Manual de algoritmos del semillero de programación EAFIT

Ana Echavarría

22 de mayo de 2013

Índice

1. Plantilla

```
using namespace std;
#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 <stack>
#include <list>
#include <map>
#include <set>

// Template para recorrer contenedores usando iteradores
```

```
#define foreach(x, v) for (typeof (v).begin() x=(v).begin(); \
                           x !=(v).end(); ++x)
// Template que imprime valores de variables para depurar
#define D(x) cout << #x " = " << (x) << endl

// Función para comparar dos dobles sin problemas de presición
// Retorna -1 si x < y, 0 si x = y, 1 si x > y
const double EPS = 1e-9;
int cmp (double x, double y, double tol = EPS){
    return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}


int main() {
    // Entrada y salida desde / hacia archivo
    // Eliminar si la entrada es estándar
    // Cambiar in.txt / out.txt por los archivos de entrada/salida
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);

    return 0;
}
```

.....

2. Grafos

2.1. 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);
         }
      }
   }
}
```

.....

2.2. 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);
}
```

.....

2.3. 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());
}
```

.....

2.4. 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.
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;
}
```

2.5. 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;
}
```

2.6. 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.

```
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;
}
```

2.7. 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
```

2.7.1. 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]);`

2.7.2. 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]) );`

2.7.3. 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]) )`

2.8. 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;
}
```

.....

2.9. Algoritmo de Kruskal

2.9.1. 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;
}
```

.....

2.9.2. 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;
}
```

.....

2.10. 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 super-fuente / super-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 al grafo la red de flujos
```

```
    // 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. Teoría de números

3.1. 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);
}
```

.....

3.2. 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: $\text{lcm}(a,b) = \dfrac{|a \cdot b|}{\gcd(a,b)}$

3.3. Criba de Eratóstenes

Encuentra los primos desde 1 hasta un límite n.
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);
    }
}
```

.....

3.4. 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;
}
```

.....

3.5. Exponenciación logarítmica

3.5.1. Propiedades de la operación módulo

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

3.5.2. 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;
```

```
    }
}
```

3.6. Combinatoria

3.6.1. 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];
        }
    }
}
```

3.6.2. 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

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

4. Programación dinámica

4.1. 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. Strings

5.1. 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.

```
// Máximo tamaño de los strings
const int MAXN = 1005;
// dp[i][j] lcs de los substrings s[0..i-1] y t[0..j-1]
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){
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            if (s[i-1] == t[j-1]){
                dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
            }
        }
    }
    return dp[n][m];
}
```

5.2. Longest increasing subsequence

Halla la longitud de la subsecuencia creciente más larga que hay en un string s (también se puede usar con vectores).
Complejidad: $O(n^2)$ donde n es la longitud de s.

```
// Máximo tamaño del string
const int MAXN = 1005;
// dp[i] = la longitud de la máxima subsecuencia creciente que
// termina en el caracter i (usándolo)
int dp[MAXN];

int lis(const string &s){
    int n = s.size();
    dp[0] = 1;
    for (int i = 1; i < n; i++){
        dp[i] = 1;
        for (int j = 0; j < i; j++){
            // Cambiar el < por <= si la secuencia tiene que ser
            // es estrictamente creciente
            if (s[j] <= s[i]) dp[i] = max(dp[i], dp[j] + 1);
        }
    }

    int best = 0;
    for (int i = 0; i < n; i++) best = max(best, dp[i]);
    return best;
}
```

5.3. 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;
}
```