada conjunto un "padre" que es un elemento al azar de ese conjunto y hace que todos los elementos del conjunto "apunten" hacia ese padre.
Inicialmente se tiene una colección donde cada elemento es un conjunto unitario.
Complejidad aproximada: $O(m)$ donde m el número total de operaciones de `initialize`, `union` y `join` realizadas.

```
const int MAXN = 100005;
int p[MAXN];// El padre del conjunto al que pertenece cada nodo

// Inicializar cada conjunto como unitario
void initialize(int n){
  for (int i = 0; i <= n; ++i) p[i] = i;
}

// Encontrar el padre del conjunto al que pertenece u
int find(int u){
  if (p[u] == u) return u;
  return p[u] = find(p[u]);
}

// Unir los conjunto a los que pertenecen u y v
void join(int u, int v){
  int a = find(u);
  int b = find(v);
  if (a == b) return;
  p[a] = b;
}
```

3.9.2. Algoritmo de Kruskal

Dado un grafo no dirigido y conexo, retorna el costo del árbol de mínima expansión de ese grafo.
El costo del árbol de mínima expansión también se puede ver como el mínimo costo de las aristas de manera que haya un camino entre cualquier par de nodos.
Utiliza Union-Find para ver rápidamente qué aristas generan ciclos.
Complejidad: $O(m \log n)$ donde n es el número de nodos y m es el número de aristas.

```
struct edge{
   int start, end, weight;

   edge(int u, int v, int w){
      start = u; end = v; weight = w;
   }
   bool operator < (const edge &other) const{
      return weight < other.weight;
   }
};

const int MAXN = 100005;
vector <edge> edges; // Lista de aristas y no lista de adyacencia
int p[MAXN];         // El padre de cada conjunto (union-find)

// Incluir las operaciones de Union-Find (initialize, find, join)

int kruskal(int n){
    initialize(n);
    sort(edges.begin(), edges.end());

    int total = 0;
    for (int i = 0; i < edges.size(); ++i){
        int u = edges[i].start;
        int v = edges[i].end;
        int w = edges[i].weight;
        if (find(u) != find(v)){
            total += w;
            join(u, v);
        }
    }
    return total;
```

```
}
```

.....

3.10. Algoritmo de máximo flujo

Dado un grafo con capacidades enteras, halla el máximo flujo entre una fuente s y un sumidero t.
Como el máximo flujo es igual al mínimo corte, halla también el mínimo costo de cortar aristas de manera que s y t queden desconectados.
Si hay varias fuentes o varios sumideros poner una súper-fuente / súper-sumidero que se conecte a las fuentes / sumideros con capacidad infinita.
Si los nodos también tienen capacidad, dividir cada nodo en dos nodos: uno al que lleguen todas las aristas y otro del que salgan todas las aristas y conectarlos con una arista que tenga la capacidad del nodo.
Complejidad: $O(n \cdot m^2)$ donde n es el número de nodos y m es el número de aristas.

```
const int MAXN = 105;
// Lista de adyacencia de la red residual
vector <int> g [MAXN];
// Capacidad de aristas de la red de flujos
int c [MAXN][MAXN];
// El flujo de cada arista
int f [MAXN][MAXN];
//El predecesor de cada nodo en el camino de aumentación de s a t
int prev [MAXN];

void connect (int i, int j, int cap){
    // Agregar SIEMPRE las dos aristas a g (red residual) así el
    // grafo sea dirigido. Esto es porque g representa la red
    // residual que tiene aristas en los dos sentidos.
    g[i].push_back(j);
    g[j].push_back(i);
    c[i][j] += cap;
    // Omitir esta línea si el grafo es dirigido
    c[j][i] += cap;
}

// s = fuente, t = sumidero, n = número de nodos
int maxflow(int s, int t, int n){
    for (int i = 0; i <= n; i++){
        for (int j = 0; j <= n; j++){
            f[i][j] = 0;
        }
    }

    int flow = 0;
    while (true){
        for (int i = 0; i <= n; i++) prev[i] = -1;

        queue <int> q;
        q.push(s);
        prev[s] = -2;

        while (q.size() > 0){
            int u = q.front(); q.pop();
            if (u == t) break;
            for (int i = 0; i < g[u].size(); ++i){
                int v = g[u][i];
                if (prev[v] == -1 and c[u][v] - f[u][v] > 0){
                    q.push(v);
                    prev[v] = u;
                }
            }
        }
        if (prev[t] == -1) break;

        int extra = 1 << 30;
        int end = t;
        while (end != s){
            int start = prev[end];
            extra = min(extra, c[start][end] - f[start][end]);
            end = start;
        }

        end = t;
        while (end != s){
            int start = prev[end];
            f[start][end] += extra;
            f[end][start] = -f[start][end];
            end = start;
        }

        flow += extra;
    }
```

```
    return flow;
}
```

.....

3.11. Teorema de Konig

3.11.1. Ejemplo, aplicación Max Flow - Mimimum Vertex Cover - Maximum Matching Size

El algoritmo de máximo flujo es equivalente al problema de $MinimumVertexCover$ en un grafo bipartito, pero como el Vertex Cover es NP-hard, entonces el teorema de $Konig$ nos dice que el $MinimumVertexCover$ es igual al $MaximumMatchingSize$, el cual puede ser encontrado por un máximo flujo estándar.
A continuación se presenta un problema para esta aplicación.

As an example, one year there were six interns: Stephen Cook, Vinton Cerf, Edmund Clarke, Judea Pearl, Shafi Goldwasser, and Silvio Micali. They were able to self-organize into three teams:

- Stephen Cook, Vinton Cerf, and Edmund Clarke (whose last names all begin with C)
- Shafi Goldwasser and Silvio Micali (whose first names begin with S)
- Judea Pearl (not an interesting group, but everyone's first name in this group starts with J)

As a historical note, the company was eventually shut down due to a rather strange (and illegal) hiring practice—they refused to hire any interns whose last names began with the letter S, T, U, V, W, X, Y, or Z. (First names were not subject to such a whim, which was fortunate for our friend Vinton Cerf.)
Input: Each year's group of interns is considered as a separate trial. A trial begins with a line containing a single integer N, such that $1 \leq N \leq 300$, designating the number of interns that year. Following that are N lines—one for each intern—with a line having a first and last name separated by one space. Names will not have any punctuation, and both the first name and last name will begin with an uppercase letter. In the case of last names, that letter will have an additional constraint that it be in the range from 'A' to 'R' inclusive. The end of the input is designated by a line containing the value 0. There will be at most 20 trials.
Output: For each trial, output a single integer, k, designating the minimum number of teams that were necessary.

La entrada y saida ejemplo es:

Example Input:	Example Output:
6 Stephen Cook Vinton Cerf Edmund Clarke Judea Pearl Shafi Goldwasser Silvio Micali 9 Richard Hamming Marvin Minskey John McCarthy Edsger Dijkstra Donald Knuth Michael Rabin John Backus Robert Floyd Tony Hoare 0	3 6

Entonces la solución sería, utilizando $MaxFlow$:

```
const int MAXN = 55; //26 letras nombres +
                     //26 letras apellidos + 3
int n;
vector <int> g[MAXN];
int c[MAXN][MAXN];
int f[MAXN][MAXN];
int prev[MAXN];

void
connect (int i, int j) {
    g[i].push_back(j);
    g[j].push_back(i);
    c[i][j] = 1;
}
```

```
int
maxflow(int s, int t) {
    for (int i = 0; i < MAXN; i++) {
        for (int j = 0; j < MAXN; j++) {
            f[i][j] = 0;
        }
    }

    int flow = 0;
    while (true) {
        for (int i = 0; i < MAXN; i++) prev[i] = -1;
        queue <int> q;
        q.push(s);
        prev[s] = -2;

        while (!q.empty()) {
            int u = q.front(); q.pop();
            if (u == t) break;
            for (int i = 0; i < g[u].size(); i++) {
                int v = g[u][i];
                if (prev[v] == -1 and c[u][v] - f[u][v] > 0) {
                    q.push(v);
                    prev[v] = u;
                }
            }
        }
        if (prev[t] == -1) break;

        int extra = 1 << 30;
        int end = t;
        while (end != s) {
            int start = prev[end];
            extra = min(extra, c[start][end] - f[start][end]);
            end = start;
        }
        end = t;
        while (end != s) {
            int start = prev[end];
            f[start][end] += extra;
            f[end][start] = -f[start][end];
            end = start;
        }
        flow += extra;
    }
    return flow;
}

void
limpiar() {
    for (int i = 0; i < MAXN; i++) {
        g[i].clear();
    }
}

int
main() {
    while (cin >> n && n) {
        limpiar();
        for (int i = 0; i < n; i++) {
            string nombre, apellido;
            cin >> nombre >> apellido;
            if (apellido[0] == 'S' || apellido[0] == 'T' ||
                apellido[0] == 'U' || apellido[0] == 'V' ||
                apellido[0] == 'W' || apellido[0] == 'X' ||
                apellido[0] == 'Y' || apellido[0] == 'Z') continue;
            int nodoNombre, nodoApellido;
            nodoNombre = nombre[0] - 'A' + 1; //A = 1
            nodoApellido = apellido[0] - 'A' + 27; //Z = 52
            connect(nodoNombre, nodoApellido);
            connect(0, nodoNombre);
            connect(nodoApellido, 53);
        }
        cout << maxflow(0, 53) << endl;
    }
  return 0;
}
```

.....

4. Teoría de números

4.1. Números romanos

A continuación están las funciones para pasar del sistema romano a árabe. Valores: I = 1, V = 5, X = 10, L = 50, C = 100, D = 500 y M = 1000;

4.1.1. Árabe a Romano

OJO: Tener en cuenta que el rango de conversión es 1 - 3999

```
string
arabicToRoman(int num) {
  string uni[10] = {"", "I", "II", "III", "IV", "V",
                    "VI", "VII", "VIII", "IX"};
  string deci[10]={"", "X", "XX", "XXX", "XL", "L",
                   "LX", "LXX", "LXXX", "XC"};
  string cen[10]={"", "C", "CC", "CCC", "CD", "D",
                  "DC", "DCC", "DCCC", "CM"};
  string mil[4]={"", "M", "MM", "MMM"};
  int nUni = num % 10;
  int nDec = (num / 10) % 10;
  int nCen = ((num / 10) / 10) % 10;
  int nMil = (((num / 10) / 10) / 10) % 10;
  string ans = mil[nMil];
  ans += cen[nCen];
  ans += deci[nDec];
  ans += uni[nUni];
  return ans;
}
```

4.1.2. Romano a Árabe

OJO: Tener en cuenta que el rango de conversión es 1 - 3999

```
int
romanToArabic(string num) {
  map <char, int> RtoA;
  RtoA['I'] = 1;  RtoA['V'] = 5;  RtoA['X'] = 10; RtoA['L'] = 50;
  RtoA['C'] = 100;RtoA['D'] = 500;RtoA['M'] = 1000;

  int value = 0;
  for (int i = 0; num[i]; i++) {
    if (num[i+1] && RtoA[num[i]] < RtoA[num[i+1]]) {
      value += RtoA[num[i+1]] - RtoA[num[i]];
      i++;
    }
    else value += RtoA[num[i]];
  }
  return value;
}
```

4.2. Divisores de un número

Imprime los divisores de un número (cuidado que no lo hace en orden).
Complejidad: $O(\sqrt{n})$ donde n es el número.

```
void divisors(int n){
    int i;
    for (i = 1; i * i < n; ++i){
        if (n % i == 0) printf("%d\n%d\n", i, n/i);
    }
    // Si existe, imprimir su raiz cuadrada una sola vez
    if (i * i == n) printf("%d\n", i);
}
```

4.3. Máximo común divisor y mínimo común múltiplo

Para hallar el máximo común divisor entre dos números `a` y `b` ejecutar el comando `__gcd(a, b)`.

Para hallar el mínimo común múltiplo: $\text{lcm}(a,b) = \dfrac{|a \cdot b|}{\gcd(a,b)}$

4.4. Criba de Eratóstenes

Encuentra los primos desde 1 hasta un límite n.
`sieve[i]` es falso sí y solo sí i es un número primo.
Complejidad: $O(n)$ donde n es el límite superior.

```
const int MAXN = 1000000;
bool sieve[MAXN + 5];
vector <int> primes;

void build_sieve(){
    memset(sieve, false, sizeof(sieve));
    sieve[0] = sieve[1] = true;

    for (int i = 2; i * i <= MAXN; i ++){
        if (!sieve[i]){
```

```
            for (int j = i * i; j <= MAXN; j += i){
                sieve[j] = true;
            }
        }
    }
    for (int i = 2; i <= MAXN; ++i){
        if (!sieve[i]) primes.push_back(i);
    }
}
```

4.5. Factorización prima de un número

Halla la factorización prima de un número a positivo. Si a es negativo llamar el algoritmo con $|a|$ y agregarle -1 a la factorización.
Se asume que ya se ha ejecutado el algoritmo para generar los primos hasta al menos $\sqrt{a}$.
El algoritmo genera la lista de primos en orden de menor a mayor.
Utiliza el hecho de que en la factorización prima de a aparece máximo un primo mayor a $\sqrt{a}$.
Complejidad aproximada: $O(\sqrt{a})$

```
const int MAXN = 1000000; // MAXN > sqrt(a)
bool sieve[MAXN + 5];
vector <int> primes;

vector <long long> factorization(long long a){
    // Se asume que se tiene y se llamó la función build_sieve()
    vector <long long> ans;
    long long b = a;
    for (int i = 0; 1LL * primes[i] * primes[i] <= a; ++i){
        int p = primes[i];
        while (b % p == 0){
            ans.push_back(p);
            b /= p;
        }
    }
    if (b != 1) ans.push_back(b);
    return ans;
}
```

4.6. Exponenciación logarítmica

4.6.1. Propiedades de la operación módulo

- $(a \bmod n) \bmod n = a \bmod n$
- $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$
- $(a \cdot b) \bmod n = ((a \bmod n) \cdot (b \bmod n)) \bmod n$
- $\left(\frac{a}{b}\right) \bmod n \neq \left(\frac{a \bmod n}{b \bmod n}\right) \bmod n$

4.6.2. Big mod

Halla rápidamente el valor de $b^p \bmod m$ para $0 \leq b, p, m \leq 2147483647$
Si se cambian los valores por `long long` los límites se cambian por $0 \leq b, p \leq 9223372036854775807$ y $1 \leq m \leq 3037000499$.
Complejidad: $O(\log p)$

```
int bigmod(int b, int p, int m){
    if (p == 0) return 1;
    if (p % 2 == 0){
        int mid = bigmod(b, p/2, m);
        return (1LL * mid * mid) % m;
    }else{
        int mid = bigmod(b, p-1, m);
        return (1LL * mid * b) % m;
    }
}
```

4.7. Combinatoria

4.7.1. Coeficientes binomiales

Halla el valor de $\binom{n}{k}$ para $0 \leq k \leq n \leq 66$. Para $n > 66$ los valores comienzan a ser muy grandes y no caben en un `long long`.
Complejidad: $O(n^2)$

```
const int MAXN = 66;
unsigned long long choose[MAXN+5][MAXN+5];

void binomial(int N){
```

```
    for (int n = 0; n <= N; ++n) choose[n][0] = choose[n][n] = 1;

    for (int n = 1; n <= N; ++n){
        for (int k = 1; k < n; ++k){
            choose[n][k] = choose[n-1][k-1] + choose[n-1][k];
        }
    }
}
```

.....

4.7.2. Propiedades de combinatoria

- El número de permutaciones de n elementos diferentes es $n!$
- El número de permutaciones de n elementos donde hay m_1 elementos repetidos de tipo 1, m_2 elementos repetidos de tipo 2, ..., m_k elementos repetidos de tipo k es

$$\frac{n!}{m_1!m_2!\cdots m_k!}$$

- El número de permutaciones de k elementos diferentes tomados de un conjunto de n elementos es

$$\frac{n!}{(n-k)!} = k!\binom{n}{k}$$

5. Programación dinámica

5.1. Longest increasing subsequence

Halla la longitud de la subsecuencia creciente más larga que hay en un arreglo (también se puede usar con strings).

5.1.1. Orden cuadrático

Retorna un **vector con los elementos** que componen la subsecuencia creciente más larga del arreglo arr.
Si se necesita sólo la longitud de la subsecuencia ignorar las líneas que tienen comentado un $*$ y retornar lo que se requiera.
Complejidad: $O(n^2)$ donde n es la longitud de arr.

```
const int MAXN = 1005;
int dp[MAXN];
int prev[MAXN]; //*
int arr[MAXN];

vector <int>
lis(int n) {
  int maxi = 1;
  int indexMaxi = 0; //*
  dp[0] = 1;
  prev[0] = -1; //*
  for (int i = 1; i < n; i++) {
    dp[i] = 1;
    prev[i] = -1; //*
    for (int j = i - 1; j >= 0; j--) {
      //>   estrictamente creciente
      //>=  dos elementos iguales son ambos incluídos
      if (arr[i] > arr[j] && dp[j] + 1 > dp[i]) {
        dp[i] = dp[j] + 1;
        prev[i] = j; //*
        if (dp[i] > maxi) {
          maxi = dp[i];
          indexMaxi = i; //*
        }
      }
    }
  }
  //Si se necesita sólo el tamaño retornar maxi aquí
  vector <int> seq;
  for (int i = indexMaxi; i >= 0; i = prev[i]) {
    seq.push_back(arr[i]);
  }
  reverse(seq.begin(), seq.end());
  return seq;
}
```

.....

5.1.2. Orden logarítmico imprimiendo sólo longitud

Retorna el **tamaño** de la longitud de la subsecuencia creciente más larga del arreglo arr.
Complejidad: $O(n \log_2 n)$ donde n es la longitud de arr.

```
const int MAXN = 100005;
int arr[MAXN]; //Almacena los elementos
int dp[MAXN]; //Almacena la secuencia creciente más larga
```

```
int
binarySearch(int low, int high, int key) {
  while (high - low + 1 > 1) {
     int mid = low + (high-low) / 2;
     (arr[mid] >= key ? high : low) = mid;
  }
  return high;
}

int
lis(int n) {
  int pointer = 1; //Siempre apunta un lugar vacío
  memset(dp, 0, sizeof(dp[0]) * n);
  dp[0] = arr[0];
  pointer = 1;
  for (int i = 1; i < n; i++) {
    if (arr[i] < dp[0])
      dp[0] = arr[i]; //Nuevo valor más pequeño
    else if (arr[i] > dp[pointer - 1])
      //Quiere extender la secuencia
      dp[pointer++] = arr[i];
    else {
      //Quiere ser el actual candidate de una secuencia
      //existente, reemplazará un valor piso de la tabla dp
      int index = binarySearch(0, pointer - 1, arr[i]);
      dp[index] = arr[i];
    }
  }
  return pointer;
}
```

.....

5.1.3. Orden logarítmico imprimiendo la secuencia

Retorna un **vector con los elementos** que componen la subsecuencia creciente más larga del arreglo arr.
Complejidad: $O(n \log_2 n)$ donde n es la longitud de arr.

```
const int MAXN = 100005;
int arr[MAXN]; //Almacena los números
int dp[MAXN]; //Almacena índices de los números
int prev[MAXN]; //Almacena índices de los antecesores

//OJO: Este binarySearch cambia con respecto al anterior
int
binarySearch(int low, int high, int key) {
  while (high - low + 1 > 1) {
     int mid = low + (high-low) / 2;
     if (arr[dp[mid]] > key) high = mid;
     else low = mid + 1;
  }
  return high;
}

vector <int>
lis(int n) {
  memset(dp, 0, sizeof(dp[0]) * n);
  //memset(prev, 0xFF, sizeof(prev[0]) * n);
  dp[0] = 0;
  prev[0] = -1;
  int pointer = 1; //Siempre apunta a una posición vacía
  for (int i = 1; i < n; i++) {
    if (arr[i] < arr[dp[0]]) {
      //Nuevo valor más pequeño
      dp[0] = i; //Almaceno el index
      prev[i] = -1; //El 0xFF del memset ya tiene un -1
    }
    else if (arr[i] > arr[dp[pointer - 1]]) {
      //Quiere extender la secuencia
      prev[i] = dp[pointer - 1];
      dp[pointer++] = i;
    }
    else {
      //Quiere ser un candidato potencial de una secuencia futura
      //Va a reemplazar un valor piso en la tabla dp
      int index = binarySearch(0, pointer - 1, arr[i]);
      prev[i] = dp[index - 1];
      dp[index] = i;
    }
  }

  vector <int> sec;
  for (int i = dp[pointer - 1]; i >= 0; i = prev[i]) {
    sec.push_back(arr[i]);
  }
```

```
  reverse(sec.begin(), sec.end());
  return sec;
}
```

5.2. Problema de la mochila

Halla el valor máximo que se puede obtener al empacar un subconjunto de n objetos en una mochila de tamaño W cuando se conoce el valor y el tamaño de cada objeto.
Complejidad: $O(n \times W)$ donde n es el número de objetos y W es la capacidad de la mochila.

```
// Máximo número de objetos
const int MAXN = 2005;
// Máximo tamaño de la mochila
const int MAXW = 2005;
// w[i] = peso del objeto i (i comienza en 1)
int w[MAXN];
// v[i] = valor del objeto i (i comienza en 1)
int v[MAXN];
// dp[i][j] máxima ganancia si se toman un subconjunto de los
// objetos 1 .. i y se tiene una capacidad de j
int dp[MAXN][MAXW];

int knapsack(int n, int W){
    for (int j = 0; j <= W; ++j) dp[0][j] = 0;

    for (int i = 1; i <= n; ++i){
        for (int j = 0; j <= W; ++j){
            dp[i][j] = dp[i-1][j];
            if (j - w[i] >= 0){
                dp[i][j] = max(dp[i][j], dp[i-1][j-w[i]] + v[i]);
            }
        }
    }
    return dp[n][W];
}
```

5.3. Edit distance

Calcula cual es el mínimo costo de convertir de un string s al string t, utilizando las operaciones de *reemplazar*, *insertar* y *borrar*.
Complejidad: $O(n \times m)$ donde n y m son los tamaños de los string.

```
const int MAXN = 5005;
int dp[MAXN][MAXN];
// dp[i][j] = Min cost of turning s[0..i) into t[0..j)
// Allowed operations are deletion, insertion and
// substitution (in the string s) .Note that the same
// result can be achieved deleting from s or inserting in t
// and viceversa

int
solve(string s, string t, int n, int m) {
  // Turn the empty string into t[0..j), add to s all
  // of the characters in t
  for (int j = 0; j <= m; j++) dp[0][j] = j;

  // Turn s[0..i) into the empty string, delete all
  // of the characters in s
  for (int i = 0; i <= n; i++) dp[i][0] = i;


  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      // Turn s[0..i) into t[0..j)

      // If the characters match, keep going
      if (s[i-1] == t[j-1]) dp[i][j] = dp[i-1][j-1];
      else {
        int substituteCost, insertCost, deleteCost;
        substituteCost = insertCost = deleteCost = 1;
        // Try substituting s[i-1] for t[j-1] and turn
        // s[0..i-1) into t[0..j-1)
        dp[i][j] = dp[i - 1][j - 1] + substituteCost;
        // Try deleting character s[i-1] and turn
        // s[0..i-1) into t[0..j)
        dp[i][j] = min(dp[i][j], dp[i - 1][j] + deleteCost);
        // Try inserting character t[j-1] and turn
        // s[0..i) into t[0..j-1)
        dp[i][j] = min(dp[i][j], dp[i][j - 1] + insertCost);
      }
```

```
    }
  }
  return dp[n][m];
}
```

5.4. Formas de sumar un número

Calcula la cantidad de formas en la que puedo obtener un número n con k sumandos.
Complejidad $O(MAXN^2)$ asumiendo, que la n y la k tienen el mismo límite máximo.
Notar que al inicio del main estamos construyendo la matriz, si no es necesario construirla toda, sino hasta la n y k que me den, entonces lo hago.

```
const int MAXN = 105;
//El número puede ser demasiado grande
const int MOD = 1000000;
// n: número que quiero hallar
// k: con cuántos sumandos
int n, k;
// Filas: k, Columnas: n
int dp[MAXK][MAXN];

void
build() {
  //Para cualquier n, si tengo 1 sumando, sólo tengo una opción
  //Para cualquier j, si tengo que sumar 0, sólo tengo una opción
  for (int j = 1; j < MAXN; j++) dp[1][j] = 1, dp[j][0] = 1;

  for (int i = 2; i < MAXN; i++) {
    for (int j = 1; j < MAXN; j++) {
      dp[i][j] = (dp[i-1][j] + dp[i][j-1]) % MOD;
    }
  }
}

int
main() {
  build();
  while (cin >> n >> k && n && k) {
    //Al final, la respuesta está en dp[k][n]
    cout << dp[k][n] << endl;
  }
  return 0;
}
```

5.5. Longest Common Substring

Permite hallar la subcadena común más larga, estrictamente continua, entre dos strings s y t.
El ejemplo actual también tiene el método *backtrack* que permite reconstruir la solución. Complejidad $O(n \times m)$ donde n y m son los tamaños del string.

```
const int MAXN = 1005;
int dp[MAXN][MAXN];

string
backtrack(const string &s, int i, int j, string p) {
  if (dp[i][j] == 0) {
    reverse(p.begin(), p.end());
    return p;
  }
  else {
    p += s[i-1];
    return backtrack(s, i-1, j-1, p);
  }
}

string
lcsubstr(const string &s, const string &t, int n, int m) {
  for (int i = 0; i <= n; i++) dp[i][0] = 0;
  for (int j = 0; j <= m; j++) dp[0][j] = 0;

  string ret = "";
  int maxi = 0;
  int maxI = -1, maxJ = -1;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      if (s[i-1] == t[j-1]) {
        dp[i][j] = dp[i-1][j-1] + 1;
        if (dp[i-1][j-1] + 1 > maxi) {
          maxI = i;
          maxJ = j;
```

```
          maxi = dp[i-1][j-1] + 1;
        }
      }
      else dp[i][j] = 0;
    }
  }
  if (maxI != -1 && maxJ != -1) {
    //Reconstruimos el longest common substring
    ret = backtrack(s, maxI, maxJ, "");
  }
  return ret;
}

int
main() {
  string s, t;
  cin >> s >> t;
  string lcsstring = lcsubstr(s, t, s.size(), t.size());
  if (lcsstring == "") printf("No common sequence.\n");
  else cout << lcsstring << endl;
  return 0;
}
```

5.6. Longest Common Subsequence

Halla la longitud de la máxima subsecuencia (no substring) de dos cadenas s y t.
Una subsecuencia de una secuencia s es una secuencia que se puede obtener de s al borrarle algunos de sus elementos (probablemente todos) sin cambiar el orden de los elementos restantes.
El algoritmo también se puede aplicar para vectores de elementos, no sólo para strings.
Complejidad: $O(n \times m)$ donde n es la longitud de s y m es la longitud de t.

```
const int MAXN = 1005;
int dp[MAXN][MAXN];

int
lcs(const string &s, const string &t) {
  int n = s.size(), m = t.size();
  for (int j = 0; j <= m; j++) dp[0][j] = 0;
  for (int i = 0; i <= n; i++) dp[i][0] = 0;

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      if (s[i-1] == t[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }
  }
  return dp[n][m];
}
```

5.7. Shortest Common Supersequence

El *Shortest Common Supersequence* es el string más corto z tal que los strings s y t sean subsecuencias de z.
Para hallar esto, simplemente es ejecutar el *Longest Common Subsequence* entre los strings s y t, y el tamaño de la supersecuencia más corta será $(n + m)$ - $lcs(s, t)$, donde n y m son los tamaños de los strings s y t respectivamente.

5.8. Maximum subarray (non-adjacent) sum (1D)

Dado un arreglo con números positivos, encontrar la máxima suma de una subsecuencia, teniendo en cuenta que los números de dicha secuencia no pueden ser adyacentes.
Por ejemplo `3 2 7 10` debería retornar 13 (suma de 3 y 10), `3 2 5 10 7` debería retornar 15 (suma de 3, 5 y 7).
Complejidad: $O(n)$ donde n es el tamaño del arreglo.

```
vector <int> arr;

int
findMaxSum(int n) {
  int incl = arr[0];
  int excl = 0;
  int excl_new;
  for (int i = 1; i < n; i++) {
    // Current max excluding i.
    excl_new = (incl > excl) ? incl : excl;
    // Current max including i.
    incl = excl + arr[i];
    excl = excl_new;
```

```
  }
  // Return max of incl and excl
  return ((incl > excl) ? incl : excl);
}
```

5.9. Maximum subarray sum (1D) - Kadane's Algorithm

Encuentra el subarreglo continuo (que contiene al menos un entero positivo) con la suma más grande.
Complejidad: $O(n)$ donde n es el tamaño del arreglo.

```
int
kadane(vector <int> a, int n) {
  int max_so_far = 0, max_ending_here = 0;
  for(int i = 0; i < n; i++) {
    max_ending_here = max_ending_here + a[i];
    if(max_ending_here < 0) max_ending_here = 0;
    if(max_so_far < max_ending_here) max_so_far = max_ending_here;
  }
  return max_so_far;
}
```

5.9.1. Maximum circular subarray sum

Si el arreglo es circular y queremos hallar la suma más grande se hace el siguiente procedimiento:

- Aplicamos $Kadane's$ al arreglo sin modificar, y se almacena como máximo: `maxi`.
- Se itera sobre el arreglo para calcular la suma acumulada `acc` e invertir los valores del arreglo: `a_inv[i] = -a[i]`.
- La respuesta es el máximo entre el valor del punto 1 y la suma acumulada más el resultado de un nuevo llamado al algoritmo con el arreglo invertido: `max(maxi, acc + kadane(a_inv))`

5.10. Maximum subrectangle sum (2D)

Permite hallar el sub-rectángulo de una matriz con la mayor suma, la suma de un rectángulo es la suma de todos sus elementos. Un sub-rectángulo es cualquier sub-arreglo continuo de tamaño 1×1 o mayor, localizado dentro del toda la matriz.

5.10.1. Naïve solution - $O(n^4)$

Complejidad: $O(n^4)$ donde n es el tamaño de un lado de la matriz (asumiendo que son iguales).
Cuidado que puede no pasar con 100. Lo idea sería `MAXN` $= \sqrt[4]{10^6}$

```
const int MAXN = 105;
const int INF = 1 << 30;
int n;
int arr[MAXN][MAXN];

int
main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> arr[i][j];
            if (i > 0) arr[i][j] += arr[i-1][j];
            if (j > 0) arr[i][j] += arr[i][j-1];
            if (i > 0 && j > 0) arr[i][j] -= arr[i-1][j-1];
        }
    }
    int maxi = -1*INF;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = i; k < n; k++) {
                for (int l = j; l < n; l++) {
                    int sub = arr[k][l];
                    if (i > 0) sub -= arr[i-1][l];
                    if (j > 0) sub -= arr[k][j-1];
                    if (i > 0 && j > 0) sub += arr[i-1][j-1];
                    maxi = max(maxi, sub);
                }
            }
        }
    }
    cout << maxi << endl;
    return 0;
}
```

5.10.2. Using Kadane's - $O(n^3)$

```
const int ROW = 105;
```

```
const int COL = 105;
int M[ROW][COL];
int arr[ROW];
int start, finish;
int n;

int
kadane() {
  int sum = 0, maxSum = INT_MIN, i;
  finish = -1;
  int local_start = 0;
  for (i = 0; i < n; ++i) {
    sum += arr[i];
    if (sum < 0) {
      sum = 0;
      local_start = i + 1;
    }
    else if (sum > maxSum) {
      maxSum = sum;
      start = local_start;
      finish = i;
    }
  }
  if (finish != -1) return maxSum;
  // Special Case: When all numbers in arr are negative
  maxSum = arr[0];
  start = finish = 0;
  // Find the maximum element in array
  for (i = 1; i < n; i++) {
    if (arr[i] > maxSum) {
      maxSum = arr[i];
      start = finish = i;
    }
  }
  return maxSum;
}

void
findMaxSum() {
  // Variables to store the final output
  int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;
  int left, right, i;
  int sum;

  for (left = 0; left < COL; ++left) {
    memset(arr, 0, sizeof(arr));
    for (right = left; right < COL; ++right) {
      for (i = 0; i < ROW; ++i) arr[i] += M[i][right];
      sum = kadane();
      if (sum > maxSum) {
        maxSum = sum;
        finalLeft = left;
        finalRight = right;
        finalTop = start;
        finalBottom = finish;
      }
    }
  }
  printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);
  printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);
  printf("Max sum is: %d\n", maxSum);
}
```

5.11. Maximum subrectangle sum (3D)

Permite hallar la mayor suma en un arreglo 3D. Complejidad $O(n^6)$ donde n es el tamaño de uno de los lados del cubo.

5.12. Partition problem

Determina si un arreglo de números puede ser particionado en dos conjuntos tal que la suma de ambos sea la misma.
Complejidad: $O(sum * n)$, donde n es el tamaño del arreglo y sum es la suma de cada conjunto. (notar que este algoritmo no es eficiente para arreglos con grandes sumas, para ese caso intentar una solución recursiva $O(2^n)$)

```
vector <int> arr;

/* part[i][j] = true if a subset of {arr[0], arr[1], ..arr[j-1]}
                has sum equal to i, otherwise false */
bool findPartiion (int n) {
  int sum = 0;
  int i, j;
  for (i = 0; i < n; i++) sum += arr[i];
  if (sum % 2 != 0) return false;
```

```
  bool part[sum / 2 + 1][n + 1];
  // Initialize top row as true
  for (i = 0; i <= n; i++) part[0][i] = true;
  // Initialize leftmost column, except part[0][0], as 0
  for (i = 1; i <= sum / 2; i++) part[i][0] = false;
    // Fill the partition table in botton up manner
    for (i = 1; i <= sum/2; i++) {
      for (j = 1; j <= n; j++) {
        part[i][j] = part[i][j-1];
        if (i >= arr[j-1])
          part[i][j] = part[i][j] || part[i - arr[j-1]][j-1];
      }
    }
  /* Uncomment this part to print table.
   for (i = 0; i <= sum/2; i++)
   {
     for (j = 0; j <= n; j++)
        printf ("%4d", part[i][j]);
     printf("\n");
   } */
  return part[sum/2][n];
}
```

.....

5.13. Longest Palindromic Substring

Dado un string, encontrar el substring (estrictamente continuo) más largo que sea palíndromo. Por ejemplo si tenemos `"forgeeksskeegfor"` la salida debería ser: `"geeksskeeg"`
Complejidad: Creo que el primer método $O(n^3)$, el segundo $O(n)$ y el tercero $O(n^2)$.

```
/**
 * Date 07/29/2015
 * @author Tushar Roy
 * Given a string find longest palindromic substring in this string.
 */
public class LongestPalindromeSubstring {
  // I think this takes O(n^3)
  public int longestPalindromeSubstringEasy(char arr[]) {
    int longest_substring = 1;
    for (int i = 0; i < arr.length; i++) {
      int x, y;
      int palindrome;
      x = i;
      y = i + 1;
      palindrome = 0;
      while (x >= 0 && y < arr.length && arr[x] == arr[y]) {
        x--;
        y++;
        palindrome += 2;
      }
      longest_substring = Math.max(longest_substring, palindrome);
      x = i - 1;
      y = i + 1;
      palindrome = 1;
      while (x >= 0 && y < arr.length && arr[x] == arr[y]) {
        x--;
        y++;
        palindrome += 2;
      }
      longest_substring = Math.max(longest_substring, palindrome);
    }
    return longest_substring;
  }

  /**
   * Linear time Manacher's algorithm to find longest palindromic
     substring.
   * There are 4 cases to handle
   * Case 1 : Right side palindrome is totally contained under
     current palindrome. In this case do not consider this as
     center.
   * Case 2 : Current palindrome is proper suffix of input.
               Terminate the loop in this case.
               No better palindrom will be found on right.
   * Case 3 : Right side palindrome is proper suffix and its
               corresponding left side
               palindrome is proper prefix of current palindrome.
               Make largest such point as
   * next center.
   * Case 4 : Right side palindrome is proper suffix but its
               left corresponding palindrome is be beyond current
               palindrome. Do not consider this as center because
               it will not extend at all.
   *
```

```
 * To handle even size palindromes replace input string with
   one containing $ between every
   input character and in start and end.
 */
public int longestPalindromicSubstringLinear(char input[]) {
  int index = 0;
  // preprocess the input to convert it into type abc -> $a$b$c$
  // to handle even length case.
  // Total size will be 2*n + 1 of this new array.
  char newInput[] = new char[2*input.length + 1];
  for(int i=0; i < newInput.length; i++) {
    if(i % 2 != 0) {
      newInput[i] = input[index++];
    } else {
      newInput[i] = '$';
    }
  }
  // create temporary array for holding largest palindrome at
  // every point.
  // There are 2*n + 1 such points.
  int T[] = new int[newInput.length];
  int start = 0;
  int end = 0;
  //here i is the center.
  for(int i=0; i < newInput.length; ) {
    //expand around i. See how far we can go.
    while(start >0 && end < newInput.length-1 &&
          newInput[start-1] == newInput[end+1]) {
      start--;
      end++;
    }
    //set the longest value of palindrome around center i at T[i]
    T[i] = end - start + 1;

    // this is case 2. Current palindrome is proper suffix of input.
    // No need to proceed. Just break out of loop.
    if(end == T.length -1) {
      break;
    }
    // Mark newCenter to be either end or end + 1 depending on if we
    // dealing with even or old number input.
    int newCenter = end + (i%2 ==0 ? 1 : 0);
    for(int j = i + 1; j <= end; j++) {
      // i - (j - i) is left mirror. Its possible left mirror might
      // go beyond current center palindrome. So take minimum of
      // either left side palindrome or distance of j to end.
      T[j] = Math.min(T[i - (j - i)], 2 * (end - j) + 1);
      // Only proceed if we get case 3. This check is to make sure
      // we do not pick j as new center for case 1 or case 4.
      // As soon as we find a center lets break out of this inner
      // while loop.
      if(j + T[i - (j - i)]/2 == end) {
        newCenter = j;
        break;
      }
    }
    // make i as newCenter. Set right and left to atleast the value
    // we already know should be matching based of left side
    // palindrome.
    i = newCenter;
    end = i + T[i]/2;
    start = i - T[i]/2;
  }

  //find the max palindrome in T and return it.
  int max = Integer.MIN_VALUE;
  for(int i = 0; i < T.length; i++) {
      int val;
      /*if(i%2 == 0) {
        val = (T[i] -1)/2;
      } else {
        val = T[i]/2;
      }*/
      val = T[i]/2;
      if(max < val) {
        max = val;
      }
  }
  return max;
}

// I think this takes O(n^2)
public int longestPalindromeDynamic(char []str){
  boolean T[][] = new boolean[str.length][str.length];
  for(int i=0; i < T.length; i++){
```

```
      T[i][i] = true;
  }
  int max = 1;
  for(int l = 2; l <= str.length; l++){
    int len = 0;
    for(int i=0; i < str.length-l+1; i++){
      int j = i + l-1;
      len = 0;
      if(l == 2){
        if(str[i] == str[j]){
          T[i][j] = true;
          len = 2;
        }
      }
      else{
        if(str[i] == str[j] && T[i+1][j-1]){
          T[i][j] = true;
          len = j -i + 1;
        }
      }
      if(len > max) {
        max = len;
      }
    }
  }
  return max;
}

public static void main(String args[]) {
  LongestPalindromeSubstring lps = new LongestPalindromeSubstring();
  System.out.println(lps
    .longestPalindromicSubstringLinear("abba"
                                       .toCharArray()));
  System.out.println(lps
    .longestPalindromicSubstringLinear("abbababba"
                                       .toCharArray()));
  System.out.println(lps
    .longestPalindromicSubstringLinear("babcbaabcbaccba"
                                        .toCharArray()));
  System.out.println(lps
    .longestPalindromicSubstringLinear("cdbabcbabdab"
                                        .toCharArray()));
}
```

```
}
```

.....

5.14. Longest Palindromic Subsequence

Dado un string, encontrar el tamaño de la subsecuencia (no necesariamente continua). Por ejemplo para `BBABCBCAB` la respuesta es 7, porque `BABCBAB` es la subsecuencia palindrómica más larga.

```
int lps(const string &str) {
  int n = str.size();
  int i, j, cl;
  int L[n][n];
  for (i = 0; i < n; i++) L[i][i] = 1;
  for (cl = 2; cl <= n; cl++) {
    for (i = 0; i < n - cl + 1; i++) {
      j = i + cl - 1;
      if (str[i] == str[j] && cl == 2) L[i][j] = 2;
      else if (str[i] == str[j]) L[i][j] = L[i+1][j-1] + 2;
      else L[i][j] = max(L[i][j-1], L[i+1][j]);
    }
  }
  return L[0][n-1];
}
```

.....

6. Strings

6.1. Algoritmo de KMP

Encuentra si el string `needle` aparece en en string `haystack`.
Si no se retorna directamente `true` cuando se halla la primera ocurrencia, el algoritmo encuentra todas las ocurrencias de `needle` en `haystack`.
La primera parte del algoritmo llena el arreglo `border` donde `border[i]` es la longitud del borde del prefijo de `needle` que termina en la posición `i`. Un borde de una cadena s es la cadena más larga que es a la vez prefijo y sufijo de s pero que es diferente de s.
Complejidad: $O(n)$ donde n es el tamaño de `haystack`.

```
bool kmp(const string &needle, const string &haystack){
```

```
   int m = needle.size();
   vector<int> border(m);
   border[0] = 0;

   for (int i = 1; i < m; ++i) {
      border[i] = border[i - 1];
      while (border[i] > 0 and needle[i] != needle[border[i]])
         border[i] = border[border[i] - 1];
      if (needle[i] == needle[border[i]]) border[i]++;
   }

   int n = haystack.size();
   int seen = 0;
   for (int i = 0; i < n; ++i){
      while (seen > 0 and haystack[i] != needle[seen])
         seen = border[seen - 1];
      if (haystack[i] == needle[seen]) seen++;
      if (seen == m) return true; // Ocurre entre [i - m + 1, i]
   }
   return false;
}
```

.....

6.2. Algoritmo de Booth - Lexicographically minimal string rotation

Encuentra la rotación de un string que contiene el menor orden lexicográfico de todas las rotaciones posibles. Por ejemplo, la menor rotación lexicográfica de `bbaaccaadd` sería `aaccaaddbb`, y de `abcdeabcdea` sería `aabcdeabcde`.
El algoritmo concatena el string a sí mismo y computa una tabla basándose en el algoritmo de KMP.
El método devuelve el **índice en base 0 a primera letra correspondiente a la mejor rotación**, si se necesita el string completo se deberá utilizar substring para reconstruirlo: `str.substr(ind, str.size()) + str.substr(0, ind)`
Complejidad: $O(n)$ donde n es el tamaño del string.

```
int
booth(const string &str) {
  string s = str + str;
  int n = s.size();
  vector <int> f(n, -1);
  int k = 0;
  for (int j = 1; j < n; ++j) {
    int i = f[j - k - 1];
    while (i != -1 && s[j] != s[k + i + 1]) {
      if (s[j] < s[k + i + 1]) k = j - i - 1;
      i = f[i];
    }
    if (i == -1 && s[j] != s[k + i + 1]) {
      if (s[j] < s[k + i + 1]) k = j;
      f[j - k] = -1;
    }
    else f[j - k] = i + 1;
  }
  return k;
}
```

.....

6.3. Formatos de impresión

6.3.1. Números

Con la función *printf* podemos dar distintos formatos de impresión, por ejemplo justificar, ceros o espacios iniciales, etc.

```
int
main() {
  //Width 3
  printf("%3d\n", 45);
  //Width 3 leading 0's
  printf("%03d\n", 45);
  //Width 6, after point precision 2
  printf("%6.2f\n", 4.123);
  //Width 6 leading 0's, after point precision 2
  printf("%06.2f\n", 4.123);
  printf("%x\n", 255); //Prints in hexa lowercase
  printf("%X\n", 255); //Prints in hexa uppercase
  printf("%o\n", 16); //Prints in octal
  return 0;
}
```

.....

6.3.2. Strings

- The printf("%s", "Hello, world!"); statement prints the string (nothing special happens.)

- The printf("%15s", "Hello, world!"); statement prints the string, but print 15 characters. If the string is smaller the "empty" positions will be filled with "whitespace."
- The printf("%.10s", "Hello, world!"); statement prints the string, but print only 10 characters of the string.
- The printf("%-10s", "Hello, world!"); statement prints the string, but prints at least 10 characters. If the string is smaller "whitespace" is added at the end. (See next example.)
- The printf("%-15s", "Hello, world!"); statement prints the string, but prints at least 15 characters. The string in this case is shorter than the defined 15 character, thus "whitespace" is added at the end (defined by the minus sign.)
- The printf("%.15s", "Hello, world!"); statement prints the string, but print only 15 characters of the string. In this case the string is shorter than 15, thus the whole string is printed.
- The printf("%15.10s", "Hello, world!"); statement prints the string, but print 15 characters. If the string is smaller the "empty" positions will be filled with "whitespace." But it will only print a maximum of 10 characters, thus only part of new string (old string plus the whitespace positions) is printed.
- The printf("%-15.10s", "Hello, world!"); statement prints the string, but it does the exact same thing as the previous statement, accept the "whitespace" is added at the end.

6.3.3. Formato en Java

Para utilizar formatos de impresión en Java se debe usar *System.out.format* y darle formatos muy parecidos a los de C++, acá podemos ver varios ejemplos. Se pueden dar incluso formatos para impresión de *Calendar*.

```
import java.util.Calendar;
import java.util.Locale;

public class Main {

    public static void main(String[] args) {
      long n = 461012;
      System.out.format("%d%n", n);      //  -->  "461012"
      System.out.format("%08d%n", n);    //  -->  "00461012"
      System.out.format("%+8d%n", n);    //  -->  " +461012"
      System.out.format("%,8d%n", n);    // -->  " 461,012"
      System.out.format("%+,8d%n%n", n); //  -->  "+461,012"

      double pi = Math.PI;

      System.out.format("%f%n", pi);       // -->  "3.141593"
      System.out.format("%.3f%n", pi);     // -->  "3.142"
      System.out.format("%10.3f%n", pi);   // -->  "     3.142"
      System.out.format("%-10.3f%n", pi);  // -->  "3.142"
      System.out.format(Locale.ENGLISH,
                        "%-10.4f%n%n", pi); // -->  "3.1416"

      Calendar c = Calendar.getInstance();
      //  -->  "September 3, 2014"
      System.out.format("%tB %te, %tY%n", c, c, c);
      //  -->  "3:21 am"
      System.out.format("%tl:%tM %tp%n", c, c, c);
      //  -->  "09/03/14"
      System.out.format("%tD%n", c);
    }
}
```

.....

7. Otros

7.1. Binary Search

Permite buscar un elemento (*key*) en un arreglo **ordenado** (*arr*). Complejidad: $O(n \log_2 n)$ donde n es la cantidad de elementos del arreglo.

7.1.1. Traditional algorithm

Returns the index of some element that is equal to the given value (if there are multiple elements, it returns some arbitrary one).

```
// invariants: value > arr[i] for all i < low
//             value < arr[i] for all i > high
//             or <= and => if mutiple elements of
//                           the same value are allowed.
int binary_search(vector <int> arr, int value) {
  int n = arr.size();
  int low = 0;
```

```
  int high = n - 1;
  while (low <= high) {
     int mid = (low + high) / 2;
     if (arr[mid] > value) high = mid - 1;
     else if (arr[mid] < value) low = mid + 1;
     else return mid;
  }
  return low; // I didn't find the value.
              // so it would be inserted at low index.
}
```

.....

7.1.2. Lower bound

Returns the leftmost place where the given element can be correctly inserted.

```
// invariants: value > arr[i] for all i < low
//             value <= arr[i] for all i > high
int lower_bound(vector <int> arr, int value) {
  int n = arr.size();
  int low = 0;
  int high = n - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (arr[mid] >= value) high = mid - 1;
    else low = mid + 1;
  }
  return low;
}
```

.....

7.1.3. Upper bound

Returns the rightmost place where the given element can be correctly inserted.

```
// invariants: value >= arr[i] for all i < low
//             value < arr[i] for all i > high
int upper_bound(vector <int> arr, int value) {
  int n = arr.size();
  int low = 0;
  int high = n - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    // Comparing with > instead of >=
    if (arr[mid] > value) high = mid - 1;
    else low = mid + 1;
  }
  return low;
}
```

.....

7.2. BitSet

Almacena bits, simula un array de booleanos, optimizando el espacio, cada elemento puede ser accedido por el operador [ ]

```
int
main() {
  bitset<16> set1; //Inicializa todo en 0
  bitset<16> set2(0xFA2); //Inicializa con valor hex
  bitset<16> set3(string("0101101")); //Inicializa a partir
                                      //del string
  cout << set3.count() << endl; //Imrpime 4
  cout << set3.test(0) << endl; //Imprime true
  cout << set3.test(1) << endl; //Imprime false
  cout << set1.any() << endl; //False porque set1 no tiene
                              //bits en 1
  cout << set1.none() << endl; //True
  //cout << set2.all() << endl; //False, almenos hay
                                //un bit apagado
  cout << set1.set() << endl; //Poner todos en 1
  cout << set1.set(2) << endl; //Poner pos 2 en 1
  cout << set1.reset(1) << endl; //Apaga el pos 1
  cout << set1.flip(2) << endl; //Hacer flip al pos 2
  string s = set1.to_string <char, string::traits_type,
                             string::allocator_type>();
  cout << s << endl;
  cout << set1.to_ulong() << endl;
  return 0;
}
```

.....

7.3. Máximo orden dado un n

En la siguiente tabla se encuentran los límites de orden máximo respecto a un número de elementos n del problema.

n	Worst AC Algorithm	Comment
≤ 10	$O(n!), O(n^6)$	e.g. Enumerating a Permutation
≤ 20	$O(2^n), O(n^5)$	e.g. DP + Bitmask Technique
≤ 50	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, choosing ${}_nC_{k=4}$
≤ 100	$O(n^3)$	e.g. Floyd Warshall's
$\leq 1K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort
$\leq 100K$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree
$\leq 1M$	$O(n), O(\log_2 n), O(1)$	Usually, contest problem has $n \leq 1M$ (e.g. to read input)

7.4. Suma de grandes números en C++

Como sabemos, en C++ los enteros (*int*) tienen un rango de $(2^31) - 1$, y los *longlong* tienen un rango de $(2^64) - 1$, por esto no podemos almacenar números de más de 18 19 números, para esto se debe usar la clase *BigInteger* en *Java*, o si no hay que preocuparse por tiempo, podemos representar los números como *string* y utilizar la siguiente funciones como la siguiente:

```
string
sumar(const string &a, const string &b) {
  int ia = a.size() - 1, ib = b.size() - 1;
  int llevo = 0;
  string ans = "";
  while (ia >= 0 || ib >= 0) {
    int sum = (ia >= 0 ? toInt(a[ia--]) : 0)
            + (ib >= 0 ? toInt(b[ib--]) : 0);
    sum += llevo;
    llevo = sum / 10;
    ans += toStr(sum % 10);
  }
  if (llevo) ans += toStr(llevo);
  reverse(ans.begin(), ans.end());
  return ans;
}
```

.....

7.5. Largest Rectangle in a Histogram

Encuentra el área rectangular más grande posible en un histograma, donde el rectángulo es determinado por un número de barras continuas. Se asume que todas las barras tienen el mismo ancho de 1 unidad.
Complejidad: $O(n)$ donde n es la cantidad de barras en el histograma.

```
const int MAXN = 100005;
typedef long long ll;
int n;
ll h[MAXN];
stack <int> s;

/* DO NOT call this. Call largest_rectangle */
ll
calc(int i) {
  int smallest = s.top(); s.pop();
  ll cur = h[smallest] * (s.empty() ? i : i - s.top() - 1);
  return cur;
}

/* h[i] = height of the bar i.
   n = total number of bars in the histogram.
   s = an empty stack. */
ll
largest_rectangle() {
  ll ans = 0LL;
  int i = 0;
  while (i < n) {
    if (s.empty() || h[s.top()] <= h[i]) s.push(i++);
    else ans = max(ans, calc(i));
  }
  while (!s.empty()) ans = max(ans, calc(i));
  return ans;
}
```

.....

8. Struct

Las estructuras en C++ son muy útiles a la hora de hacer algún tipo de dato muy específico o usar funciones de comparación personalizadas

8.1. Función de comparación

En este caso tenemos una función de comparación que involucra los dos atributos x y y de la estructura *dato*

```
struct dato {
  //x tiene más prioridad que y, (Ver abajo la funcion <)
  int x, y;
  dato (int X, int Y) : x(X), y(Y) {}

  //Función que reemplaza el operador < entre dos 'dato'
  //retorna true si this es menor, false de lo contrario.
  bool operator < (const dato &otro) const {
    if (x < otro.x) return true;
    else if (x == otro.x) {
      //Como las x son iguales, entro a preguntar por las y
      if (y < otro.y) return true;
      else return false;
    }
    else { //x > otro.x
       return false;
    }
  }
};

int
main() {
  int x, y;
  vector <dato> v;
  while (cin >> x >> y && x) {
    v.push_back(dato(x, y));
  }
  sort(v.begin(), v.end());
  cout << "Ordenados: " << endl;
  for (int i = 0; i < v.size(); i++) {
    cout << v[i].x << " " << v[i].y << endl;
  }
  return 0;
}
```

.....

8.2. Radix sort

Complejidad: $O(w \times n)$ donde n es la cantidad de elementos del arreglo y w la longitud del mayor número.

```
int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
      if (arr[i] > max) max = arr[i];
    return max;
}
/*
 * count sort of arr[]
 */
void countSort(int arr[], int n, int exp) {
    int output[n];
    int i, count[10] = {0};
    for (i = 0; i < n; i++) count[(arr[i] / exp) % 10]++;
    for (i = 1; i < 10; i++) count[i] += count[i - 1];
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (i = 0; i < n; i++) arr[i] = output[i];
}
/*
 * sorts arr[] of size n using Radix Sort
 */
void radixsort(int arr[], int n) {
    int m = getMax(arr, n);
    for (int exp = 1; m / exp > 0; exp *= 10)
      countSort(arr, n, exp);
}
```

.....

9. C++

9.1. Strings con arreglo de caracteres

Para lograr una implementación eficiente, se pueden trabajar los strings como arreglo de caracteres, es decir, como en C, existen varias funciones que pueden ser útiles, NOTA: no todas las funciones se encuentran en el ejemplo.

```
#include <iostream>
#include <stdio.h>
#include <string>

using namespace std;
```

```
const int MAXN = 1005;
char s[MAXN];
string cs;

int
main() {
  while (true) {
    scanf("%s", s); //Only a token stored in char array
    cout << "scanf: |" << s << "|" << endl;
    gets(s); //Until end of line or EOF stored in char array
    cout << "gets: |" << s << "|" << endl;
    getline(cin, cs); //Until end of line or EOF stored in string
    cout << "getline: |" << cs << "|" << endl;
    // ---------------------------------------
    for (int i = 0; i < strlen(s); i++) cout << s[i] << " - ";
    cout << endl;
    char dest[MAXN];
    strncpy(dest, s, 5); //Dest, source, length (size_t)
    dest[5] = '\0'; //ALWAYS ADD NULL CHARACTER AT END POSITION
    cout << "cpyDest: |" << dest << "|" << endl;
    strncat(dest, s, 3); //Dest, source, length (size_t)
    cout << "catDest: |" << dest << "|" << endl;
    dest[4] = '\0'; //Cut string, result length 5
    if (strncmp(s, dest, 5) == 0) cout << "equals 5" << endl;
    if (strncmp(s, dest, 4) == 0) cout << "equals 4" << endl;
  }
  return 0;
}
```