Primer Parcial de Programación 3
29 de setiembre de 2016

- Este parcial dura **3** horas y consta de 7 carillas. El total de puntos es **40**.
- En los enunciados llamamos `C*` a la extensión de `C` al que se agrega el operador de pasaje por referencia `&`, comentarios en línea, declaración de tipos y enumerados como en `C++`, los operadore `new` y `delete` y el tipo de datos `bool`.
- **NO** se puede utilizar ningún tipo de material de consulta. Salvo que se indique lo contrario podrá usarse todo lo visto en el teórico, práctico y laboratorio sin demostrarlo, indicando claramente lo que se está usando.
- Justifique sus respuestas.

Se requiere:

I) Numerar todas las hojas e incluir en cada una el **nombre, cédula de identidad, número de página y cantidad de hojas entregadas**.

II) Utilizar las hojas de **un solo lado** y escribir con lápiz.

III) Iniciar cada ejercicio en hoja nueva.

Ejercicio 1 (13 puntos)

(a) Indique la veracidad de las siguientes afirmaciones justificando todos los pasos. Sea una función $f : N \rightarrow R^*$ una función arbitraria. Se cumple entonces que:

I. $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$

II. $f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$

(b) Considere un hospital que cuenta con S salas de internación, las cuales se identifican por un número $[0..S-1]$, y n pacientes internados. Considere las siguientes operaciones y sus respectivos órdenes:

- `int obtenerCantidadCamasLibres(Estructura E, int id)` Devuelve la cantidad de camas libres de una sala identificada por su número de sala, en $O(1)$ peor caso.
- `void internarPaciente(Estructura E, char * documento)` Se ingresa un nuevo paciente en el hospital, identificado por documento, en $O(log\ n + S)$ caso promedio
- `void altaMedicaPaciente(Estructura E, char * documento)` Se realiza el egreso de un paciente que se encontraba internado, dado su documento, en $O(log\ n)$ caso promedio.
- `int* obtenerListaDeSalasDisponibles(Estructura E)` Retorna una lista de salas con camas libres (no asignadas a pacientes) ordenadas en forma ascendente según su número identificador de sala, en $O(S)$ peor caso.
- `int obtenerIdentificadorSalaPaciente(Estructura E, char * documento)` Dado el documento de identidad de un paciente dentro del hospital, devuelve el número de sala en el que se encuentra internado, en $O(log\ n)$ caso promedio.

I. Dibuje un diagrama detallando una estructura de datos que permita realizar las operaciones anteriores y respete los órdenes indicados de cada una de ellas. Justifique el cumplimiento de dichos órdenes.

II. Implemente el pseudocódigo de la operación `internarPaciente` definiendo los cambios necesarios en la estructura. No es necesario especificar la implementación de operaciones de TADs cuyas estructuras fueron utilizadas para resolver la parte anterior.

Solución:

(a) Sea $k \in \mathbb{R}^+$. Vamos a probar que $kn \in O(n)$ y $kn \in \Omega(n)$.

Para probar $kn \in O(n)$ hay que encontrar $c_1 \in \mathbb{R}^+$ y $n_1 \in \mathbb{N}$ tal que $\forall n > n_1$ se cumple $kn \leq c_1 n$. Eligiendo $n_1 = 0$ y $c_1 = k$ se cumple de manera evidente lo requerido. De manera similar, para probar $kn \in \Omega(n)$ lo que hay que demostrar es que $kn \geq c_2 n$ para todo $n > n_2$ para algún par $n_2 \in \mathbb{N}$ y $c_2 \in \mathbb{R}^+$. Y esto también se cumple eligiendo $n_2 = 0$ y $c_2 = k$.

Entonces en particular se puede ver que $4n \in O(n)$ y $(1/2)n \in \Omega(n)$.

I. FALSO.
Se va a probar usando la regla del límite (ver Ejercicio 4 del Práctico 2). Tomando $f(n) = 4n$, se cumple que $f(n) \in O(n)$. Pero como $\lim(\frac{2^n}{2^{4n}}) = 0$ se concluye que $2^{4n} \notin O(2^n)$.

II. FALSO.
Se va a probar mediante la definición de Ω. Se toma $f(n) = (1/2)n$, que cumple $f(n) \in \Omega(n)$. Hay que demostrar que para todo par $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$, existe $n > n_0$ tal que $2^{n/2} < c \cdot 2^n$. O sea que hay que probar que existe $n > n_0$ que cumple $n/2 < \log c + n$, es decir $n > -2 \log c$. Y esto se cumple para cualquier n mayor que $\max(-2 \log c, n_0)$.

(b) I. Diagrama de la multiestructura...

II. Pseudocodigo

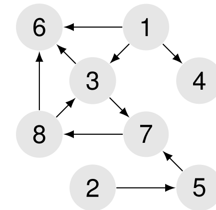
```
struct Sala{
  int id;
  int cantidadTotal;
  int cantidadLibres;
}

struct Paciente{
   char *documento;
   Sala *sala;
}

struct ABB{
   Paciente *paciente;
   ABB *izq;
   ABB *der;
}


struct Estructura{
   Sala *salas;
   ABB *pacientes; //ordenado por documento
}

void internarPaciente(Estructura E, char *documento){
   int * lista = obtenerListaDeSalasDisponibles(E);
   if (lista != NULL) {
         int id = lista[0];
         Sala sala = E.salas[id];
         sala->cantidadLibres--;
         Paciente * paciente = new Paciente;
         paciente->documento = documento;
         paciente->sala = sala;
         InsertarABB(E.pacientes, paciente);
   }
}
```

Ejercicio 2 (14 puntos)

Sea $G = (V, A)$ un grafo dirigido simple. En una recorrida DFS se denota con A_T el conjunto de aristas que pertencen a los árboles de cubrimiento, llamadas aristas *de árbol* (tree). Las aristas que pertencen a $A - A_T$ se clasifican en: *de regreso* (back), *directas* (forward) y *cruzadas* (cross).

(a) Defina las aristas de regreso, directas y cruzadas.

(b) Muestre los árboles de cubrimiento que resultan de la recorrida DFS en el grafo de la figura de la derecha (cuando haya que elegir entre más de un vértice, tanto al recorrer las listas de adyacencia como al iniciar un nuevo árbol, se debe elegir el vértice identificado con el número menor). Incluya las aristas de regreso, directas y cruzadas, etiquetando cada una con su tipo. Escriba a la izquierda de cada vértice el tiempo en que fue descubierto (*prenum*) y a la derecha el tiempo en que se terminó de procesar (*postnum*). El tiempo se incrementa en una unidad cada vez que un vértice es descubierto o terminado de procesar. El primer vértice es descubierto en el tiempo 1.



(c) En esta parte se elimina la restricción establecida en la parte b de elegir el menor cuando haya más de un vértice para elegir (es decir, se permite elegir cualquiera de ellos). ¿Podría una recorrida DFS de ese mismo grafo dar como resultado un único árbol de cubrimiento? Es caso afirmativo muestre el ejemplo. En otro caso explique por qué no puede haber menos de dos árboles

(d) Abajo se muestra el seudocódigo incompleto de una recorrida DFS. Se tratan como variables globales el grafo $G = (V, A)$, la cantidad *n* de vértices en *V*, el entero *tiempo*, la variable booleana *hay_ciclo* y los arreglos *prenum* y *postnum*, ambos de *n* enteros. Cada arista tiene un atributo llamado *tipo* cuyo valor puede ser: *arbol*, *regreso*, *directa*, *cruzada* o estar indefinido.

Al finalizar la recorrida debe quedar asignado el valor correcto de `(v,w).tipo` para cada $(\mathtt{v}, \mathtt{w}) \in A$, y el valor de *hay_ciclo* debe ser `true` si y sólo si en G hay algún ciclo.

```
Recorrida
  Para i desde 1 hasta n
    prenum[i] ← ∞
    postnum[i] ← ∞
  Para cada a ∈ A
    a.tipo ← indefinido
  hay_ciclo ← false
  tiempo ← 0
  Para i desde 1 hasta n
    <invocacion>
```

```
DFS(v)
  <preprocesamiento>
  Para cada (v,w) ∈ A
    Si prenum[w] = ∞
      (v,w).tipo ← arbol
      DFS(w)
    en otro caso
      <caso no-arbol>
  <posprocesamiento>
```

Escriba el seudocódigo de `invocacion`, `preprocesamiento`, `posprocesamiento` y `caso no-arbol` que completan los algoritmos **Recorrida** y **DFS(v)**. No se puede utilizar otras variables o funciones.

Solución:

(a) Las aristas de $A - A_T$ se clasifican según la relación ancestro-descendiente de sus vértices en los árboles resultado de la recorrida DFS.

Regreso Son las aristas que van desde un vértice hacia un ancestro.

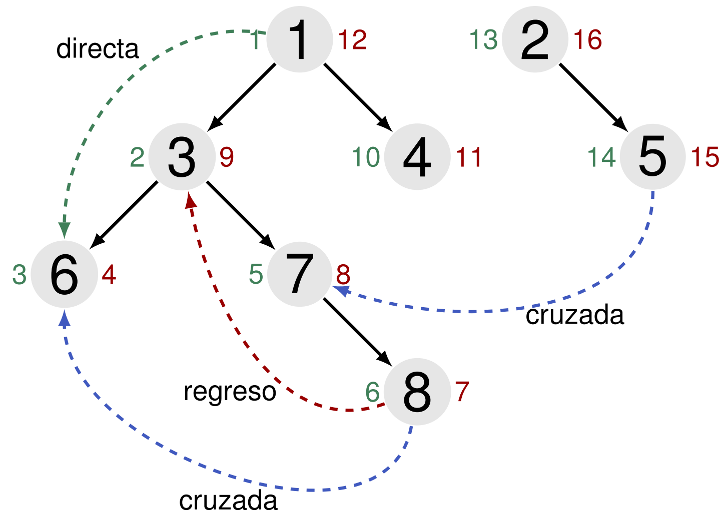
Directa Son las aristas que van desde un vértice hacia un descendiente.

Cruzada Son las aristas que van desde un vértice a otro que no es ancestro ni descendiente.

(b) En el diagrama se muestran los dos árboles que resultan de la recorrida, con raíces en los vértices 1 y 2. Las aristas de árbol se muestran con líneas rectas, negras. Las aristas de $A - A_T$ se muestran con líneas curvas con trazo discontinuo: en rojo las de regreso, en verde las directas y en azul las

cruzadas. A la izquierda de cada vértice, en verde, se muestra el tiempo en que fue descubierto, y a la derecha, en rojo, el tiempo en que se terminó de procesar.

Se puede ver que hay aristas cruzadas entre vértices de distintos árboles y también entre vértices del mismo árbol. Asimismo, hay aristas cruzadas que van hacia un vértice más profundo y otras que van hacia un vértice menos profundo.



(c) No puede haber menos de dos árboles porque los vértices 1 y 2 son fuentes por lo que necesariamente deben ser raíces.

(d)

Los valores de `prenum` y `postnum` se pueden usar para determinar, respectivamente, si un vértice ha sido visitado y si se ha terminado de procesar. La condición $\texttt{prenum}[\texttt{i}] = \infty$ significa que el vértice `i` no ha sido visitado. Sólo cuando esa condición se cumple se debe invocar `DFS(i)`, tanto en <invocacion> como (tal como se ve en el código) recursivamente. Este último caso ocurre al procesar la arista `(v,w)` que, por lo tanto, es una arista de árbol.
Si al procesar la arista se determina que no pertenece al árbol, se comparan los valores de `prenum[v]` y `prenum[w]`. Si $\texttt{prenum}[\texttt{v}] < \texttt{prenum}[\texttt{w}]$, entonces `v` es ancestro de `w`, por lo que la arista `(v,w)` es directa. Por otra parte, toda arista de retorno o cruzada va hacia un vértice que fue visitado antes (esto es, $\texttt{prenum}[\texttt{v}] > \texttt{prenum}[\texttt{w}]$). Las aristas cruzadas van hacia un vértice que ya se terminó de procesar (en este aspecto son similares a las directas) lo cual se comprueba con $\texttt{postnum}[\texttt{w}] \neq \infty$. Las aristas de retorno van hacia un ancestro, por lo tanto hacia un vértice que todavía no se terminó de procesar ($\texttt{postnum}[\texttt{w}] = \infty$), lo cual manifiesta la existencia de un ciclo formado por una secuencia de aristas de árbol desde `v` hasta `w` y la arista de retorno desde `w` hasta `v`.

```
<invocacion>
   Si prenum[i] = ∞
     DFS(i)

<preprocesamiento>
   tiempo ← tiempo + 1
   prenum[v] ← tiempo

<posprocesamiento>
   tiempo ← tiempo + 1
   postnum[v] ← tiempo
```

```
<caso no-arbol>
   Si prenum[v] < prenum[w]
     (v,w).tipo ← directa
   en otro caso
     // prenum[v] > prenum[w]
     Si postnum[w] ≠ ∞
       (v,w).tipo ← cruzada
     en otro caso
       // postnum[w] = ∞
       (v,w).tipo ← retorno
       hay_ciclo = true
```

Ejercicio 3 (13 puntos)

Merge sort es una estrategia de tipo Divide& Conquer para ordenar arrays que consiste en dividir un array en partes de igual tamaño, ordenar cada parte, y luego utilizar las partes ordenadas para ordenar el array original.

Suponga que en lugar de dividir a la mitad cada paso del Merge Sort como se vió en el teórico, el array a ser ordenado se divide en tres, se ordena cada tercio, y finalmente se las combina usando una función de merge de tres partes.

Como ayuda se recuerda el pseudocodigo de la función Merge vista en el teórico. Se ignoran los casos base.

```
C = output [largo = n]
A = 1er array ordenado [largo n/2]
B = 2do array ordenado [largo n/2]

Merge(Arreglo A, Arreglo B, Arreglo C)
    i = 1
    j = 1
    for k = 1 to n
        if A(i) < B(j)
            C(k) = A(i)
            i++
        else [B(j) < A(i)]
            C(k) = B(j)
            j++
        end
    end
```

(a) Escriba el pseudocódigo de la función *3_way_merge* que sustituye a la Merge presentada (el array de entrada no tiene repetidos).

(b) ¿Cuál es el orden asintótico del tiempo de ejecución de este algoritmo? Justifique brevemente utilizando los conceptos volcados en clase.

- $O(n)$
- $O(nlog(n))$
- $O(n(log(n))^2)$
- $O(n^2log(n))$

Solución:

(a)

```
D = output [largo = 3n]
A = 1er array ordenado [largo n]
B = 2do array ordenado [largo n]
C = 3er array ordenado [largo n]

3_way_merge(Arreglo A, Arreglo B, Arreglo C, Arreglo D)
    i = 1
    j = 1
    k = 1
    for l = 1 to 3n
        if A(i) < B(j) && A(i) < C(k)
            D(l) = A(i)
            i++
        else if B(j) < A(i) && B(j) < C(k)
            D(l) = B(j)
            j++
        else
```

```
            D(l) = C(k)
            k++
        end
    end
```

(b) $O(n)$:
Se puede observar claramente que cada arreglo se recorre una única vez, por lo tanto siempre se realizarán una cantidad $3n * k$ operaciones, donde k es una constante arbitraria. De esta forma es que $O(Merge) \in O(n)$. $O(nlog(n)), O(n(log(n))^2), O(n^2log(n))$:
Es fácil ver que asintóticamente $n \leq nlog(n)$, $n \leq n(log(n))^2$, $n \leq n^2log(n)$ y por lo tanto, recordando la definición de orden: $O(Merge) \in O(nlog(n))$, $O(Merge) \in O(n(log(n))^2)$ y $O(Merge) \in O(n^2log(n))$.

Nota: en la corrección se consideró correcta una interpretación de la letra que se hizo en alguno de los salones en la que la pregunta refería al Merge Sort y no a la 3_way_merge.