Resumen de algoritmos para torneos de programación

Andrés Mejía

7 de octubre de 2011

Índice

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 For(i, a, b) for (int i=(a); i<(b); ++i)
#define foreach(x, v) for (typeof (v).begin() x = (v).begin(); \
                           x != (v).end(); ++x)
#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;
}

#define INPUT_FILE "problemname"

int main(){
    freopen(INPUT_FILE ".in", "r", stdin); // Read from file

    return 0;
}
```

....

2. Teoría de números

2.1. Big mod

```
//retorna (b^p)mod(m)
// 0 <= b,p <= 2147483647
// 1 <= m <= 46340
int bigmod(int b, int p, int m){
  int mask = 1;
  int pow2 = b % m;
  int r = 1;
  while (mask){
    if (p & mask) r = (r * pow2) % m;
    pow2 = (pow2 * pow2) % m;
    mask <<= 1;
  }
  return r;
}
// Si se cambian los int por long longs los
// valores de entrada deben cumplir:
// 0 <= b,p <= 9223372036854775807
// 1 <= m <= 3037000499
// Si se cambian por unsigned long longs:
```

```
// 0 <= b,p <= 18446744073709551615
// 1 <= m <= 4294967295
```

2.2. Criba de Eratóstenes

Field-testing:

- *SPOJ* - 2912 - Super Primes
- *Live Archive* - 3639 - Prime Path

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

SIZE	Tiempo (s)
100000	0.003
1000000	0.060
10000000	0.620
100000000	7.650

```
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;
      }
    }
  }
}
```

2.3. Divisores de un número

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

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

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	$\frac{(n+r-1)!}{r!(n-1)!}$

Tomado de *Matemática discreta y sus aplicaciones*, Kenneth Rosen, 5$^{\text{ta}}$ edición, McGraw-Hill, página 315.

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. 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;
}
```

4.2. Minimum spanning tree: Algoritmo de Prim

```
//Complejidad: O(E log V)
//¡El grafo debe ser no digirido!
typedef string node;
typedef pair<double, node> edge;
//edge.first = peso de la arista, edge.second = nodo al que se
//dirige
typedef map<node, vector<edge> > graph;

double prim(const graph &g){
  double total = 0.0;
  priority_queue<edge, vector<edge>, greater<edge> > q;
  q.push(edge(0.0, g.begin()->first));
  set<node> visited;
  while (q.size()){
    node u = q.top().second;
    double w = q.top().first;
    q.pop(); //!!
    if (visited.count(u)) continue;
    visited.insert(u);
    total += w;
    vector<edge> &vecinos = g[u];
    for (int i=0; i<vecinos.size(); ++i){
      node v = vecinos[i].second;
      double w_extra = vecinos[i].first;
      if (visited.count(v) == 0){
        q.push(edge(w_extra, v));
      }
    }
  }
  return total; //suma de todas las aristas del MST
}
```

4.3. 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;
}
```

. . . .

4.4. 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.
}
```

. . . .

4.5. 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;
}
```

....

4.6. 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;
}
```

4.7. 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;
}
```

....

4.8. 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;
}
```

....

4.9. 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++;
  }
}
```

....

4.10. 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 $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$. 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, 4.9).

5. Programación dinámica

5.1. Longest common subsequence

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

int lcs(const string &s, const string &t){
  int m = s.size(), n = t.size();
  if (m == 0 || n == 0) return 0;
  for (int i=0; i<=m; ++i)
```

```
    dp[i][0] = 0;
  for (int j=1; j<=n; ++j)
    dp[0][j] = 0;
  for (int i=0; i<m; ++i)
    for (int j=0; j<n; ++j)
      if (s[i] == t[j])
dp[i+1][j+1] = dp[i][j]+1;
      else
dp[i+1][j+1] = MAX(dp[i+1][j], dp[i][j+1]);
  return dp[m][n];
}
```

....

5.2. 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. Strings

6.1. Algoritmo de Knuth-Morris-Pratt (KMP)

Computa el arreglo *border* que contiene la longitud del borde más largo de todos los prefijos de una cadena.
Un borde de una cadena es otra cadena más corta que es a la vez prefijo y sufijo de la original (por ejemplo, *aba* es un border de *abacaba* porque es más corta que *abacaba* y es al mismo tiempo prefijo y sufijo de *abacaba*. *ab* también es un borde de *abacaba*. *abac* no es un borde de *abacaba* porque no es un sufijo).

En el código, `border[i]` contiene el borde más grande del prefijo de longitud i de *needle* (*needle* es el patrón que se quiere buscar en la otra cadena). Una ejemplo del arreglo `border` es:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
needle	a	b	a	c	a	b	a	c	a	b	a	d	a	b	
border	-1	0	0	1	0	1	2	3	4	5	6	7	0	1	2

```
////////////////////////////////////////////////////////////////
//      Knuth-Morris-Pratt algorithm for string matching      //
//                    Complexity: O(n + m)                    //
////////////////////////////////////////////////////////////////

// Reports all occurrences of 'needle' in 'haystack'.
void kmp(const string &needle, const string &haystack) {
  // Precompute border function
  int m = needle.size();
  vector<int> border(m + 1);
  border[0] = -1;
  for (int i = 0; i < m; ++i) {
    border[i+1] = border[i];
    while (border[i+1] > -1 and
           needle[border[i+1]] != needle[i]) {
      border[i+1] = border[border[i+1]];
    }
```

```
      border[i+1]++;
    }

    // Now the actual matching
    int n = haystack.size();
    int seen = 0;
    for (int i = 0; i < n; ++i){
      while (seen > -1 and needle[seen] != haystack[i]) {
        seen = border[seen];
      }
      if (++seen == m) {
        printf("Needle occurs from %d to %d\n", i - m + 1, i);
        seen = border[m];
      }
    }
}
```

6.2. Algoritmo de Aho-Corasick

Sirve para buscar muchos patrones en una cadena. Por ejemplo, dada la cadena *ahishers* y los patrones $\{he, she, hers, his\}$, encuentra que *his* aparece en la posición 1, *he* aparece en la posición 4, *she* aparece en la posición 3 y *hers* aparece en la posición 4.

Complejidad: $O(n+m)$ donde n es la longitud de la cadena en la que hay que buscar y m es la suma de las longitudes de todos los patrones.

```
////////////////////////////////////////////////////////////////
//         Aho-Corasick's algorithm, as explained in          //
//          http://dx.doi.org/10.1145/360825.360855           //
////////////////////////////////////////////////////////////////

// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 6 * 50 + 10;

// Number of characters in the alphabet.
const int MAXC = 26;

// Output for each state, as a bitwise mask.
// Bit i in this mask is on if the keyword with index i
// appears when the machine enters this state.
int out[MAXS];

// Used internally in the algorithm.
int f[MAXS]; // Failure function
int g[MAXS][MAXC]; // Goto function, or -1 if fail.

// Builds the string matching machine.
//
// words - Vector of keywords. The index of each keyword is
//         important:
//         "out[state] & (1 << i)" is > 0 if we just found
//          word[i] in the text.
// lowestChar - The lowest char in the alphabet.
//              Defaults to 'a'.
// highestChar - The highest char in the alphabet.
//               Defaults to 'z'.
//               "highestChar - lowestChar" must be <= MAXC,
//               otherwise we will access the g matrix outside
//               its bounds and things will go wrong.
//
// Returns the number of states that the new machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(const vector<string> &words,
                         char lowestChar = 'a',
                         char highestChar = 'z') {
    memset(out, 0, sizeof out);
    memset(f, -1, sizeof f);
    memset(g, -1, sizeof g);

    int states = 1; // Initially, we just have the 0 state

    for (int i = 0; i < words.size(); ++i) {
        const string &keyword = words[i];
        int currentState = 0;
        for (int j = 0; j < keyword.size(); ++j) {
            int c = keyword[j] - lowestChar;
            if (g[currentState][c] == -1) {
                // Allocate a new node
```

```
            g[currentState][c] = states++;
        }
        currentState = g[currentState][c];
    }
    // There's a match of keywords[i] at node currentState.
    out[currentState] |= (1 << i);
}

// State 0 should have an outgoing edge for all characters.
for (int c = 0; c < MAXC; ++c) {
    if (g[0][c] == -1) {
        g[0][c] = 0;
    }
}

// Now, let's build the failure function
queue<int> q;
// Iterate over every possible input
for (int c = 0; c <= highestChar - lowestChar; ++c) {
    // All nodes s of depth 1 have f[s] = 0
    if (g[0][c] != -1 and g[0][c] != 0) {
        f[g[0][c]] = 0;
        q.push(g[0][c]);
    }
}
while (q.size()) {
    int state = q.front();
    q.pop();
    for (int c = 0; c <= highestChar - lowestChar; ++c) {
        if (g[state][c] != -1) {
            int failure = f[state];
            while (g[failure][c] == -1) {
                failure = f[failure];
            }
            failure = g[failure][c];
            f[g[state][c]] = failure;

            // Merge out values
            out[g[state][c]] |= out[failure];
            q.push(g[state][c]);
        }
            }
    }

    return states;
}

// Finds the next state the machine will transition to.
//
// currentState - The current state of the machine. Must be
//                between 0 and the number of states - 1,
//                inclusive.
// nextInput - The next character that enters into the machine.
//             Should be between lowestChar and highestChar,
//             inclusive.
// lowestChar - Should be the same lowestChar that was passed
//              to "buildMatchingMachine".

// Returns the next state the machine will transition to.
// This is an integer between 0 and the number of states - 1,
// inclusive.
int findNextState(int currentState, char nextInput,
                                    char lowestChar = 'a') {
    int answer = currentState;
    int c = nextInput - lowestChar;
    while (g[answer][c] == -1) answer = f[answer];
    return g[answer][c];
}


// How to use this algorithm:
//
// 1. Modify the MAXS and MAXC constants as appropriate.
// 2. Call buildMatchingMachine with the set of keywords to
//    search for.
// 3. Start at state 0. Call findNextState to incrementally
//    transition between states.
// 4. Check the out function to see if a keyword has been
//    matched.
//
// Example:
//
```

```
// Assume keywords is a vector that contains
// {"he", "she", "hers", "his"} and text is a string that
// contains "ahishers".
//
// Consider this program:
//
// buildMatchingMachine(keywords, 'a', 'z');
// int currentState = 0;
// for (int i = 0; i < text.size(); ++i) {
//    currentState = findNextState(currentState, text[i], 'a');
//
//    Nothing new, let's move on to the next character.
//    if (out[currentState] == 0) continue;
//
//    for (int j = 0; j < keywords.size(); ++j) {
//        if (out[currentState] & (1 << j)) {
//            //Matched keywords[j]
//            cout << "Keyword " << keywords[j]
//                 << " appears from "
//                 << i - keywords[j].size() + 1
//                 << " to " << i << endl;
//        }
//    }
// }
//
// The output of this program is:
//
// Keyword his appears from 1 to 3
// Keyword he appears from 4 to 5
// Keyword she appears from 3 to 5
// Keyword hers appears from 4 to 7

///////////////////////////////////////////////////////////////
//              End of Aho-Corasick's algorithm             //
///////////////////////////////////////////////////////////////
```

....

6.3. Suffix arrays y longest common prefix

```
////////////////////////////////////////////////////////////////
//             Begins Suffix Arrays implementation            //
////////////////////////////////////////////////////////////////
//             O(n log n) - Manber and Myers algorithm        //
//      Refer to "Suffix arrays: A new method for on-line     //
//          string searches", by Udi Manber and Gene Myers    //
////////////////////////////////////////////////////////////////
//Usage:
// - Fill str with the characters of the string.
// - Call SuffixSort(n), where n is the length of the string
//   stored in str.
// - That's it!

//Output:
// pos = The suffix array. Contains the n suffixes of str 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]

const int N = 1000005; // Maximum size of input string
int str[N]; //input
int rank[N], pos[N]; //output
int cnt[N], next[N]; //internal
bool bh[N], b2h[N];

// Compares two suffixes according to their first characters
bool smaller_first_char(int a, int b){
  return str[a] < str[b];
}

void suffixSort(int n){
  //sort suffixes according to their first characters
  for (int i=0; i<n; ++i){
    pos[i] = i;
  }
  sort(pos, pos + n, smaller_first_char);
  //{pos contains the list of suffixes sorted by their first
```

```
//character}

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){
  //{bh[i] == false if the first h characters of pos[i-1] ==
  // the first h characters of pos[i]}
  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; // We are done! Lucky bastards!
  //{suffixes are separted in buckets containing strings
  // starting with the same h characters}

  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 Arrays implementation           //
/////////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////////
//         Begin of Longest Common Prefix implementation       //
/////////////////////////////////////////////////////////////////
//                             O(n)                            //
//   Refer to "Linear-Time Longest-Common-Prefix Computation   //
//     in Suffix Arrays and Its Applications" by Toru Kasai,   //
// Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park.//
/////////////////////////////////////////////////////////////////

int height[N];
// - height[i] = length of the longest common prefix of suffix
//   pos[i] and suffix 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--;
  }
 }
}
////////////////////////////////////////////////////////////////////
//         End of Longest Common Prefix implementation            //
////////////////////////////////////////////////////////////////////
```

....

7. Geometría

7.1. Área de un polígono

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

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

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

....

7.2. Centro de masa de un polígono

Si P es un polígono simple (no se intersecta a sí mismo) su centro de masa está dado por:

$$\bar{C}_x = \frac{\iint_R x\,dA}{M} = \frac{1}{6M}\sum_{i=1}^{n}(y_{i+1} - y_i)(x_{i+1}^2 + x_{i+1} \cdot x_i + x_i^2)$$

$$\bar{C}_y = \frac{\iint_R y\,dA}{M} = \frac{1}{6M}\sum_{i=1}^{n}(x_i - x_{i+1})(y_{i+1}^2 + y_{i+1} \cdot y_i + y_i^2)$$

Donde M es el área del polígono.

Otra posible fórmula equivalente:

$$\bar{C}_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

$$\bar{C}_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

7.3. 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;
}
```

....

7.4. Convex hull: Andrew's monotone chain

Complejidad: $O(n \log_2 n)$

```
// Convex Hull: Andrew's Monotone Chain Convex Hull
// Complexity: O(n log n) (But lower constant than Graham Scan)

typedef long long CoordType;

struct Point {
  CoordType x, y;
  bool operator <(const Point &p) const {
    return x < p.x || (x == p.x && y < p.y);
  }
};

// 2D cross product.  Returns a positive value, if OAB makes a
// counter-clockwise turn, negative for clockwise turn, and zero
// if the points are collinear.
CoordType cross(const Point &O, const Point &A, const Point &B){
  return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

// Returns a list of points on the convex hull in
// counter-clockwise order.  Note: the last point in the returned
// list is the same as the first one.
vector<Point> convexHull(vector<Point> P){
  int n = P.size(), k = 0;
  vector<Point> H(2*n);
  // Sort points lexicographically
  sort(P.begin(), P.end());
  // Build lower hull
  for (int i = 0; i < n; i++) {
```

```
    while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
    H[k++] = P[i];
  }
  // Build upper hull
  for (int i = n-2, t = k+1; i >= 0; i--) {
    while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
    H[k++] = P[i];
  }
  H.resize(k);
  return H;
}
```

7.5. 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);
}
```

7.6. 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);
}
```

7.7. Determinar si un polígono es convexo

```
/*
  Returns positive if a-b-c makes a left turn.
  Returns negative if a-b-c makes 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;
}
```

....

7.8. 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;
}
```

....

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

Field-testing:

- *TopCoder* - SRM 187 - Division 2 Hard - PointInPolygon
- *UVa* - 11665 - Chinese Ink

```
//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 );
}
```

7.10. Hallar la 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);
  }
}
```

7.11. Hallar la intersección de dos segmentos de recta

Field-testing:

- *UVa* - 11665 - Chinese Ink

```
/*
  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;
  }
}
```

....

7.12. Determinar si dos segmentos de recta se intersectan o no

Si sólo se necesita determinar si dos segmentos se intersectan, pero no hallar el punto de intersección, se puede usar este código que es más corto que 7.11. La función `point_in_box` es la misma que en 7.11.

```
/*
  Returns the cross product of the segment that goes from
  (x1, y1) to (x3, y3) with the segment that goes from
  (x1, y1) to (x2, y2)
*/
int direction(int x1, int y1, int x2, int y2, int x3, int y3) {
    return (x3 - x1) * (y2 - y1) - (y3 - y1) * (x2 - x1);
}

/*
  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(int x1, int y1,
                                  int x2, int y2,

                                  int x3, int y3,
                                  int x4, int y4){

    int d1 = direction(x3, y3,   x4, y4,   x1, y1);
    int d2 = direction(x3, y3,   x4, y4,   x2, y2);
    int d3 = direction(x1, y1,   x2, y2,   x3, y3);
    int d4 = direction(x1, y1,   x2, y2,   x4, y4);
    bool b1 = d1 > 0 and d2 < 0 or d1 < 0 and d2 > 0;
    bool b2 = d3 > 0 and d4 < 0 or d3 < 0 and d4 > 0;
    if (b1 and b2) return true;
    if (d1 == 0 and point_in_box(x1, y1,     x3, y3, x4, y4))
        return true;

    if (d2 == 0 and point_in_box(x2, y2,     x3, y3, x4, y4))
        return true;

    if (d3 == 0 and point_in_box(x3, y3,     x1, y1, x2, y2))
        return true;

    if (d4 == 0 and point_in_box(x4, y4,     x1, y1, x2, y2))
        return true;

    return false;
}
```

8. Estructuras de datos

8.1. 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 $O(log_2 n)$.

```
class FenwickTree{
  vector<long long> v;
  int maxSize;

public:
  FenwickTree(int _maxSize) : maxSize(_maxSize+1) {
    v = vector<long long>(maxSize, 0LL);
  }

  void add(int where, long long what){
    for (where++; where <= maxSize; where += where & -where){
      v[where] += what;
    }
  }

  long long query(int where){
    long long sum = v[0];
    for (where++; where > 0; where -= where & -where){
      sum += v[where];
    }
    return sum;
  }

  long long query(int from, int to){
    return query(to) - query(from-1);
  }

};
```

8.2. Segment tree

```
class SegmentTree{
public:
  vector<int> arr, tree;
  int n;

  SegmentTree(){}
  SegmentTree(const vector<int> &arr) : arr(arr) {
    initialize();
  }

  //must be called after assigning a new arr.
  void initialize(){
    n = arr.size();
    tree.resize(4*n + 1);
    initialize(0, 0, n-1);
  }

  int query(int query_left, int query_right) const{
    return query(0, 0, n-1, query_left, query_right);
  }

  void update(int where, int what){
    update(0, 0, n-1, where, what);
  }

private:
  int initialize(int node, int node_left, int node_right);
  int query(int node, int node_left, int node_right,
            int query_left, int query_right) const;
  void update(int node, int node_left, int node_right,
              int where, int what);
};

int SegmentTree::initialize(int node,
                            int node_left, int node_right){
  if (node_left == node_right){
    tree[node] = node_left;
    return tree[node];
  }
  int half = (node_left + node_right) / 2;
  int ans_left = initialize(2*node+1, node_left, half);
  int ans_right = initialize(2*node+2, half+1, node_right);

  if (arr[ans_left] <= arr[ans_right]){
    tree[node] = ans_left;
  }else{
    tree[node] = ans_right;
  }
  return tree[node];
}

int SegmentTree::query(int node, int node_left, int node_right,
                       int query_left, int query_right) const{
  if (node_right < query_left || query_right < node_left)
    return -1;
  if (query_left <= node_left && node_right <= query_right)
    return tree[node];

  int half = (node_left + node_right) / 2;
  int ans_left = query(2*node+1, node_left, half,
                       query_left, query_right);
  int ans_right = query(2*node+2, half+1, node_right,
                        query_left, query_right);

  if (ans_left == -1) return ans_right;
  if (ans_right == -1) return ans_left;

  return (arr[ans_left] <= arr[ans_right] ? ans_left : ans_right);
}

void SegmentTree::update(int node, int node_left, int node_right,
                         int where, int what){
  if (where < node_left || node_right < where) return;
  if (node_left == where && where == node_right){
    arr[where] = what;
    tree[node] = where;
    return;
  }
  int half = (node_left + node_right) / 2;
  if (where <= half){
    update(2*node+1, node_left, half, where, what);
```

```
  }else{
    update(2*node+2, half+1, node_right, where, what);
  }
  if (arr[tree[2*node+1]] <= arr[tree[2*node+2]]){
    tree[node] = tree[2*node+1];
  }else{
    tree[node] = tree[2*node+2];
  }
}
```

9. Misceláneo

9.1. El *parser* más rápido del mundo

- Cada no-terminal: un método
- Cada lado derecho:
  - invocar los métodos de los no-terminales o
  - Cada terminal: invocar proceso *match*
- Alternativas en una producción: se hace un *if*

No funciona con gramáticas recursivas por izquierda ó en las que en algún momento haya varias posibles escogencias que empiezan por el mismo caracter (En ambos casos la gramática se puede factorizar).

Ejemplo: Para la gramática:

$$A \longrightarrow (A)A$$

$$A \longrightarrow \epsilon$$

```
//A -> (A)A | Epsilon

#include <iostream>
#include <string>

using namespace std;

bool ok;
char sgte;
int i;
string s;

bool match(char c){
  if (sgte != c){
    ok = false;
  }
  sgte = s[++i];
}

void A(){
  if (sgte == '('){
    match('(');
    A(); match(')'); A();
  }else if (sgte == '$' || sgte == ')'){
    //nada
  }else{
    ok = false;
  }
}

int main(){
  while(getline(cin, s) && s != ""){
    ok = true;
    s += '$';
    sgte = s[(i = 0)];
    A();
    if (i < s.length()-1) ok = false; //No consumi toda la cadena
    if (ok){
      cout << "Accepted\n";
    }else{
      cout << "Not accepted\n";
    }

  }
}
```

9.2. Checklist para corregir un Wrong Answer

Consideraciones que podrían ser causa de un Wrong Answer:

- Overflow.
- El programa termina anticipadamente por la condición en el ciclo de lectura. Por ejemplo, se tiene `while (cin >> n >> k && n && k)` y un caso válido de entrada es n = 1 y k = 0.
- El grafo no es conexo.
- Puede haber varias aristas entre el mismo par de nodos.
- Las aristas pueden tener costos negativos.
- El grafo tiene un sólo nodo.
- La cadena puede ser vacía.
- Las líneas pueden tener espacios en blanco al principio o al final (Cuidado al usar `getline` o `fgets`).
- El arreglo no se limpia entre caso y caso.
- Estás imprimiendo una línea en blanco con un espacio (`printf(" \n")` en vez de `printf("\n")` ó `puts(" ")` en vez de `puts("")`).
- Hay pérdida de precisión al leer variables como `double` y converterirlas a enteros. Por ejemplo, en C++, `floor(0.29 * 100) == 28`.
- La rana se puede quedar quieta.
- El producto cruz está invertido. Realmente es $|< a_x, a_y > \times < b_x, b_y >| = a_x \mathbf{b_y} - a_y \mathbf{b_x} \neq a_x \mathbf{b_x} - a_y \mathbf{b_y}$.

9.3. Redondeo de dobles

Para redondear un doble a k cifras, usar:

$$\frac{\lfloor 10^k \cdot x + 0{,}5 \rfloor}{10^k}$$

Ejemplo:

```
double d = 1.2345;
d = floor(1000 * d + 0.5) / 1000;
```

Al final, `d` es 1.235.

9.3.1. Convertir un doble al entero más cercano

Código	*Valores originales (d)*	*Nuevos valores (k)*
`int k = floor(d + 0.5 + EPS);` (con `EPS = 1e-9`)	0.0	0
	0.1	0
	0.5	1
	0.4999999999999999	1
	`cos(1e-7) * 0.5` = 0.4999999999999975	1
	0.9	1
	1.0	1
	1.4	1
	1.5	2
	1.6	2
	1.9	2
	2.0	2
	2.1	2
	-0.0	0
	-0.1	0
	-0.5	0
	-0.4999999999999999	0
	`-cos(1e-7) * 0.5` = -0.4999999999999975	0
	-0.9	-1
	-1.0	-1
	-1.4	-1
	-1.5	-1
	-1.6	-2
	-1.9	-2
	-2.0	-2
	-2.1	-2

Código	*Valores originales (d)*	*Nuevos valores (k)*
`int k = floor(d + 0.5);`	0.0	0
	0.1	0
	0.5	1
	0.4999999999999999	0
	`cos(1e-7) * 0.5` = 0.4999999999999975	0
	0.9	1
	1.0	1
	1.4	1
	1.5	2
	1.6	2
	1.9	2
	2.0	2
	2.1	2
	-0.0	0
	-0.1	0
	-0.5	0
	-0.4999999999999999	0
	`-cos(1e-7) * 0.5` = -0.4999999999999975	0
	-0.9	-1
	-1.0	-1
	-1.4	-1
	-1.5	-1
	-1.6	-2
	-1.9	-2
	-2.0	-2
	-2.1	-2

9.3.2. Redondear un doble a cierto número de cifras de precisión

10. Java

10.1. Entrada desde entrada estándar

Este primer método es muy fácil pero es mucho más ineficiente porque utiliza Scanner en vez de BufferedReader:

```
import java.io.*;
import java.util.*;

class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()){
            String s= sc.nextLine();
            System.out.println("Leí: " + s);
        }
    }
}
```

....

Este segundo es más rápido:

```
import java.util.*;
import java.io.*;
import java.math.*;

class Main {
    public static void main(String[] args) throws IOException {
      BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
      String line = reader.readLine();
      StringTokenizer tokenizer = new StringTokenizer(line);
      int N = Integer.valueOf(tokenizer.nextToken());
      while (N-- > 0){
        String a, b;
        a = reader.readLine();
        b = reader.readLine();

        int A = a.length(), B = b.length();
```

```
      if (B > A){
        System.out.println("0");
      }else{
        BigInteger dp[][] = new BigInteger[2][A];
        /*
          dp[i][j] = cantidad de maneras diferentes
          en que puedo distribuir las primeras i
          letras de la subsecuencia (b) terminando
          en la letra j de la secuencia original (a)
        */

        if (a.charAt(0) == b.charAt(0)){
          dp[0][0] = BigInteger.ONE;
        }else{
          dp[0][0] = BigInteger.ZERO;
        }
        for (int j=1; j<A; ++j){
          dp[0][j] = dp[0][j-1];
          if (a.charAt(j) == b.charAt(0)){
            dp[0][j] = dp[0][j].add(BigInteger.ONE);
          }
        }

        for (int i=1; i<B; ++i){
          dp[i%2][0] = BigInteger.ZERO;
          for (int j=1; j<A; ++j){
            dp[i%2][j] = dp[i%2][j-1];
            if (a.charAt(j) == b.charAt(i)){
              dp[i%2][j] = dp[i%2][j].add(dp[(i+1)%2][j-1]);
            }
          }
        }
        System.out.println(dp[(B-1)%2][A-1].toString());
      }
    }
  }
}
```

....

10.2. Entrada desde archivo

```
import java.io.*;
import java.util.*;
public class BooleanTree {
public static void main(String[] args) throws FileNotFoundException {
System.setIn(new FileInputStream("tree.in"));
System.setOut(new PrintStream("tree.out"));
Scanner reader = new Scanner(System.in);
N = reader.nextInt();
for (int c = 1; c <= N; ++c) {
int res = 100;
if (res < 1000)
System.out.println("Case #" + c + ": " + res);
else
System.out.println("Case #" + c + ": IMPOSSIBLE");
}
}
}
```

....

10.3. Mapas y sets

Programa de ejemplo:

```
import java.util.*;

public class Ejemplo {
    public static void main(String[] args){
        /*
         * Mapas
         * Tanto el HashMap como el TreeMap funcionan,
         * pero tienen diferentes detalles
         * y difieren en algunos métodos (Ver API).
         */
        System.out.println("Maps");
        //TreeMap<String, Integer> m = new TreeMap<String, Integer>();
        HashMap<String, Integer> m = new HashMap<String, Integer>();
        m.put("Hola", new Integer(465));
        System.out.println("m.size() = " + m.size());

        if (m.containsKey("Hola")){
```

```
            System.out.println(m.get("Hola"));
        }

        System.out.println(m.get("Objeto inexistente"));


        /*
         * Sets
         * La misma diferencia entre TreeSet y HashSet.
         */
        System.out.println("\nSets");
        /*
         * *OJO: El HashSet no está en orden, el TreeSet sí.
         */
        //HashSet<Integer> s = new HashSet<Integer>();
        TreeSet<Integer> s = new TreeSet<Integer>();
        s.add(3576);
        s.add(new Integer("54"));
        s.add(new Integer(1000000007));

        if (s.contains(54)){
            System.out.println("54 presente.");
        }

        if (s.isEmpty() == false){
            System.out.println("s.size() = " + s.size());
            Iterator<Integer> i = s.iterator();
            while (i.hasNext()){
                System.out.println(i.next());
                i.remove();
            }
            System.out.println("s.size() = " + s.size());
        }
    }
}
```

....

La salida de este programa es:

```
Maps
m.size() = 1
465
null

Sets
54 presente.
s.size() = 3
54
3576
1000000007
s.size() = 0
```

Si quiere usarse una clase propia como llave del mapa o como elemento del set, la clase debe implementar algunos métodos especiales: Si va a usarse un TreeMap ó TreeSet hay que implementar los métodos `compareTo` y `equals` de la interfaz `Comparable` como en la sección 10.4. Si va a usarse un HashMap ó HashSet hay más complicaciones.
Sugerencia: Inventar una manera de codificar y decodificar la clase en una String o un Integer y meter esa representación en el mapa o set: esas clases ya tienen los métodos implementados.

10.4. Colas de prioridad

Hay que implementar unos métodos. Veamos un ejemplo:

```
import java.util.*;


class Item implements Comparable<Item>{
    int destino, peso;

    Item(int destino, int peso){
        this.peso = peso;
        this.destino = destino;
    }
    /*
     * Implementamos toda la javazofia.
     */
```

```
    public int compareTo(Item otro){
        // Return < 0 si this < otro
        // Return   0 si this == otro
        // Return > 0 si this > otro
        /* Un nodo es menor que otro si tiene menos peso */
        return peso - otro.peso;
    }
    public boolean equals(Object otro){
        if (otro instanceof Item){
            Item ese = (Item)otro;
            return destino == ese.destino && peso == ese.peso;
        }
        return false;
    }
    public String toString(){
        return "peso = " + peso + ", destino = " + destino;
    }
}

class Ejemplo {
    public static void main(String[] args) {
        PriorityQueue<Item> q = new PriorityQueue<Item>();
        q.add(new Item(12, 0));
        q.add(new Item(4, 1876));
        q.add(new Item(13, 0));
        q.add(new Item(8, 0));
        q.add(new Item(7, 3));
        while (!q.isEmpty()){
            System.out.println(q.poll());
        }

    }
}
```

La salida de este programa es:

```
peso = 0, destino = 12
peso = 0, destino = 8
peso = 0, destino = 13
peso = 3, destino = 7
peso = 1876, destino = 4
```

Vemos que la función de comparación que definimos no tiene en cuenta `destino`, por eso no desempata cuando dos `Items` tienen el mismo `peso` si no que escoge cualquiera de manera arbitraria.

11. C++

11.1. Entrada desde archivo

```
#include <iostream>
#include <fstream>

using namespace std;

int _main(){
  freopen("entrada.in", "r", stdin);
  freopen("entrada.out", "w", stdout);

  string s;
  while (cin >> s){
    cout << "Leí " << s << endl;
  }
  return 0;
}


int main(){
  ifstream fin("entrada.in");
  ofstream fout("entrada.out");

  string s;
  while (fin >> s){
    fout << "Leí " << s << endl;
  }
  return 0;
}
```

11.2. Strings con caractéres especiales

```
#include <iostream>
#include <cassert>
#include <stdio.h>
#include <assert.h>
#include <wchar.h>
#include <wctype.h>
#include <locale.h>

using namespace std;

int main(){
  assert(setlocale(LC_ALL, "en_US.UTF-8") != NULL);
  wchar_t c;

  wstring s;
  while (getline(wcin, s)){
    wcout << L"Leí : " << s << endl;
    for (int i=0; i<s.size(); ++i){
      c = s[i];
      wprintf(L"%lc %lc\n", towlower(s[i]), towupper(s[i]));
    }
  }

  return 0;
}
```

Nota: Como alternativa a la función getline, se pueden utilizar las funciones fgetws y fputws, y más adelante swscanf y wprintf:

```
#include <iostream>
#include <cassert>
#include <stdio.h>
#include <assert.h>
#include <wchar.h>
#include <wctype.h>
#include <locale.h>
using namespace std;
int main(){
  assert(setlocale(LC_ALL, "en_US.UTF-8") != NULL);
  wchar_t in_buf[512], out_buf[512];
  swprintf(out_buf, 512,
           L"¿Podrías escribir un número?, Por ejemplo %d. "
           "¡Gracias, pingüino español!\n", 3);
  fputws(out_buf, stdout);
  fgetws(in_buf, 512, stdin);
  int n;
  swscanf(in_buf, L"%d", &n);
  swprintf(out_buf, 512,
           L"Escribiste %d, yo escribo ¿ÔÏàÚÑ~\n", n);
  fputws(out_buf, stdout);
  return 0;
}
```

11.3. Imprimir un doble con `cout` con cierto número de cifras de precisión

Tener cuidado con números negativos, porque el comportamiento es diferente.

```
#include <iomanip>

cout << fixed << setprecision(3) << 1.1225 << endl;
```