Solución Primer Parcial de Programación 3 (1/10/2009)

Instituto de Computación, Facultad de Ingeniería

Ejercicio 1 (14 puntos)

1. Cabe notar que en la primera iteración del *for* con la variable *i* nunca entra ya que A[0] > A[0] es siempre false y esto se tiene en cuenta para los siguientes razonamientos.

   Otra observación, es que la entrada siempre es de tamaño mayor igual a 1 por lo que *T(0)* no está definido. Notar que el *n* que se pasa por parámetro en la función no es igual al tamaño del arreglo.

   Cuenta los elementos del arreglo entre A[1] y A[n] que son menores iguales que A[0], más los que no se repiten en el resto del arreglo.

   Ya que la operación básica es la asignación de r, las líneas del algoritmo que se consideran son r = 1 y r = 0.

   r = 0 siempre se ejecuta en la inicialización del algoritmo y en algunas oportunidades del *for*.

   Peor caso:

   Uno de los casos en que se da cuando todas las entradas A[0] < A[1] = .. = A[n] son iguales.

   - 1 inicialización inicial de r
   - n+1 iteraciones por el for con variable i
     - 1 por la primer inicialización de r.
   - n iteraciones por el for con variable i que es verdadero A[i] > A[0]
     - Pueden haber n-1 elementos repetidos (for variable j)

   $T_w(n+1) = 1 + (n+1)+n(n-1)=n^2+2$

   Ajustando el parámetro de la función: $T_w(n) =(n-1)^2 +2$

   Mejor caso:

   Una entrada tiene como costo mínimo es cuando para todo i en {0,..,n}o bien A[i] ≤ A[0] o A[i] es único.

   - 1 inicialización inicial de r
   - n+1 iteraciones por el for
     - 1 por la primer inicialización de r

   $T_B(n+1) = n + 2$

   Ajustando el parámetro de la función: $T_B(n) = n + 1$

Caso promedio:

El algoritmo A se puede descomponer en la concatenación de dos algoritmos A1 y A2 independientes.

Uno que es la primer inicialización de r que tiene costo 1 con probabilidad 1.

El algoritmo A2 consiste en el bloque del for.

Para una entrada E cualquiera se cumple que:

$$T(E) = T^{A1}(E) + T^{A2}(E)$$

Ya dijimos que $T_A^{A1}(n) = 1 * 1 = 1$. Ahora resta calcular $T_A^{A2}(n)$.

El costo del algoritmo A2 es la suma de los costos que hay para cada iteración de la variable i. O sea el sub-algoritmo que chequea que A[i] > A[0] y luego si esta condición es verdadera chequea que sea único entre A[0] y A[n].

Si *A[i] > A[0], A[i] = A[j] , i <> j* se tiene que incrementar el costo en 1.

Caso contrario no tiene costo.

Si denotamos como $T^{C(i)}(n+1)$ como el costo de fragmento de código:

```
if (A[i] > A[0]){
    for (j=0; j<n+1; j++)
        if (i!=j)
            if(A[i]==A[j])
                r = 0;
}
```

Si i <> 0 entonces

$$T_A^{C(i)}(n+1) = P(A[i] > A[0])(n-1)P(A[0] = A[1]) * 1 = \frac{1}{2}(n-1)\frac{1}{k+1} = \frac{n-1}{2(k+1)}$$

El factor $(n-1)P(A[0] = A[1]) * 1$ viene del for con la variable j. Hay n-1 casos en que *i <> j* que todos tienen la misma probabilidad y su costo es 1.

$$T_A^{A2} = \sum_{i=0}^{n} 1 + T_A^{C(i)}(n) = n + 1 + \sum_{i=1}^{n} T_A^{C(i)}(n) = n + 1 + nT_A^{C(1)}(n) = n + 1 + n\frac{n-1}{2(k+1)}$$

Por lo que $T_A(n+1) = T_A^{A1}(n+1) + T_A^{A2}(n+1) = 1 + n + 1 + n\frac{n-1}{2(k+1)}$

Ajustando el parámetro de la función:

$$T_A(n) = 1 + n + (n-1)\frac{n-2}{2(k+1)}$$

Simplificamos los órdenes:

$T_w(n) = (n-1)^2 + 2$

$T_B(n) = n + 1$

$$T_A(n) = 1 + n + (n-1)\frac{n-2}{2(k+1)}$$

$T_w(n) \in \Theta(n^2)$, $T_B(n) \in \Theta(n)$, $T_A(n) \in \Theta(n^2)$

2. Sean $T1(n) \in O(T2(n))$ y $T2(n) \in O(f(n))$.

Por definición de orden superior:

*(*1)* $T1(n) \in O(T2(n))$ sii $\left(\exists c_0 \in R^+\right)\left(\exists n_0 \in N\right)\left(\forall n \in N\right)\left(n \geq n_0 \rightarrow T_1(n) \leq c_0 T_2(n)\right)$

*(*2)* $T2(n) \in O(f(n))$ sii $\left(\exists c_1 \in R^+\right)\left(\exists n_1 \in N\right)\left(\forall n \in N\right)\left(n \geq n_1 \rightarrow T_2(n) \leq c_1 f(n)\right)$

a) $T1(n) + T2(n) \in O(T_1(n))$ **Falso**

Contraejemplo:

$T1(n) = n$, $T2(n) = n^2$, $f(n) = n^2$

Se verifica que $T2(n) \in O(f(n))$ con $c_1 = 1$ y $n_1 = 0$.

Se verifica que $T1(n) \in O(T2(n))$ con $c_0 = 1$ y $n_0 = 0$.

No se cumple que $T1(n) + T2(n) \in O(T_1(n))$. Para probarlo se supone por absurdo que sí.

$T1(n) + T2(n) \in O(T_1(n))$

Definición de orden

Sii $\left(\exists c_0 \in R^+\right)\left(\exists n_0 \in N\right)\left(\forall n \in N\right)\left(n \geq n_0 \rightarrow n + n^2 \leq c_0 n\right)$

Despejando

Sii $\left(\exists c_0 \in R^+\right)\left(\exists n_0 \in N\right)\left(\forall n \in N\right)\left(n \geq n_0 \rightarrow n(1 - c_0) + n^2 \leq 0\right)$

Sacando factor común

Sii $\left(\exists c_0 \in R^+\right)\left(\exists n_0 \in N\right)\left(\forall n \in N\right)\left(n \geq n_0 \rightarrow n(1 - c_0 + n) \leq 0\right)$

Absurdo ya que a partir de cierto n_0 los factores n y $n(1 - c_0 + n)$ son mayores que cero.

b) $|T1(n) - T2(n)| \in O(f(n))$ **Verdadero**

Por la desigualdad triangular y luego porque las funciones de costos son positivas

$| T_1(n) - T_2(n) | \leq | T_1(n) | + | T_2(n) | = T_1(n) + T_2(n)$

Sea $n_2 = \max\{n_0, n_1\}$ y $c_2 = \max\{c_0, c_1\}$ entonces de las proposiciones *(*1)* y *(*2)* se concluye que $(\forall n \in N)(n \geq n_2 \rightarrow T_1(n) \leq c_0 c_1 f(n))$

Por lo que $(\forall n \in N)(n \geq n_2 \rightarrow T_1(n) + T_2(n) \leq c_0 c_1 f(n) + c_1 f(n) = (c_0 c_1 + c_1) f(n))$

Como para cualquier n se cumple: $| T_1(n) - T_2(n) | \leq T_1(n) + T_2(n)$ entonces

$(\forall n \in N)(n \geq n_2 \rightarrow | T_1(n) - T_2(n) | \leq (c_0 c_1 + c_1) f(n))$

Sii $|T1(n) - T2(n)| \in O(f(n))$.

c) $T1(n) * T2(n) \in O(f(n))$ **Falso**

Utilizamos $T1(n) = n$, $T2(n) = n^2$, $f(n) = n^2$ que ya verificamos en la parte a. Ahora suponemos por absurdo que se cumple.

$T1(n) * T2(n) \in O(f(n))$

Definición de orden

Sii $(\exists c_0 \in R^+)(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow n^3 \leq c_0 n^2)$

Operando

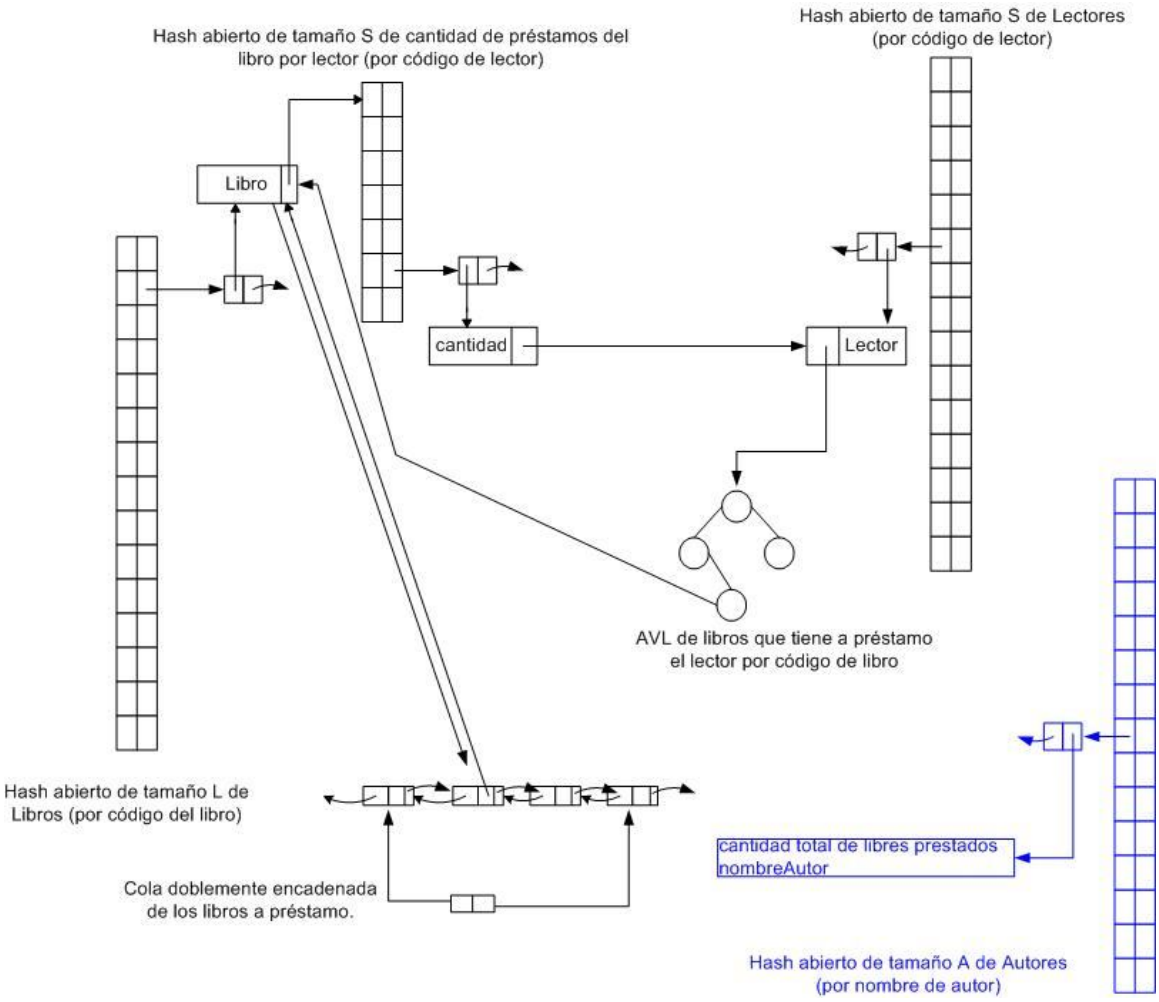
Sii $(\exists c_0 \in R^+)(\exists n_0 \in N)(\forall n \in N)(n \geq n_0 \rightarrow n^2 (n - c_0) \leq 0)$

Lo cual es absurdo ya que a partir del entero después de $\lfloor c_0 \rfloor + 1$, el número $n^2 (c_0 + n)$ es positivo.

Ejercicio 2 (14 puntos)

1. La estructura *Biblioteca* propuesta tiene 3 punteros: uno que apunta al hash de Lectores, otro que apunta al hash de Libros y otro que apunta a la cola doblemente encadenada de los libros que se encuentran a préstamo.

   La estructura propuesta se presenta en la siguiente figura:



   Nota: en color azul se indica la extensión que se debe realizar a la multiestructura para resolver la parte 3).

   void imprimirLibrosPrestados(Biblioteca B)

   Caso promedio:
   - para resolver esta operación se debe recorrer la Cola de libros prestados. Los libros prestados se van insertando al final de la Cola a medida que se van prestando, por lo tanto, se debe recorrer los elementos del primero al último para que estén impresos en ordenes cronológico según su inserción.
   - acceder a la información de cada Libro (título y autor) es orden constante ya que se accede desde el puntero del nodo de la Cola al Libro.

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

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

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

void imprimirLibrosPrestadosDeUnLector(String codigoSocio, Biblioteca B)

Caso promedio:

- encontrar al Lector con código *codigoSocio* en el caso promedio es O(1) (a través del hash de Lectores)
- dado que un lector tiene como máximo *P* libros prestados en un momento dado, recorrer el AVL de los libros prestados es orden *P* en el caso promedio, ya que se visitan todos los libros prestados del lector. La recorrida se debe realizar en orden ya que la inserción es por el código del libro en orden.
- luego, acceder al tema del libro para imprimirlo junto al código del libro es orden constante, ya que se accede a la información de cada Libro a través de la referencia que se almacena en cada nodo del AVL hacia el Libro.

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

void devolverLibro(String codigoLibro, String codigoSocio, Biblioteca &B)

Caso promedio:

- encontrar al Lector con código *codigoSocio* en el caso promedio es O(1) (a través del hash de Lectores)
- borrar el libro de código *codigoLibro* del AVL de libros prestados del lector es *O(logP)* en el caso promedio.
- encontrar el Libro con código *codigoLibro* en el caso promedio es O(1) (a través del hash de Libros)
- encontrar el nodo en la Cola de libros prestados correspondiente al Libro es orden constante, ya que se tiene un puntero del Libro al nodo de la Cola de libros prestados.
- borrar el nodo que representa el libro prestado de código *codigoLibro* de la Cola de libros prestados es orden constante, ya que la Cola es doblemente encadenada.

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

int consultarHistorialDeUnLibro(String codigoLibro, String codigoSocio, Biblioteca B)

Caso promedio:

- encontrar el Libro con código *codigoLibro* en el caso promedio es *O(1)* (a través del hash de Libro)
- luego encontrar el Lector de código *codigoSocio* en el caso promedio es *O(1)* (a través del hash de Lectores que tiene cada Libro. Para cada lector se tiene una referencia al lector y un cardinal (*cantidad*) que representa la cantidad de veces que el libro fue prestado al lector.
- Si el lector no pertenece al hash

se imprime 0
Sino se accede a la información (*cantidad*) del lector, orden constante.

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

2. ***void prestarLibroAUnSocio(String codigoLibro, String codigoSocio, Biblioteca &B)***

Caso promedio:
Se debe verificar si el libro ya se encuentra a préstamo al momento de invocar la operación, ya que no hay una precondición que indique si el libro se encuentra a préstamo o no. Para esto,

- encontrar el libro es *O(1)* en el caso promedio ya que se accede al libro a través del hash de lectores
- consultar si existe referencia a un nodo de la Cola de libros prestados es orden constante.

Si el libro ya se encuentra a préstamo la operación no realiza ninguna acción, sino:

- insertar el nodo que representa el libro prestado de código *codigoLibro* de la Cola de libros prestados es orden constante, ya que la Cola mantiene un puntero al último nodo insertado.
- actualizar la referencia al último elemento de la Cola es orden constante.
- referenciar del nodo insertado en la Cola de libros prestados al Libro, es O(1) (a través del hash de Libros)
- encontrar al Lector con código *codigoSocio* en el caso promedio es O(1) (a través del hash de Lectores del Libro)
- Si el lector ya había pedido antes el libro prestado:
    - incrementar en uno el campo *cantidad* es orden constante.

  sino:
    - crear el nodo que representa la cantidad de veces que el lector pidió prestado es orden constante
    - inicializar en uno *cantidad* es orden constante
    - referenciar al Lector es O(1) en el caso promedio (a través del hash de Lectores)
- encontrar al Lector con código *codigoSocio* para actualizar el conjunto de libros que tiene prestado en el caso promedio es O(1) (a través del hash de Lectores)
- insertar el libro de código *codigoLibro* del AVL de libros prestados del lector es *O(logP)* en el caso promedio.
- referenciar del nodo insertado en el AVL al libro recién prestado.

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

3. ***void imprimirInformacionAutor(String nombreAutor, Biblioteca B)***

Una solución posible para resolver eficientemente esta operación es agregar a la multiestructura un hash de Autores, donde para cada autor se registra su nombre y un entero que representa la cantidad total de veces que libros de su autoría han sido prestados.

La única operación de las partes anteriores que se ve afectada es la operación ***prestarLibroAUnSocio*** ya que se debe actualizar también la cantidad total de libros prestados del autor al momento de prestar el libro. La operación ***prestarLibroAUnSocio*** sigue respetando

el orden requerido *O(logP)* en el caso promedio, ya que actualizar la información anterior requiere las siguientes acciones:

- encontrar al Autor con código *codigoSocio* en el caso promedio es O(1) (a través del hash de Autores)
- Si ya se había prestado al menos un libro del autor, entonces:
    - incrementar en uno el campo *cantidad* es orden constante.

  sino:
    - crear el nodo que representa la cantidad total de veces que se pidieron a préstamo libros del autor es orden constante
    - inicializar en uno *cantidad* y el nombre del autor es orden constante.

Peor caso:

- encontrar al Autor con nombre *nombreAutor* en el peor caso es O(A) (a través del hash de Autores)
- Si no se encuentra al autor, se devuelve 0 ya que nunca se prestó un libro de su autoría, orden constante
- Si se encuentra el autor, entonces se devuelve el valor de la variable cantidad total, orden constante

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

Ejercicio 3 (12 puntos)

1. Un algoritmo basado en DFS para encontrar un ordenamiento topológico de un DAG consta de realizar una recorrida DFS numerando los nodos en pos-orden, es decir numerando cada nodo antes de retroceder en la búsqueda luego de haber explorado a todos sus adyacentes. De este modo, las relaciones de precedencia quedan representadas por el orden descendiente de dicha numeración. Es decir, que para un grafo de *n* nodos el orden topológico queda determinado por los nodos que hayan sido identificados en el siguiente orden: n, n-1,...,1, 0.

   Además, dado que un DAG no es fuertemente conexo es necesario realizar las llamadas correspondientes de modo de cubrir todos los vértices.

```
#define N …

typedef int Grafo[N][N];

void recorr_dfs_posorden(Grafo g, int nodo, int*marcados, int*
ordentopo, int &pos) {
      marcados[nodo] = 1;
      for(int i =0; i <N;i++) {
            if (g[nodo][i]==1){   // Para cada nodo adyacente
                  if (marcados[i] == 0) {
                        recorr_dfs_posorden(g, i, marcados,
                        ordentopo, pos);
                  }
            }
      }
      ordentopo[pos]=nodo;  // Se agrega el nodo al recorrido.
      pos--;
}

int* ordenTopologico(Grafo g){
      int marcados[N]={0};
      int* ordentopo = new int[N];
      int pos = N-1;

      for (int k=0;k<N;k++){
                  if (marcados[k]==0) //Recorrida general necesaria
                  porque g no es necesariamente fuertemente conexo
                  recorr_dfs_posorden(g, k, marcados, ordentopo,
pos);
      }
      return ordentopo;
   }
```

2. El orden topológico ejecutando 1) queda determinado de la siguiente manera: 2,1,4,5,0,3

3. No necesariamente, la unicidad de orden topológico dependerá no solo de la topología particular del grafo donde podrán existir distintos ordenamientos para nodos que no tengan relación de precedencia, así como también queda sujeto al orden definido en la selección de nodos iniciales y adyacentes para realizar las recorridas DFS. Dicho lo anterior, se puede presentar como contraejemplo a la afirmación, que otro orden posible para el grafo presentado en a) podría ser:

   Lógica, Programación 1, Programación 2, Matemáticas 1, Matemáticas 2, Bases de Datos