Apunte ACM

Equipos FaMAF

1 de noviembre de 2003

Índice

1. Plantilla

Muchos algoritmos usan estas macros.

```
// Ciclos
#define FOR(i,m,n) for (int i=(m);i<(n);i++)
// Inicialización de arrays
#define INIT(a,n,v) do {FOR (i_,0,(n)) a[i_]=(v);} while(0)

#define mod(a, b) ((a)%(b)<0 ? (a)%(b)+(b) : (a)%(b))
```

2. Estructuras de datos

2.1. Lista enlazada

Las listas enlazadas son secuencias de nodos conectado cada uno al siguiente. La representación usual es con cada nodo con un puntero al siguiente nodo, pero hay varias variantes (muchas de ellas ortogonales) a esta representación

1. Agregar un nodo "centinela" al principio y/o al final, para evitar casos particulares en varias operaciones. Suele ser conveniente que el centinela final apunte a si mismo.

2. El nodo final puede estar enlazado al inicial (lista enlazada circular) o alguno de los intermedios. (Lista enlazada circular)

3. Pueden compartirse las colas entre varias listas enlazadas (formando en realidad un up-tree)

4. Los nodos pueden tener enlaces hacia atrás (Lista doblemente enlazada)

5. El alojamiento puede ser dentro de un array (los enlaces son índices en vez de punteros).

Un ejemplo de (3,5) puede verse en el algoritmo de union-find (componentes conexas de grafos). Auí hay un ejemplo básico de una lista con las variantes (1,2), independiente de estructura subyacente.

```
#include "../template.h"

//typedef ?? item;
//typedef ?? link_t;
//struct linked_node;
//typedef ?? list_ref;

void ll_delete_after(list_ref l, link_t t) {
  link_t del = next(l,t) ;
  next(l,t)=next(next(l,t)) ;
  ll_release(l,del);
}

void ll_insert_after (list_ref l, link_t t, item v) {
  link_t n = ll_new(l);
  item(l,n) = v;
  next(l,n) = next(l,t);
  next(l,t) = n;
}
```

Con las siguientes definiciones, la lista anterior usa la variante 5 (representación en array):

```
#include "../template.h"

#define N 100

typedef (void *) item;
typedef int link_t;
struct linked_node {
  item i;
```

```
  link_t next;
}

struct linked_list {
  link_t head; // index of head (dummy) node
  link_t tail; // index of tail (dummy) node
  link_t free; // index of first free node in free chain.
  linked_node storage[N]; // node storage
}

typedef linked_list &list_ref;

// Begin specific part for arrayed linked list
ll_init(list_ref r) {
  r.head = 0 ; r.tail = 1; r.free = 2;
  r.storage.next[r.head] = r.storage.next[r.tail] = r.tail ;
  FOR (i,2,N) r.storage[i].next = i+1;
  r.storage[N-1]=-1;
}

void ll_release (list_ref l, link_t t) {
  l.storage[t].next = l.free ;
  l.free = t ;
}

link_t ll_new(list_ref l) {
  link_t r = l.free ; // if -1, we're out of space
  l.free = l.storage[l.free].next ;
  return r ;
}
#define next(l,t) ((l).storage[t].next)
#define item(l,t) ((l).storage[t].item)
```

En cambio, con estas definiciones, la representación es la usual (los enlaces son punteros a nodos en el heap):

```
#include <stdlib.h>
#include "../template.h"
```

```
typedef (void *) item;

struct linked_node ;
typedef struct linked_node *link_t;
typedef struct linked_node {
  item i;
  link_t next;
}

struct linked_list {
  link_t head; // index of head (dummy) node
  link_t tail; // index of tail (dummy) node
}

typedef linked_list *list_ref;

// Begin specific part for arrayed linked list
ll_init(list_ref r) {
  r->head = malloc(sizeof(*(r->head))) ;
  r->tail = malloc(sizeof(*(r->head))) ;
  r->head->next = r->tail->next = r->tail;
}

void ll_release (list_ref l, link_t t) {
  free(t);
}

link_t ll_new(list_ref l) {
  return malloc (sizeof(*link_t)) ;
}
#define next(l,t) ((t)->next)
#define item(l,t) ((t)->item)
```

Notar que para los caso más simple, C++ provee el contenedor `slist<T>`, una lista enlazada, o `list<T>`, una lista doblemente enlazada, con muchas operaciones ya implementadas

2.2. Stack

Un stack es una estructura secuencial donde las inserciones, borrados y accesos son siempre en un mismo extremo.

Puede implementarse trivialmente sobre listas enlazadas, o arrays. La STL provee una clase `stack<T>` que probablemente sea suficiente para todo uso.

Es practico para manejar estructuras anidadas/recursivas, por ejemplo recorridas en árboles, evaluación de expresiones implementaciones iterativas de algoritmos recursivos, etc.

2.3. Queue

Una queue es una estructura secuencial donde las inserciones son siempre en un extremo, y los borrados y accesos siempre en el otro.

Una implementación posible es usando listas enlazadas, teniendo un puntero al final (para las inserciones). Otra variante es una estructura indexable (por ej, array), con indices al primer y siguiente-al-último elemento, y se hace la cola circular. Notar que la cola esta vacía cuando $head = tail$, y llena cuando $head = tail \oplus 1$, con lo que se desperdicia un lugar siempre. Otra variante es tener $head$ y $size$.

2.4. Árboles

2.4.1. Definiciones

Path length: Suma de las alturas de todos los nodos (la raíz tiene altura 0).

Internal path length: Suma de las alturas de todos los nodos internos.

External path length: Suma de las alturas de todos los nodos externos.

2.4.2. Propiedades

- Hay exactamente un camino conectando dos nodos cualesquiera en el árbol.
- Un árbol con N nodos tiene $N - 1$ aristas.
- Un árbol binario con N nodos internos tiene $N + 1$ nodos externos.
- La EPL de un árbol binario con N nodos supera en $2N$ a la IPL

2.4.3. Representaciones

Para un árbol n-ario, con n fijo, se puede tener cada nodo con n punteros a los hijos. NULL o un centinela representa la ausencia de un hijo dado.

Si no hay cota, puede usarse alguna secuencia dinámica (listas, por ej.) en cada nodo para listar sus hijos.

Si en el ejemplo anterior, se "colapsa" la estructura superponiendo los nodos de arbol con nodos de lista enlazada, se ve que en realidad cada nodo puede tener dos punteros, uno al primer hijo, y otro al siguiente hermano. De esta forma se puede representar cualquier bosque como árbol binario.

En muchos casos lo que hace falta es recorrer el árbol hacia arriba. En ese caso, cada nodo puede tener un enlace a su padre. En estos casos, uno normalmente tiene las hojas, o todos los nodos en alguna estructura agregada (array, o lista), para poder de ahí llegar al resto de los nodos. Un ejemple de esto es el algoritmo de union-find en la sección de conectitud de grafos. Estas estructuras normalmente son llamadas "up-trees"

Notar que muchas de las variantes posibles en listas enlazadas, siguen siendo posibles en árboles, aunque no suelen ser tan prácticas.

2.4.4. Traversing

Esta parte asume árboles binarios. Normalmente puede generalizarse haciendo las operaciones que se hacen a la izquierda y a la derecha en operaciones para todos los nodos.

La estructura usada es:

```
typedef void *item ;
struct node {
  item i;
  node *l, *r;
};
```

Preorden

```
#include <stack>
#include "tree.cc"

void preorder (node *t) {
  stack <node *> s ;
  s.push(t) ;
  while (!s.empty()) {
```

```
    t = s.top(); s.pop();
    // hacer algo con t
    if (t) s.push(t->r);
    if (t) s.push(t->l);
  }
}
```

Inorden

```
#include <stack>
#include <utility>
#include "tree.cc"

void inorder (node *t) {
  stack <pair <bool, node *> > s ;
  bool b = false ;
  if (t) s.push (make_pair(false, t->r)) ;
  s.push (make_pair(true, t)) ;
  if (t) s.push (make_pair(false, t->l)) ;
  while (!s.empty()) {
    b = s.top().first ;
    t = s.top().second ;
    s.pop();
    if (b) {
      // hacer algo con t
    } else {
      if (t) s.push (make_pair(false, t->r)) ;
      s.push (make_pair(true, t)) ;
      if (t) s.push (make_pair(false, t->l)) ;
    }
  }
}
```

Postorden

```
#include <stack>
#include <utility>
#include "tree.cc"

void postorder (node *t) {
```

```
  stack <pair <bool, node *> > s ;
  bool b = false ;
  s.push (make_pair(true, t)) ;
  if (t) s.push (make_pair(false, t->r)) ;
  if (t) s.push (make_pair(false, t->l)) ;
  while (!s.empty()) {
    b = s.top().first ;
    t = s.top().second ;
    s.pop();
    if (b) {
      // hacer algo con t
    } else {
      s.push (make_pair(true, t)) ;
      if (t) s.push (make_pair(false, t->r)) ;
      if (t) s.push (make_pair(false, t->l)) ;
    }
  }
}
```

Orden por nivel

```
#include <queue>
#include "tree.cc"

void preorder (node *t) {
  queue <node *> q ;
  q.push(t) ;
  while (!q.empty()) {
    t = q.front(); q.pop();
    // hacer algo con t
    if (t) q.push(t->l);
    if (t) q.push(t->r);
  }
}
```

3. C++

3.1. Contenedores

Hay una familia de tipos de contenedores que comparten varias operaciones:

- `iterator begin(), end()`: iteradores al comienzo y post-final del contenedor.
- `size_type size()`: cantidad de elementos
- `bool empty()`: si esta vacío o no
- `constructor()`: constructor de contenedor vacío
- `constructor(contenedor c)`: constructor de copia

3.1.1. Contenedores especializados

Los contenedores hacia adelante tienen un orden de elementos, y tienen operadores relacioneles ((des)igualdad elemento a elemento, orden lexicográfico).

Los contenedores reversibles tienen iteradores invertidos, `rbegin()` y `rend()`.

Los contenedores de acceso aleatorio tienen indexación (operador `[]`, como los arrays de C).

3.2. Secuencias

Una categoría especial de contenedores que preservan orden lineal. Tienen las siguientes operaciones adicionales:

- `constructor(size_type n)`: secuencia de n elementos armados con el constructor básico
- `constructor(size_type n, const T& t)`: secuencia de n elementos copias de t
- `T front()`: primer elemento
- `insert(iterator p,T item)`: inserción de *item* en posición p
- `insert(iterator p,size_type n, T item)`: inserción de n copias de *item* en posición p

- `insert (iterator p, i, j)`: inserción de $[*i... * j]$ en posición p.
- `erase (iterator p)`: borrado en posición p
- `erase (iterator p, q)`: borrado de $[*p... * q]$
- `clear()`

3.2.1. Secuencias especializadas

Las secuencias de inserción por delante agregan el método `push_front()` para inserción, y el método `pop_front()` para borrado.

Las secuencias de inserción por detrás agregan los métodos análogos `push_back()` y `pop_back()`

3.3. Lista enlazada simple

La lista enlazada simple es `slist<T>`. Es más chica y más rápida para operaciones básicas que `list<T>`, pero no tiene backward o reverse iterators. Es una secuencia de inserción por delante.

No usar insert() y erase() en un iterador, usar insert_after() y erase_after(). Los primeros son $O(n)$, los otros $O(1)$. previous() también es $O(n)$

Algunos métodos especiales de este tipo y útiles son: merge(l2) (pasada de merging, $O(n)$), reverse() ($O(n)$), sort() (sort estable, $O(n \log n)$ comparaciones), unique() (remueve elementos consecutivos iguales, $O(n)$), y $<$ (comparación lexicográfica).

3.4. Lista enlazada doble

La lista enlazada doble es `list<T>`. Es más grande y lenta para operaciones básicas que `slist<T>`, pero tiene backward y reverse iterators, y más operaciones que se pueden hacer en $O(1)$. Es un contenedor reversible, y secuencia de inserción por delante y por detrás.

Algunos métodos especiales de este tipo y útiles son: merge(l2) (pasada de merging, $O(n)$), reverse() ($O(n)$), sort() (sort estable, $O(n \log n)$ comparaciones), unique() (remueve elementos consecutivos iguales, $O(n)$)

3.5. Iteradores

3.6. Streams

Tomado de:

http://www.cs.nmsu.edu/~rth/cs/cs177/streamio.html
ostream, istream, iostream are declared in the header file iostream.
cout, cerr, clog are instances of ostream.
cin is an instance of istream.

3.6.1. Format flags

cout and cin contain several single-bit flags for format control. They are named in an enum in class ios as:

```
ios::skipws     // skips whitspace on input
ios::left       // left justification
ios::right      // right justifiction
ios::internal   // pads after sign or base character
ios::dec        // decimal format for integers
ios::oct        // octal format for integers
ios::hex        // hex format for integers
ios::showbase   // show the base character for octal or hex
ios::showpoint  // show the decimal point for all floats
ios::uppercase  // uppercase A-F for hex
ios::showpos    // show +ve sign for numbers
ios::scientific // use exponential notation
ios::fixed      // used oridnary decimal notation
ios::unitbuf    // flush the buffer
```

There are also three combination fields:

```
ios::basefield    = ios::dec | ios::oct | ios::hex
ios::adjustfield  = ios::left | ios::right | ios::internal
ios::floatfield   = ios::scientific | ios::fixed
```

These flags can be set with the member function setf. e.g. :

```
cout.setf(ios::showpos);
```

They can be or'd together:

```
cout.setf(ios::showpos | ios::uppercase);
```

or some bits can be unset while one is being set:

```
cout.setf(ios::oct, ios::dec | ios::oct | ios::hex);
```

This sets the bit for oct, after unsetting the bits for oct, dec and hex, ensuring that only one of the bits is turned on. unsetf turns bits off:

```
cout.unsetf(ios::showpos);
```

3.6.2. Otros modificadores

```
cout << setw(n); // Fija el ancho del _siguiente_ valor a n
cout << setfill(c); // Fija el caracter de rellena a c
cout << setprecision(2); // Fija la cantidad de decimales.
```

3.6.3. Extraction of chars and strings from an input stream

The function get can be used to read any character, including whitespace `cin.get(c);`

Strings can be input teminated by whitespace, or limited by using width. e.g.

```
char s[21];
cin.width(20);
cin >> s;
```

Also, get can be used:

```
char s[21];
cin.get(s, 20);
```

3.6.4. Errors on input / EOF

If an unexpected character is encountered or end-of-file occurs, then the stream is in a failure state and can be tested by: `if (cin.fail()) ...`

Another way to do the same thing is to use the stream itself, converted to an integer: `if (!cin) ...`

This is the same as using the fail function.

3.7. Algoritmos básicos

Poniendo `#include <algorithm>` se tiene acceso a funciones max, min, swap, que hacen lo que uno se imagina.

3.8. Cosas Útiles

3.8.1. Imprimir un array/lista/contenedor

Se puede combinar un output_stream con el algoritmo copy, para imprimir contenedores STL o arrays de esta forma:

```
#include <algorithm>
  int A[] = {8, 3, 6, 1, 2, 5, 7, 4};
  const int N = sizeof(A) / sizeof(int);
  list <int> l (A, A+N); // make from A
  // \n es el separador
  copy(A, A+N, ostream_iterator<int>(cout, "\n"));
  copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
```

3.8.2. Agregarle operadores a un tipo

Ejemplo:

```
struct vec {int x; int y;};

vec operator + (vec v1, vec v2) {
  vec result = {v1.x+v2.x, v1.y+v2.y} ;
  return result ;
}

int main (void) {
  vec u = {2, 5};
  vec v = {3, -1};
  vec w = u + v;
  cout << w.x << ' ' << w.y ;
}
```

Esto suele ser útil para agregar un operador `<` a algo que hay que ordenar.

4. Permutaciones

Bibliografía: Knuth, Matemática Discreta I

Los siguientes algoritmos operan sobre permutaciones de un conjunto $S = \{1..n\}$. Una permutación π es una función biyectiva de $S \to S$.

4.1. Representaciones

Una permutación usualmente se representa como un arreglo p tal que $p[i] = \pi(i)$. A veces se escriben permutaciones como producto de otras permutaciones (donde el producto es la composición de funciones, $\pi\lambda = \lambda \circ \pi$).

Una clase especial de permutaciones son los ciclos. Los ciclos se representan con la notación $(x_0 x_1 x_2 \dots x_{k-1})$, que representa a la permutación π tal que $\pi(x_i) = x_{i \oplus 1}$ ($\oplus$ representa suma módulo k), y es la identidad en el resto de los valores. Dos ciclos son disjuntos si los conjuntos de elementos que afectan son disjuntos.

Es un teorema que "Toda permutación puede escribirse como producto de ciclos disjuntos". Este teorema tiene como consecuencia una representación alternativa de permutaciones, en vez del array. Hay algoritmos para convertir de una representación a la otra.

4.2. Conversión de array a ciclos

Problema: Dado un array representando una permutación π de $\{1..n\}$, escribirla como producto de ciclos disjuntos

Entrada: Un array T tal que $T[i] = \pi(i)$ para i entre 1 y n (notar que es 1-based).

Salida: por salida estándar el producto de ciclos.

Orden: $O(n)$

Notas: el algoritmo destruye el array de entrada T

```
#include "../template.h"
#include <iostream>

using namespace std;

// Muestra T como producto de ciclos ; Destruye T
void showperm_cyclic (int T[], int count) {
  int nt = 0 ;
  int t = 1;
  while (nt < count) {
    for (;T[t]==0;t++)
      ; // T no esta etiquetado
    int i = t;
    if (T[i] == i) {T[i] = 0; nt++;} // ciclo de tamaño 1
    else {
      cout << "(" ;
      while (T[i]>0) {
        int j = i ;
        cout << j << " ";
        i = T[j] ;
        T[j] = 0 ; nt++;
      }
      cout << ")" ;
    }
  }
  cout << endl ;
}
```

4.3. Producto de ciclos

Problema: Dado una permutación expresada como producto de ciclos (no necesariamente disjuntos), transformarla en un array.

Entrada: Un array $input[l]$ que representa la productoria de ciclos. Este arreglo puede contener los valores $LEFT$ o $RIGHT$ que indican comienzo y fin de ciclo, o componentes del ciclo

Salida: almacena en T la permutación resultante π, donde $T[i] = \pi(i)$ para i entre 1 y n (notar que es 1-based).

Orden: $O(l \max n)$

Notas: El algoritmo tiene comportamiento indefinido con la entrada malformada.

```
// Calcula una permutación dada como producto de ciclos
// Knuth, TAOCP, algoritmo 1.3.3.B
#include <algorithm>
#include "../template.h"
const int LEFT = -1 ;
const int RIGHT = -2 ; // abre y cierra parentesis

// input[1] is the input. Values LEFT, RIGHT indicate begin and end of
// cycles. result is stored in T[1]...T[n] (note that this is 1-based)
void permutation_product (const int input[], int l, int T[], int n) {
  FOR (k,1,n+1) T[k] = k ; // T es la identidad
  int p = l-1; // posicion en la secuencia de entrada

  int Z, j ;
  while (p>=0) {
    int i = input[p];
    switch (i) {
    case LEFT:
      T[j] = Z ; break;
    case RIGHT:
      Z = 0; break;
    default:
      swap (T[i], Z) ;
      if (T[i]==0) j = i ;
      break;
    }
    p--;
  }
}
```

4.4. Permutaciones inversas

4.4.1. Versión corta

Dada una permutación T como array, la forma simple de invertirla es usando una tabla auxiliar con el siguiente algoritmo

```
int Y[n+1] ; // Permutacion de salida
FOR (k,1,n+1) Y[T[k]] = k ;
```

4.4.2. Versión in-place

En el caso de no poder usarse una tabla auxiliar, el siguiente algoritmo invierte una permutación X "in-place":

```
// Calcula la inversa de una permutación dada como array, X[1]...X[n]
// NO usa una tabla auxiliar, lo hace inplace
// Knuth, TAOCP, Algoritmo 1.3.3.I

void invert_permutation (int T[], int n) {
  int m,i,j;

  j = -1 ;
  for (m = n; m>0; m--) {
    i = T[m];
    if (i > 0) {
      while (i > 0) {
        T[m] = j;
        j = -m;
        m = i ;
        i = T[m] ;
      }
      i = j ;
    }
    T[m] = -i ;
  }
}
```

4.5. Generación de permutaciones

La STL de C++ provee rutinas para generar/iterar sobre permutaciones, `next_permutation()` y su inverso, `previous_permutation()`. Reciben iteradores de comienzo y fin de la secuencia a permutar. Los elementos de la secuencia deben tener un operador $<$. Las permutaciones se recorren en orden lexicográfico. Normalmente devuelve true, excepto cuando salta de la permutación "mayor" a la permutación "menor".

Ejemplo:

```
template <class BidirectionalIterator>
void snail_sort(BidirectionalIterator first,
                BidirectionalIterator last) {
  while (next_permutation(first, last)) {}
}
```

5. Sorting

5.1. Sorting lineal

5.2. Sorting topológico

Problema: Dado un orden *parcial* $\prec$ en el conjunto $P = \{1..n\}$, obtener un orden *total* $<$ tal que $\forall x, y \in P. x \prec y \Rightarrow x < y$.

Entrada: Un conjunto de pares $S \subseteq \prec$ suficiente para definir $\prec$.

Salida: Una secuencia de valores de P que define el orden $<$.

Orden: $O(|in| \max |P|)$

Notas: El algoritmo es capaz de detectar si la entrada es inconsistente (i.e., hay un ciclo en la relación). Asume que $P = \{x \in \mathcal{N} : 1 \leq x \leq n\}$ para algun n.

```
// Orden topológico
// Toma in, una lista de pares (j,k) en un orden parcial O, con 1<=j,k<=n;
// extiende la lista result con el orden topológico.
//
// Sii la entrada es consistente devuelve true

// Knuth, TAOCP, algorithm 2.2.3.T

#include <list>
#include <algorithm>
#include "template.h"

using namespace std ;
const int n = 9; // size of P

list <int> i1 ; // first elements of input pairs
list <int> i2 ; // second elements of input pairs

bool topological_sort (list <pair <int,int> > in, list <int> &result) {

  int count[n+1] ; // count of direct predecessors of i
  INIT (count,n+1,0);
  int *qlink = count; // qlink is an alias for count
  list <int> succ[n+1]; // direct successors of i

  int remaining = n ;

  // Process input
  list <pair <int,int> >::const_iterator p = in.begin() ;
  while (p != in.end()) {
    count[p->second]++; // Add predecessor of k
    succ[p->first].push_front (p->second); // Add successor of j
    p++;
  }

  // Input data is now in our smart data structure

  int r = 0; // rear of the queue
  qlink[0] = 0;
  FOR (k,1,n+1)
    if (count[k]==0)
      r = qlink[r] = k;
  int f = qlink[0] ; // front of the queue

  while (f!=0) { // queue not empty
    remaining--;
    list <int>::const_iterator p = succ[f].begin() ;
    list <int>::const_iterator q = succ[f].end() ;
    while (p != q) {
      count[*p]--;
      if (count[*p]==0) { // add *p to queue
        qlink[r] = *p ;
        r = *p ;
      }
      p++;
    }
    f = qlink[f] ; // remove f from queue
  }

  // now:
  //  the relation had cycles iff remaining != 0
  // qlink is a chain giving the order, ended by 0

  if (remaining) return false ;
  int q = 0 ;
  while (qlink[q] != 0) {
    result.push_back (qlink[q]);
    q = qlink[q] ;
  }
  return true;
}
```

6. Searching

7. Cadenas

8. Geometría

8.1. Definiciones generales

Esta sección usa las siguientes definiciones de tipos:

```
typedef double coord;

struct point {
  coord x, y;
};

struct line {
  point p1, p2;
};

point mp(coord x, coord y)
{
  point p = {x, y};
  return p;
}

line ml(point p1, point p2)
{
  line l = {p1, p2};
  return l;
}
```

Puede redefinirse coord a otros tipos (int, float, ¿number?). La estructura line se usa tanto para rectas como para segmentos. Si en una línea $p_1 = p_2$ notar que es una línea no válida, y muchas rutinas fallan.

8.2. Distancia

```
#include "geom_defs.cc"

// distance is already used in the STL
double distancia(point p1, point p2)
{
  coord dx = p2.x - p1.x;
  coord dy = p2.y - p1.y;
```

```
  return sqrt(dx*dx + dy*dy);
}
```

Puede usarse para calcular la longitud de un segmento l llamándola como $distancia(l.p_1, l.p_2)$

8.3. Dirección de un ángulo

Problema: Dados tres puntos, p_0, p_1, p_2, determinar si el ángulo de giro yendo de p_0 a p_1 y luego a p_2 es horario o antihorario

Entrada: tres puntos

Salida: -1 si es horario, 1 si es antihorario

Notas: Hay casos especiales para manejar si los puntos están alineados. En ese caso, esta rutina devuelve -1 si p_0 está al medio, 0 si p_1 esta al medio, ó 1 si p_2 está al medio.

```
int ccw(point p0, point p1, point p2)
  /* -1 si el recorrido p0->p1->p2 dobla a la izquierda o
   *    p0 esta entre p2 y p1
   *  1 si dobla a la derecha o p1 esta entre p0 y p2
   *  0 si p2 esta entre p0 y p1
   */
{
  coord dx1 = p1.x - p0.x,
        dx2 = p2.x - p0.x,
        dy1 = p1.y - p0.y,
        dy2 = p2.y - p0.y;

  if (dx1 * dy2 > dy1 * dx2) return 1;
  if (dx1 * dy2 < dy1 * dx2) return -1;
  if (dx1 * dx2 < 0 || dy1 * dy2 < 0) return -1;
  if (dx1 * dx1 + dy1 * dy1 >= dx2 * dx2 + dy2 *dy2) return 0;
  return 1;
}
```

8.4. Intersección

En esta subsección se resuelven dos problemas: existencia de intersección, y calculo del punto de intersección

Problema (intersect): Decidir si dos segmentos se intersecan.

Entrada: dos segmentos

Salida: true o false segun si se intersecan o no.

Notas: Hay dos casos especiales. Uno es que la intersección sea en el extremo de al menos uno de los segmentos. Otra es que los segmentos se superpongan. Esta rutina devuelve true en ambos casos. Si se cambia el $\leq$ por $<$, devuelve false en ambos.

Problema (intersection): Encontrar la intersección de dos rectas.

Entrada: dos rectas

Salida: un punto, o la constante "nopoint"

Notas: El caso especial de las rectas paralelas (lo que incluye cuando son iguales), se maneja devolviendo un valor especial llamado "nopoint" que puede redefinirse.

```
#include "geom_defs.cc"
#include "geom_ccw.cc"

bool intersect(line l1, line l2) /* Segment intersection */
  /* returns true if endpoint touches other segment.
     Change <= for < to return false instead (but also false when segments
 are collinear) */
{
  return ((ccw(l1.p1, l1.p2, l2.p1) * ccw(l1.p1, l1.p2, l2.p2) <= 0) &&
          (ccw(l2.p1, l2.p2, l1.p1) * ccw(l2.p1, l2.p2, l1.p2) <= 0));
}

#define nopoint {-111.1, -111.1}

point intersection(line l1, line l2) /* Returns nopoint on failure */
{
  point r;
  double m;
  if (l1.p1.y==l1.p2.y) { /* l1 is horizontal */
    if (l2.p1.y==l2.p2.y) { /* l2 is horizontal */
      point r = nopoint; return r;
    }
    /* l1 is horizontal l2 not */
    r.y = l1.p1.y;
    m = (l2.p2.x-l2.p1.x)/(l2.p2.y-l2.p1.y);
    r.x = l2.p1.x - m*(l2.p1.y-r.y);
  } else { /* l1 is not horizontal */
    m = (l1.p2.x-l1.p1.x)*(l2.p2.y-l2.p1.y)-(l1.p2.y-l1.p1.y)*(l2.p2.x-l2.
p1.x) ;
    if (m==0) {point r = nopoint; return r;} /* Lines are parallel */
    r.y = ((l1.p2.y-l1.p1.y)*(l2.p1.x*l2.p2.y-l2.p1.y*l2.p2.x)+
           (l2.p2.y-l2.p1.y)*(l1.p1.y*l1.p2.x-l1.p1.x*l1.p2.y))/m;
    r.x = (r.y-l1.p1.y)*(l1.p2.x-l1.p1.x)/(l1.p2.y-l1.p1.y);
  }
  return r;
}
```

9. Grafos

9.1. Representación

Esta sección usa una representación comun para grafos. Para representar aristas, está la siguiente estructura:

```
/* Edge */

typedef int weight;

typedef struct{
  int a;
  int b;
  weight w;
} edge;

edge mkedge (int a, int b, weight w) {
  edge r = {a,b,w} ;
  return r ;
}

bool operator < (edge a, edge b) {
  return a.w > b.w;
}

typedef list<edge>::iterator ei;
```

El peso puede omitirse cuando no se usa.

Hay variantes de representaciones de grafos. Una es la "matriz de adyacencia", una matriz de $V \times V$ (V es la cantidad de vértices) donde se indica en la posición $_{ij}$ la relación entre los vértices i y j. Normalmente es 1 si hay arista, 0 si no. Si las aristas tienen pesos, se usa el valor del peso. En algunos casos, si los pesos son distancias, la ausencia de aristas se puede indicar con un valor muy grande en vez de 0. Si el grafo no es dirigido la matriz es simétrica. En los algoritmos de abajo se usa una matriz global m (para evitar problemas de pasos de parámetros de estructuras bidimensionales), de $N \times N$, donde $N \geq V$. $m[i][j]$ representa la arista *desde* i *hacia* j. El algoritmo recibe el valor de V. Es decir:

```
#define N 100
```

```
weight m[N][N];

algoritmo (int v, ...) {...m[v1][v2]...
```

Otra representación usada por estos algoritmos es la lista de aristas, simplemente una `list<edge>`. En este caso usualmente se asume que no importan los vertices que no aparecen en ningún extremo de arista.

Una representación un poco más estructurada que la anterior es la lista de adyacencia. Es un array de listas de aristas indizado por vértice: i.e. `list<edge> l[V]`, donde $l[v]$ es la lista de aristas que se originan en el vértice v. En este caso debe almacenarse la arista de forma tal que la componente a de cada arista en $l[v]$ sea v.

9.2. Conectitud

Problema: dado un grafo, construir una estructura eficiente para poder consultar si hay caminos entre dos vertices.

Entrada: Una lista de aristas, para vértices en $\{0..N-1\}$

Salida: Una estructura donde para cada vertice v se puede realizar una operación $find(v)$ que devuelve un identificador único de la componente conexa.

Notas: La estructura debe inicializarse con `INIT(tacho,N,-1)`. Se agregan aristas con el procedimiento join.

```
#include "../template.h"
const int N = 100;
int tacho[N]; /* Estructura de datos para union-find */

int find(int a)
{
  int j = a, k = a, l;
  while (tacho[j] >= 0)
    j = tacho[j];
  while (k != j) {
    l = tacho[k];
    tacho[k] = j;
    k = l;
  }
  return j;
}
```

```
void join(int a, int b)
{
  int j = find(a);
  int k = find(b);
  tacho[min(j, k)] = tacho[j] + tacho[k];
  tacho[max(j, k)] = min(j, k);
}
```

9.3. Minimum spanning tree

Un spanning tree de un grafo conexo es un subgrafo con todos los vértices, conexo, sin ciclos.

Estos son dos algoritmos para resolver el problema de encontrar el spanning tree de minimo peso total (asumiendo pesos en las aristas del grafo). Tienen comportamiento ligeramente distinto cuando el grafo de entrada no es conexo. Prim además funciona en grafos dirigidos (dando un árbol dirigido)

9.3.1. Kruskal

Problema: encontrar un MST de un grafo.

Entrada: Una lista de aristas con el grafo.

Salida: Una lista de aristas con el MST.

Notas: Si el grafo de entrada no es conexo, la salida contiene las aristas de los MSTs de cada componente conexa de la entrada.

```
#include <list>
#include <queue>

#include "edge.cc"
#include "union-find.cc"

list<edge> kruskal(list<edge> l)
{
  priority_queue<edge> p;
  int i, j;
  for (i = 0; i < N; i++)
    tacho[i] = -1;
  for(ei ii = l.begin(); ii != l.end(); ii++)
    p.push(*ii);
```

```
  list<edge> res;
  while (!p.empty()) {
    edge e = p.top();
    p.pop();
    if ((i = find(e.a)) != (j = find(e.b))) {
      join(i, j);
      res.push_back(e);
    }
  }
  return res;
}
```

9.3.2. Prim

Problema: encontrar un MST de un grafo, puede ser dirigido.

Entrada: El grafo como lista de adyacencias. v la cantidad de vertices (que son $\{0..v-1\}$), p_0 un vértice de donde empezar

Salida: Una lista de aristas con el MST

Notas: Si el grafo de entrada no es conexo, la salida contiene el MSTs de la componente conexa de p_0. Si el grafo no es dirigido, para cada arista (a, b), deben ponerse las aristas de forma tal que ($l[a]$ has (a, b, w)) y ($l[b]$ has (b, a, w)) (notar el orden de los vertices). En general, el primer vertice de las aristas en $l[a]$ debe ser a.

```
#include <algorithm>
#include <list>
#include <queue>

list<edge> prim(list<edge> l[], int v, int p0)
{
  bit_vector added(v, false);
  list<edge> res;
  priority_queue<edge> can;

  added[p0] = true;
  for(ei ii=l[p0].begin(); ii != l[p0].end(); ii++)
    can.push(*ii);
  while (!can.empty()) {
    edge e = can.top();
    can.pop();
```

```
    if (!added[e.b]) {
      res.push_back(e);
      added[e.b] = true;
      for (ei ii = l[e.b].begin(); ii != l[e.b].end(); ii++)
        can.push(*ii);
    }
  }
  return res;
}
```

9.4. Traversing

9.4.1. Breadth First (BFS)

9.4.2. Depth First (DFS)

9.5. Detección de ciclos

Notar que en un grafo dirigido, para detectar la presencia de ciclos puede usarse el algoritmo de sorting topológico (pasandole la lista de aristas)

En un grafo no dirigido, puede usarse bfs/dfs en cada componente conexa, y revisando no encontrar up-edges.

9.6. Caminos más cortos

9.6.1. Dijkstra

9.6.2. Floyd

Problema: Dado un grafo dirigido, encontrar las longitudes de los caminos más cortos entre todo par de vertices.

Entrada: m, una matriz de adyacencia con las distancias (positivas), o 0 si no hay arista. v indica la cantidad de vertices en el grafo.

Salida: La matriz es modificada. $m[i, j]$ indica la longitud de camino menor entre los vértices i y j

Orden: $O(v^3)$

Notas:Las partes comentadas deben ser descomentadas para que el algoritmo tome un 0 como "no hay arista". Si se deja tal como está se puede usar un valor grande "infinito" para marcar ausencia de aristas (por ej INT_MAX/2).

```
const int N=100;
int m[N][N];

/* Floyd */
void floyd(int v)
{
  for (int y = 0; y < v; y++)
    for (int x = 0; x < v; x++)
      /*if (m[x][y]>0) */for (int j = 0; j < v; j++)
        /*if (m[y][j]>0)*/
          if (/*m[x][j]==0 ||*/ m[x][y] + m[y][j] < m[x][j])
            m[x][j] = m[x][y] + m[y][j];
}
```

9.7. Caminos más largos

El problema del camino más largo (sin ciclos) en un grafo es NP, la única opción es búsqueda exhaustiva (con BFS/DFS)

9.8. Camino más corto por todos los vertices (Traveling salesman)

Problema: Dado un grafo completo con longitudes, encontrar el camino más corto que visita todos los vértices exactamente una vez.

Este problema es NP-completo, hay soluciones ligeramente mejores a revisar todos los caminos: Ser greedy para explorar, y dejar de explorar si ya se siguió un camino muy largo; si el grafo representa distancia euclideana en un plano, la trayectoria más corta no se puede cruzar a si misma.

9.9. Ciclo simple por todos los vértices (camino Hamiltoniano)

Problema: Dado un grafo, encontrar un ciclo que pase por cada vértice exactamente una vez

Este problema es NP-completo.

10. Networks

Un network representa un sistema de cañerías. Es un grafo dirigido con pesos positivos en las aristas. Los pesos son las capacidades.

10.1. Flujo máximo/corte mínimo

Un flujo del vértice s al vertice t es una asignación de valores a cada arista tales que:

1. Asigna valores no negativos
2. La asignación a cada arista no es mayor a la capacidad de la arista
3. En cada vertice excepto en s y t, el flujo entrante iguala al saliente
4. En s el flujo saliente es mayor o igual, y en t es menor o igual.

Problema: Dado una network, s y t, encontrar un flujo que maximice el flujo saliente en s. Encontrar el mínimo corte, tal que se maximice la capacidad saliente del corte.

Entrada: Un network representado con lista *edges* de aristas, donde la arista consiste en dos vertices y una capacidad. Los índices de dos vértiices s y t.

Salida: $result[i][j]$, una matriz indicando el flujo resultante de i a j

Orden: $O(VE^2)$?

Nota: (Teorema) La capacidad saliente del mínimo corte, es igual al flujo saliente de s en el máximo flujo.

```
// Max-flow, for a network represented as a list of edges

#include <list>
#define MAXINT 10000

using namespace std;

#define NODECOUNT 8
#define UNLABELED (-2)
#define NULLLABEL (-1)

typedef int capacity;
typedef struct {
  int from, to;
  capacity cap;
} edge ;

int main (void) {
  // INPUT: network capacities, source and sink vertices
  int source, sink;

  list <edge> edges ;

  // OUTPUT: flow
  capacity result[NODECOUNT][NODECOUNT] ; // result[i][j] = resulting flow from
i to j

  // MAX-FLOW / MIN_CUT ALGORITHM
  int label[NODECOUNT] ;
  int direction[NODECOUNT] ;
  list <int> unscanned ;

  int i, j;

  // Init result to 0
  for (i=0; i<NODECOUNT; i++)
    for (j=0; j<NODECOUNT; j++)
      result[i][j] = 0;

  do {
    // Try to find augmenting chain
    // Unlabel all, set source label to --
    for (i=0; i<NODECOUNT; i++) label[i] = UNLABELED ;
    label[source] = NULLLABEL ;
    unscanned.push_back (source) ;

    list <edge> F ;
    list <edge> B ;
    edge e ;

    list <edge>::iterator iter = network.begin() ;

    for (;iter != network.end(); iter++) {
      int s = iter -> from ;
      int d = iter -> to ;
      int c = iter -> cap ;
      if (c-result[s][d] > 0) {
        F.insert (iter*)
    bool F[NODECOUNT][NODECOUNT] ; // F[i][j] == forward arc i -> j, with slack[
i][j] > 0
    bool B[NODECOUNT][NODECOUNT] ; // B[i][j] == backward arc i -> j, with resul
t[i][j] > 0
    for (i=0; i<NODECOUNT; i++)
      for (j=0; j<NODECOUNT; j++)
        F[i][j] = (flow[i][j] - result[i][j] > 0) ;

    for (i=0; i<NODECOUNT; i++)
      for (j=0; j<NODECOUNT; j++)
        B[i][j] = (result[i][j] > 0) ;

    // Labeling and scanning
    for (i=0; i<NODECOUNT; i++) scanned[i] = false;

    while (label[sink]==UNLABELED) {
      if unscanned.empty() break ; // flow cannot be augmented
      // find labeled unscanned node
      i = unscanned[1] ;

      // scan node i
      for (j=0; j<NODECOUNT; j++) {
        if (label[j]==UNLABELED && F[i][j]) {
          label[j] = i ;
          direction[j] = +1 ;
          unscanned.push_back (j) ;
        }
        if (label[j]==UNLABELED && B[j][i]) {
          label[j] = i ;
          direction[j] = -1 ;
          unscanned.push_back (j) ;
        }
      }
      unscanned.pop_front() ;
    }

    if (label[sink]!=UNLABELED) { // Found an augmenting chain. Augment

      int lambda = MAXINT;
      int pos ;

      // set lambda to amount to augment
      for (pos = sink; pos != source; pos=label[pos]) {
        if (direction[pos]==+1)
          lambda = min (lambda, flow[label[pos]][pos]-result[label[pos]][pos]) ;
        else // direction[pos]== -1
          lambda = min (lambda, result[pos][label[pos]]) ;
      }
      // augment
      for (pos = sink; pos != source; pos=label[pos]) {
        if (direction[pos]==+1)
          result[label[pos]][pos] += lambda ;
        else // direction[pos]== -1
          result[pos][label[pos]] -= lambda ;
      }
    }

  } while (label[sink]!=UNLABELED) ;

  // result [i][j] has the maximum flow
}
```

11. Matemática

11.1. Matrices

Las operaciones usuales de matrices pueden realizarse con los siguientes macros:

```
// Funciones de matrices (copia, suma, producto)
#define CP2(A,B,n,m) do {FOR (i_,(n)) FOR (j_,(m)) \
                (B)[i_][j_]=(A)[i_][j_];} while(0)
#define SUM2(A,B,C,n,m) do {FOR (i_,(n)) FOR (j_,(m)) \
                (C)[i_][j_]=(B)[i_][j_]+(A)[i_][j_];} while(0)
#define PROD2(A,B,C,n,m,p) \
                do {FOR (i_,(n)) FOR (j_,(p)) {\
                  INT sum_=0; FOR (k_,(m)) \
                    sum_+= (A)[i_][k_]*(B)[k_][j_]; \
                  (C)[i_][j_] = sum_;}} while(0)
```

`CP2` realiza matricialmente $B \leftarrow A$, donde A y B son matrices de $n \times m$.

`SUM2` realiza matricialmente $C \leftarrow A + B$, donde A, B y C son matrices de $n \times m$.

`PROD2` realiza matricialmente $C \leftarrow AB$, donde A es de $n \times m$, B de $m \times p$ y C de $n \times p$.

11.2. Números grandes

Esta subsección contiene un módulo de números grandes con la mayoría de las funciones usuales. Puede elegirse un subconjunto de operaciones necesarias para implementarse.

```
#include <string>
#include <list>
#include <math.h>
#include <utility>
#include <algorithm>
#include "../template.h"

using namespace std;

typedef list<short> number;
typedef number::iterator ni;
typedef number::reverse_iterator rni;

// Normalizacion (remueve los ceros al inicio)
number strip(number &a) {
  while (a.size() && (a.front() == 0))
    a.pop_front();
  return a;
}

// comodo para escribir más abajo
#define item0(ii,n) ((ii)!=(n).rend()?*(ii)++:0)
/*
 * Input / Output
 */

istream& operator>>(istream &c, number &n) {
  string s;
  n.clear();
  c >> s;
  for(string::iterator ii = s.begin(); ii != s.end(); ii++)
    n.push_back(*ii - '0');
  strip(n);
  return c;
}

ostream& operator<<(ostream &c, number n) {
  if (!n.size())
    c << "0";
  else
    for(ni ii = n.begin(); ii != n.end(); ii++)
      c << *ii;
  return c;
}

/*
 * Comparacion
 */

bool operator<(number a, number b) {
  if (a.size() == b.size())
    return lexicographical_compare (a.begin(), a.end(), b.begin
(), b.end()) ;
  return a.size()<b.size();
}

bool operator>(number a, number b) {
  return b<a;
}


// Convertir a long double
long double doub(number a) {
  ni ii;
  long double b = 0;
  for(ii = a.begin(); ii != a.end(); ii++)
    b = 10*b + *ii;
  return b;
}

// Convertir de ulint
number num(unsigned long int a) {
  number n;
  while(a) {
    n.push_front(a % 10);
    a /= 10;
  }
  return n;
}

// Suma
number operator+(number a, number b) {
  rni ii, jj;
  number c;
  short car = 0;
  for(ii = a.rbegin(), jj = b.rbegin(); car || (ii != a.rend())
 || (jj != b.rend());) {
    int d = item0(ii,a) + item0(jj,b) + car;
    c.push_front(d % 10);
    car = d / 10;
  }
  return c;
}

// Resta (devuelve 0 si a < b)
number operator-(number a, number b) {
  rni ii, jj;
  number c;
  short bor = 0;
  if (a < b) return c;
  for (ii = a.rbegin(), jj = b.rbegin(); ii != a.rend();ii++) {
    int d = *ii - item0(jj,b) - bor ;
    c.push_front(mod(d, 10));
    bor = d<0?1:0;
  }
  return strip(c);
}
```

```
// Multiplicacion por constante
number operator*(number a, unsigned int b) {
  rni ii;
  number c;
  int car = 0;
  if (!b) return c; // Manejar caso b==0, sino hay que normaliz
ar.
  for(ii = a.rbegin(); (ii != a.rend()) || car;) {
    int d = item0(ii,a) * b + car;
    c.push_front(d % 10);
    car = d / 10;
  }
  return c;
}

// Multiplicacion
number operator*(number a, number b) {
  rni ii;
  number c;
  for (ii = a.rbegin(); ii != a.rend(); ii++) {
    c = c + b * *ii;
    b = b * 10;
  }
  return c;
}

// Potencia
number operator^(number a, int b) {
  number c=num(1);
  for(int i = 0; i < b; i++)
    c = c * a;
  return c;
}


// Decremento (prefix operator)
number operator--(number &a) {
  rni ii = a.rbegin();
  (*ii)--;
  while((ii != a.rend()) && (*ii < 0)) {
    *ii += 10;
    ii++;
    (*ii)--;
  }
  if (ii == a.rend())
    a.clear();
  return strip(a);
}

// Incremento (prefix operator)
number operator++(number &a) {
  rni ii = a.rbegin();
  (*ii)++;
  while((ii != a.rend()) && (*ii >= 10)) {
    *ii -= 10;
    ii++;
    (*ii)++;
  }
  if (ii == a.rend())
    a.push_front(1);
  return a;
}

pair<number, number> operator/(number a, number b) {
  number q;
  number d = b, i = num(1);
  while (a >= b) {
    if (a>=d) {
      a = a - d; q = q+i;
      d = d*2; i = i*2;
    } else {
      d = b; i = num(1);
    }
  }
  return make_pair(q, a);
}

// Raiz enesima
number nrt(number a, unsigned int b) {
  number n = num(10) ^ ((a.size() - 1) / b);
  number e;
  while((e = n ^ b) != a) ++n;
  return n;
}
```

12. Otros

12.1. Equivalence problem

Problema: Dada un subconjunto describiendo una relación de equivalencia, (como lista de pares), decidir si un par de objetos es equivalente.

Este problema es en realidad el mismo tratado en la sección de grafos, componentes conexas. Se mapean los objetos a vertices del grafo, y los pares a aristas. Después, dos objetos son equivalentes sii sus vertices asociados están en la misma componente conexa.