Solución Primer Parcial de Programación 3 (30/09/2011)

Instituto de Computación, Facultad de Ingeniería

Ejercicio 1

Parte 1) *Se ha realizado una versión detallada de la solución, no era requerido este nivel de detalle en la resolución del parcial en el ejercicio 1.1. Se han indicado en negrita los elementos más importantes para el análisis.*

- Notar: Cualquier invocación a *swap* hace tres asignaciones, o sea su costo es 3.

Si se observa las líneas 4 y 5:

```
4           for(i=0; i < n-1; i = i + 1)
5                 swap(array[i], array[n-1-i]);
```

En principio puede parecer que se invierten los elementos del arreglo, pero en realidad, al continuarse haciendo swap después de la mitad, los elementos vuelven a quedar con la distribución inicial, salvo el primer y último elementos que quedan cambiados.
Por lo que dado un arreglo $(a_0, a_1, a_2, ..., a_{n-1})$ luego de la permutación el arreglo queda $(a_{n-1}, a_1, a_2, ..., a_{n-3}, a_{n-2}, a_0)$. Esto se hace **para todos los casos posibles de entrada**.
Obs.: en la corrección se aceptaron como buenas todas la posibilidades mencionadas: que el arreglo quedara como se indica, quedara invertido o quedara como estaba inicialmente.

Costo de líneas 4 y 5:

- Costo 1 por inicialización "*i=0*"
- *n-1* veces
  - "`i < n-1`". Costo 1
  - "`swap(array[i], array[n-1-i])`". Costo 3
  - "`i = i + 1`". Costo 2.
- Última comparación del for "`i < n-1`". Costo 1.

Por lo tanto,

$$T^{(1)}(n) = 1 + \sum_{i=0}^{n-2} (1+3+2) + 1 = 1 + \sum_{i=0}^{n-2} 5 + 1 = 2 + 6(n-1) = 6n - 4$$

Ahora se van a estudiar las líneas 6,7,8,9 y 10.

```
6           for(i=0; i < n-1; i = i + 1)
7                 for(j=0; j < n-(i+1); j = j + 1)
8                       if(array[j] > array[j+1])
9                             swap(array[j], array[j+1]);
10    }
```

Las líneas 6,7 y 8 se van a ejecutar siempre una cantidad constante de veces independientemente del contenido de array.
El mejor caso estará ***dado por las entradas*** que hagan que la expresión "`array[j] > array[j+1]`" siempre sea falsa, **siempre y cuando dichas entradas existan**. Es fácil encontrar una entrada así: basta que el arreglo permutado luego de las líneas 4 y 5 esté estrictamente ordenado en forma ascendente. O sea, $a_{n-1} < a_1 < a_2 < ... < a_{n-3} < a_{n-2} < a_0$.
Otro caso es que los elementos sean todos iguales.

Por otro lado, **es posible encontrar casos** de arrays de entrada que evalúen la expresión "`array[j] > array[j+1]`" **siempre en verdadero. Como para estas entradas el algoritmo siempre realiza el swap de la línea 9, éstos serán los peores casos**.
Si $a_{n-1} > a_1 > a_2 > ... > a_{n-3} > a_{n-2} > a_0$ se tiene un peor caso

Luego del análisis anterior estamos en condiciones de calcular tanto el mejor y peor caso de las líneas 6,7,8,9 y 10.

Peor caso de parte 2:

- Costo 1 por inicialización "*i=0*" (línea 6)
- La variable *i* desde *0* hasta *n-2*
  - Costo 1 por "`i < n-1`" línea 6.
  - Costo 2 por "`i = i + 1`" línea 6.
  - Costo 1 por inicialización "`j=0`" línea 7.
  - n-i-1 veces
    - Costo 2 por "`j < n-(i+1)`" línea 7
    - Costo 2 por "`j = j + 1`" línea 7
    - Costo 2 por "`(array[j] > array[j+1])`"
    - Costo 3 + 1 por "`swap(array[j], array[j+1])`"
  - Costo 2 por última comparación "`j < n-(i+1)`" línea 7
- Costo 1 por última comparación "`i < n-1`" línea 6.

$$T_W^{(2)}(n) = 1 + \sum_{i=0}^{n-2}(1+2+1+(n-i-1)(2+2+2+4)+2)+1 = 2+10n^2-10n-10\frac{(n-2)(n-1)}{2}-4n+4$$

Mejor caso de parte 2:

- Costo 1 por inicialización "*i=0*" (línea 6)
- La variable *i* desde *0* hasta *n-2*
  - Costo 1 por "`i < n-1`" línea 6.
  - Costo 2 por "`i = i + 1`" línea 6.
  - Costo 1 por inicialización "`j=0`" línea 7.
  - n-i-2 veces
    - Costo 2 por "`j < n-(i+1)`" línea 7
    - Costo 2 por "`j = j + 1`" línea 7
    - Costo 2 por "`(array[j] > array[j+1])`"
  - Costo 2 por última comparación "`j < n-(i+1)`" línea 7
- Costo 1 por última comparación "`i < n-1`" línea 6.

$$T_B^{(2)}(n) = 1 + \sum_{i=0}^{n-2}(1+2+1+(n-i-1)(2+2+2)+2)+1 = 2+\sum_{i=0}^{n-2}(6+6(n-i-1))$$

$$\boxed{T_B(n) = T^{(1)}(n) + T_B^{(2)}(n) = 3n^2 + 9n - 8}$$

$$\boxed{T_W(n) = T^{(1)}(n) + T_W^{(2)}(n) = 5n^2 + 7n - 8}$$

Parte 2)
Propiedades que deben enunciarse:

Para toda función $f : N \to R^*, g : N \to R^*$:

Regla del límite:

1. Si $\lim_{n \to +\infty} \frac{f(n)}{g(n)} \in R^+$ entonces $f \in O(g)$ y $g \in O(f)$
2. Si $\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$ entonces $f \in O(g)$ pero $g \notin O(f)$
3. Si $\lim_{n \to +\infty} \frac{f(n)}{g(n)} \to +\infty$ entonces $f \notin O(g)$ pero $g \in O(f)$

Propiedad de dualidad:
$f \in O(g) \Leftrightarrow g \in \Omega(f)$.

Definición: Orden exacto.
$\Theta(f) = O(f) \cap \Omega(f)$

a)

$\lim_{n \to +\infty} \frac{n!!}{n!} = \lim_{n \to +\infty} \frac{n!(n!-1)!}{n!} = \lim_{n \to +\infty} (n!-1)! = +\infty$.
(Regla del límite 3) entonces
$(n!)! \notin O(n!)$
(Dualidad) sii
$n! \notin \Omega((n!)!)$
(definición de Θ) entonces
$n! \notin \Theta((n!)!)$.

b)

$\lim_{n \to +\infty} \frac{(n!)!}{n!} = \lim_{n \to +\infty} \frac{n!(n!-1)!}{n!} = \lim_{n \to +\infty} (n-1)! = +\infty$

(Regla del límite 3) entonces
$n! \in O((n!)!)$

c)

$$\lim_{n\to+\infty}\frac{100n^5+23n^4}{3n^5+(n+1)^2}=\lim_{n\to+\infty}\frac{100n^5}{3n^5+(n+1)}+\lim_{n\to+\infty}\frac{23n^4}{3n^5+(n+1)^2}=100/3+0=100/3.$$

(Regla del límite 1) entonces
$100n^5+23n^4\in O(3n^5+(n+1)^2)$ y $3n^5+(n+1)^2\in O(100n^5+23n^4)$.

(Dualidad) entonces
$3n^5+(n+1)^2\in\Omega(100n^5+23n^4)$ y $100n^5+23n^4\in\Omega(3n^5+(n+1)^2)$

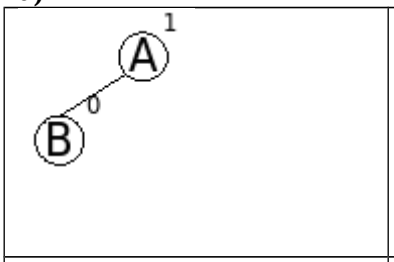
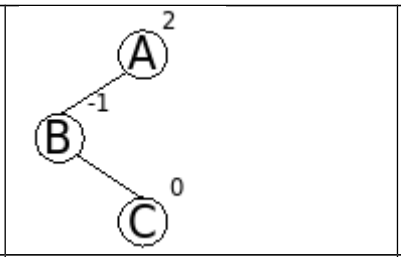
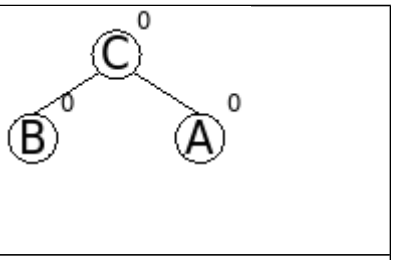
Por (definición de Θ)
$100n^5+23n^4\in\Theta(3n^5+(n+1)^2)$

Parte 3)

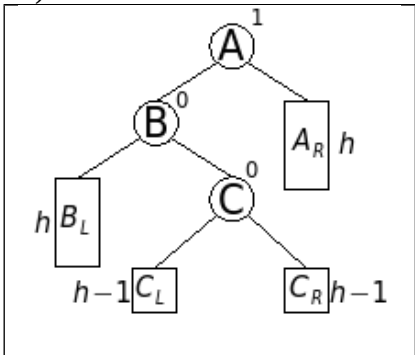
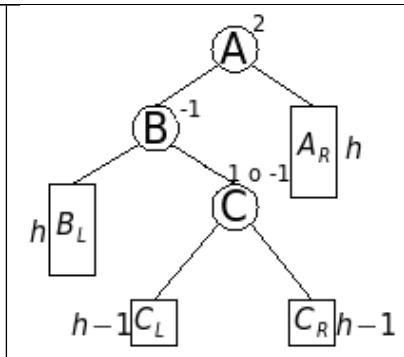
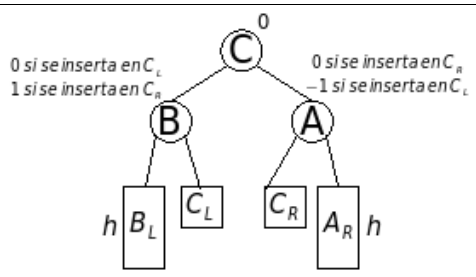
1. Tipo LR

Se presenta esta rotación en dos casos, aunque en realidad se trata del mismo, a los efectos de ver primeramente el caso sencillo y después el caso completo.

a)

		
Previo a la inserción	**Luego de la inserción**	**Rebalanceo**

b)

		
Previo a la inserción	**Luego de la inserción**	**Rebalanceo**
	En el nodo puede ser insertado en C_L o C_R. Si va a C_L el FB de C pasa a ser 1, sino va a C_R el FB de C pasa a ser –1. En cualquiera de los 2 casos, el FB de B pasa a ser -1 y el de A pasa a ser 2.	

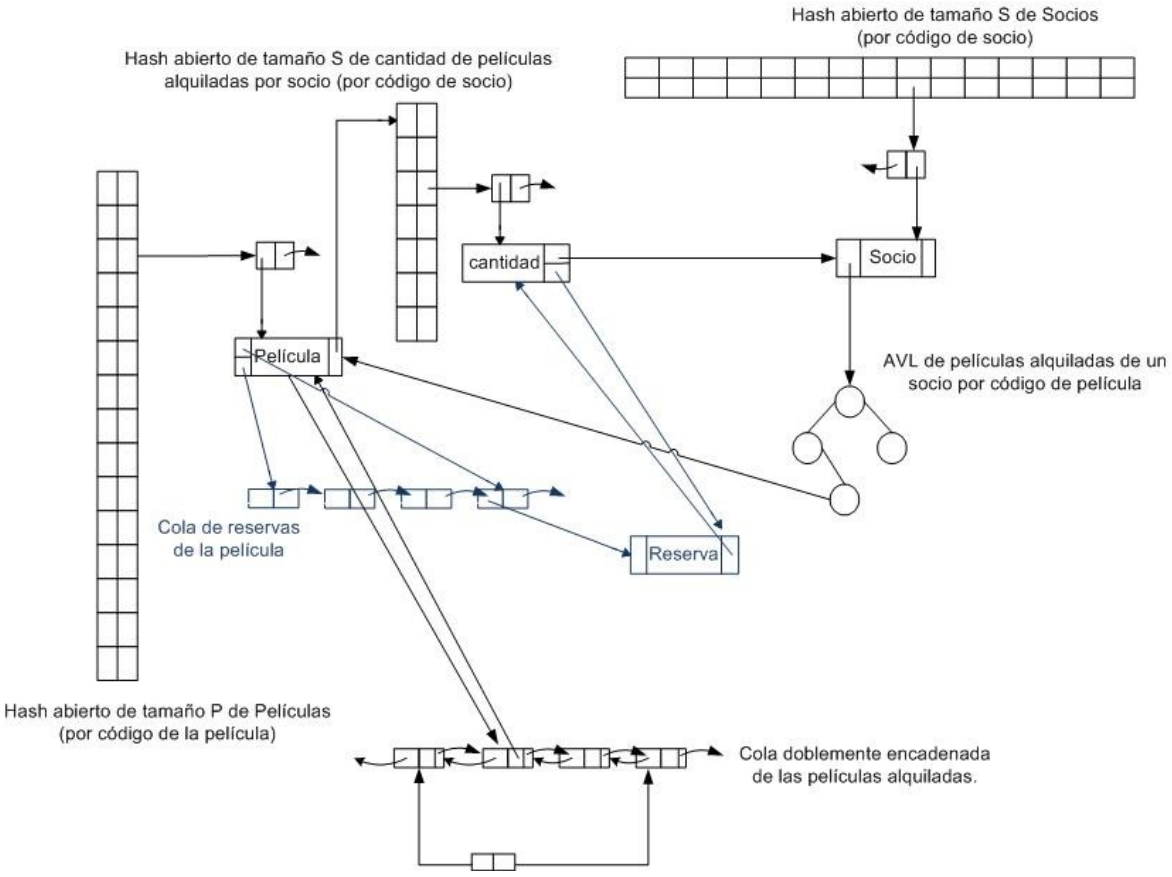
Ejercicio 2

Nota: el contenido de esta solución incluye más detalle del que se exige durante la resolución del parcial. Se incluye dicho contenido para que la solución pueda utilizarse como material de estudio en otras ediciones del curso.

Parte a)

La estructura *VideoClub* propuesta tiene 3 punteros: uno que apunta al hash de Socios, otro que apunta al hash de Películas y otro que apunta a la cola doblemente encadenada de las películas que se encuentran alquiladas en un momento dado.

La estructura propuesta se presenta en la siguiente figura:



Nota: en el diagrama en color azul se incluye la solución de la parte b por más que no era requerido en la resolución del parcial.

1. ***void imprimirPeliculasAlquiladas(VideoClub V)***

   Caso promedio:

   - para resolver esta operación se debe recorrer la Cola de películas alquiladas. Las películas alquiladas se van insertando al final de la Cola a medida que se van alquilando, por lo tanto, se debe recorrer los elementos del primero al último para que estén impresos en orden cronológico según su inserción.
   - acceder a la información de cada Película (título y género) es orden constante ya que se accede desde el puntero del nodo de la Cola a la Película.

En caso de que la Cola de películas alquiladas este vacía al momento de invocar la operación, se imprime el siguiente mensaje: “No hay películas alquiladas en este momento.”, orden constante.

Por lo tanto, el orden de la operación en el caso promedio es *O(P)*.

Nota: se aceptó tanto impresión en forma ascendente como descendente.

2. ***void imprimirPeliculasAlquiladasDeUnSocio(String codigoSocio, VideoClub V)***

Caso promedio:

- encontrar al Socio con código *codigoSocio* en el caso promedio es *O(1)* (a través del hash de Socios).
- dado que un socio tiene como máximo *A* películas alquiladas en un momento dado, recorrer el AVL de películas alquiladas es orden *A* en el caso promedio, ya que se visitan todas las películas alquiladas del socio. La recorrida se debe realizar en orden ya que la inserción es por el código de película en orden.
- luego, acceder al director de la película para imprimirlo junto al código de la película es orden constante, ya que se accede a la información de cada Película a través de la referencia que se almacena en cada nodo del AVL hacia la Película.

Por lo tanto, el orden de la operación en el caso promedio es *O(A)*.

3. **void devolverPeliculaAlquilada(String codigoPelicula, String codigoSocio, VideoClub &V)**

Caso promedio:

- encontrar al Socio con código *codigoSocio* en el caso promedio es *O(1)* (a través del hash de Socios).
- borrar la película de *codigoPelícula* del AVL de películas alquiladas del socio es *O(logA)* en el caso promedio.
- encontrar a la Película con código *codigoPelícula* en el caso promedio es *O(1)* (a través del hash de Películas).
- encontrar el nodo en la Cola de películas alquiladas correspondiente a la Película es orden constante, ya que se tiene un puntero de la Película al nodo de la Cola de películas alquiladas.
- borrar el nodo que representa la película alquilada de código *codigoPelícula* de la Cola de películas alquiladas es orden constante, ya que la Cola es doblemente encadenada.

Por lo tanto, el orden de la operación en el caso promedio es *O(logA)*.

4. **void imprimirHistorialDeUnaPeliculaParaUnSocio(String codigoPelicula, String codigoSocio, VideoClub V)**

Caso promedio:

- encontrar la película con código *codigoPelícula* en el caso promedio es *O(1)* (a través del hash de Películas).
- luego encontrar al Socio de código *codigoSocio* en el caso promedio es *O(1)* (a través del hash de Socios que tiene cada Película. Para cada socio se tiene una referencia al

Socio y un cardinal (*cantidad*) que representa la cantidad de veces que la película fue alquilada por el socio. En caso que el socio nunca haya alquilado la película se tiene el valor 0 en dicho cardinal.

Por lo tanto, el orden de la operación en el caso promedio es *O(1)*.

Parte b)

b.1) void alquilarPeliculaAUnSocio(String codigoPelicula, String codigoSocio, VideoClub &V)

Para esta operación se agrega una cola encadenada de reservas para cada película.

Peor caso:

Se debe verificar si la película ya se encuentra alquilada al momento de invocar la operación, ya que no hay una precondición que indique si la película se encuentra alquilada ó no. Para esto:

i. encontrar la película es *O(P)* en el peor caso ya que se accede a la película a través del hash de Películas.
ii. consultar si existe referencia a un nodo de la Cola de películas alquiladas es orden constante.

Por lo tanto, si la película se encuentra alquilada, el orden de la operación es *O(P)* en el peor caso. Si no se encuentra alquilada, se debe verificar si la película a alquilar se encuentra reservada. Si se encuentra reservada, se debe verificar que el socio tenga una reserva de la película, y además, es él que tiene la reserva más antigua para dicha película. Para esto:

- encontrar la película es *O(P)* en el peor caso ya que se accede a la película a través del hash de Películas (paso *i* de la verificación anterior).
- consultar si la Cola de películas de Reservas de la película es vacía, es orden constante.
- si la Cola de películas de Reservas de la Película no es vacía, verificar que el primer el socio que tiene la reserva es el socio que quiere realizar el alquiler, es orden constante (a través de la referencia que se tiene de la Reserva al nodo del hash de socios de la película, y de ahí a la referencia al Socio, *(*)*).

Por lo tanto, si la película se encuentra reservada y a su vez, se encuentra reserva por otro socio, el orden de la operación es *O(P)* en el peor caso.

En caso contrario (es decir, que el socio puede alquilar la película) se tiene que:

- insertar el nodo que representa la película de código *codigoPelicula* en la Cola de películas alquiladas es orden constante, ya que se mantiene un puntero al último nodo insertado.
- actualizar la referencia al último elemento de la Cola es orden constante.
- referenciar al nodo insertado en la Cola de películas alquiladas a la Película y viceversa, es *O(P)* (a través del hash de Películas, paso *i* de la primer verificación).
- encontrar al Socio con código *codigoSocio* en el hash de socios de la película en el peor caso es *O(S)* en caso de que no tuviera reserva, si tenía reserva esto es orden constante (ya que se accede al Socio a través *(*)*).
- incrementar en uno el campo cantidad (orden constante).
- encontrar al Socio con código *codigoSocio* en el hash de Socios para actualizar el conjunto de películas que tiene alquiladas es orden constante (a través de la referencia al Socio del nodo del hash de Socios de la película)

- insertar la película de código *codigoPelícula* en el AVL de películas alquiladas del socio es *O(logA)* en el peor caso.
- referenciar del nodo insertado en el AVL a la película recién alquilada es orden constante ya que se mantiene la referencia a la película en el paso *i*.
- si el socio tenía una reserva, eliminar la misma de la Cola de películas de Reservas de la Película es orden constante.

Por lo tanto, el orden de la operación en caso de que no tuviera reserva es *O(P + S + logA)* en el peor caso, y si tuviera reserva el orden es *O(P + logA)* en el peor caso.

Por lo tanto, el orden de esta operación en el peor caso es *O(P + S + logA).*

b.2) void reservarPeliculaAUnSocio(String codigoPelicula, String codigoSocio, String fecha, VideoClub &V)

Para esta parte se agrega una referencia del nodo de hash de Socios de una Película al nodo Reserva en caso de que el Socio tenga una reserva para la película. Esta nueva referencia no modifica los órdenes de las operaciones anteriores.

Caso promedio:
- encontrar la película con código *codigoPelícula* en el caso promedio es *O(1)* (a través del hash de Películas).
- encontrar al Socio con código *codigoSocio* en el hash de Socios que tiene cada Película es *O(1)* en el caso promedio.
- verificar que el socio no tenga una reserva para la película de *codigoPelicula* es *O(1)* (verificando si se tiene ó no una referencia a un nodo Reserva desde el nodo del hash del Socios que tiene la Película).
- referenciar a la Reserva desde el nodo del hash de Socios que tiene la Película es orden constante (ya que guardo la referencia del paso anterior a ese nodo).
- referenciar al nodo del hash de Socios que tiene la Película desde la Reserva es orden constante (ya que guardo la referencia del paso anterior a ese nodo).
- insertar la reserva al final de la cola de reservas de la película es orden constante.

Por lo tanto, el orden de la operación en el caso promedio es *O(1)*.

Algunas soluciones alternativas que se tomaron como válidas:
- Para la parte a) en vez de que cada Película tenga un hash de Socios para mantener la cantidad de veces que una película fue alquilada por un socio, se podía haber incluido un hash de películas para cada socio, en donde se mantenía la cantidad de veces que el socio alquiló cada película.
- Incluir un hash de Películas Reservadas para cada Socio para verificar en O(1) promedio si una película ya esta reservada por un socio para la parte b.2. Este hash modifica el orden de la parte b.1 a *O(2P + S + logA)* en el peor caso porque se debe eliminar esta reserva de ese hash si es que tiene una.
- No tener una referencia al Socio del nodo del hash de Socios de cada Película ó desde el nodo Reserva. Esto hace que el orden de la operación b.1 sea *O(P + 2S + logA)* en el peor caso ya que se tiene que ir a buscar al Socio en el hash de Socios.

Ejercicio 3

Parte 1)

```
ListaNodos BuscarContenido(int origen, char* cadena, int hops) {
      int N = cantNodosGrafo(red);
      int *visitados = new int[N];
      for (int i = 0; i < N; i++)
            visitados[i] = 0;
      visitados[origen] = 1;
      ColaEnteros Q = crearCola();
      encolar(Q, origen);
      ColaEnteros camino = crearCola();
      ColaEnteros nivel = crearCola();
      while (!esVaciaCola(Q) && (hops > 0)) {
            int v = desencolar(Q);
            ListaEnteros adys = obtenerAdyacentesGrafo(red, v);
            while (!esVaciaLista(adys)) {
                  w = primeroLista(adys);
                  adys = restoLista(adys);
                  if (!visitados[w]) {
                        encolar(Q, w);
                        visitados[w] = 1;
                        if (existeContenido(cadena, w))
                              encolar(camino, w);
                        encolar(nivel, hops-1);
                  }
            }
            hops = desencolar(nivel);
      }
      delete [] visitados;
}
```

Parte 2) (7 puntos)

```
void CaminosAlternativos(int origen, int fuente){
      ListaNodos camino = crearLista();
      int N = cantNodosGrafo(red);
      int marcados* = new int[N];
      int cantCaminos = 0;
      for (int i = 0; i < N; i++)
            marcados[i] = 0;
      CaminosAltDFS(origen, fuente, marcados, camino, cantCaminos);
}

void CaminosAltDFS(int inicio, int fin, int *marcados, ListaNodos
camino, int &cantCaminos){
      if (cantCaminos < C ) {
            marcados[inicio] = 1;
            ListaNodos adys = obtenerAdyacentesGrafo(red, inicio);
            encolar(inicio, camino);
            if (inicio == fin) {
                  guardarCamino(camino);
                  cantCaminos++;
            } else {
                  while (!esVacia(adys)) {
                        ady = desencolar(adys);
                        if (marcados[ady] == 0) {
                              CaminosAltDFS(ady, fin, marcados, camino,
                                    cantCaminos);
                        }
                  }
            }
            marcados[inicio]=0;
            sacarUltimo(camino);
      }
}
```