SOLUCION
Primer Parcial de Programación 3 (4/10/2013)

Instituto de Computación, Facultad de Ingeniería

- Este parcial dura **4** horas y contiene 3 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 &, las sentencias *new*, *delete*, el uso de *cout* y *cin* y el tipo `bool` predefinido en `C++`.
- NO se puede utilizar ningún tipo de material de consulta. Salvo que se indique lo contrario puede usarse todo lo visto en el teórico, práctico y laboratorio sin demostrarlo, indicando claramente lo que se está usando.

Se requiere:

i. Numerar todas las hojas e incluir en cada una el nombre y la cédula de identidad.
ii. Utilizar las hojas de un sólo lado y escribir con lápiz.
iii. Iniciar cada ejercicio en hoja nueva.
iv. Poner en la carátula la cantidad de hojas entregadas, y un índice indicando en qué hojas respondieron cada problema.

Ejercicio 1 (14 puntos)

a) Dado un grafo **G = (V, E)** dirigido con ***n*** vértices y ***n-1*** aristas. ¿Cuántas componentes fuertemente conexas puede tener como máximo y cuántas como mínimo?. Justifique detalladamente su respuesta.

En el parcial se aclaró considerar ***n*** > 2

máximo: *n*. La mayor cantidad de componentes fuertemente conexas se dará cuando el grafo sea un árbol dirigido, en ese caso hay una arista entre cada par de vértices y cada vértice será una componente conexa.

mínimo: *2*. Probaremos que hay un caso en que hay 2 componentes y luego que no puede haber menos.
La forma de colocar las n-1 aristas de manera de conectar en forma fuerte la mayor cantidad de vértices es formar un ciclo dirigido con n-1 vértices y las n-1 aristas. En ese caso quedarían 2 componentes: 1 vértice aislado y el ciclo de n-1 vértices.

Veremos que no es posible que tenga menos componentes fuertemente conexas (1):
Si el grafo fuera una sola componente fuertemente conexa, todos los vértices deberían tener al menos grado de entrada 1 y grado de salida 1 entonces habría por lo menos ***n*** aristas.

b) Dadas las siguientes primitivas de un grafo dirigido:

```
Transpuesto: Grafo → Grafo
Inducido: Grafo × Conjunto de vértices → Grafo
DFS: Grafo × Vértice → Grafo
ObtenerVertices: Grafo → Conjunto de vértices
```

Sea un grafo **G = (V, E)** dirigido y un vértice **u∈V**. Escriba un algoritmo en base a las operaciones anteriores que encuentre la componente fuertemente conexa a la que pertenece **u**.

```
V' = ObtenerVertices(DFS (G,u) )
G' = Inducido (G, V')
G^T = Transpuesto (G')
V'' = ObtenerVertices(DFS (G^T,u) )
Sol = Inducido (G, V'')
```

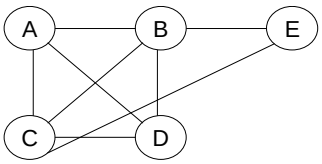
c) Considere un grafo **G = (V, E)** no dirigido. Sea **V*** ⊆ **V** tal que el subgrafo **G*** = **(V*, E*)** inducido por **V*** es completo.
Demuestre o de un contraejemplo de las siguientes afirmaciones.

1. Todos los vértices de **G*** aparecen necesariamente en el mismo árbol del bosque generado por la recorrida en profundidad (*DFS*) de **G**

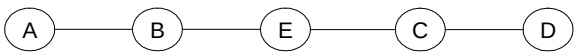
Una de las características del subgrafo **G*** es que es completo. Por lo tanto, todos sus vértices pertenecen a la misma componente conexa. A partir de esto, la recorrida DFS que pase por uno de sus vértices, necesariamente va a llegar a todos los demás.

2. Todos los vértices de **G*** aparecen necesariamente consecutivos en un camino de un árbol del bosque generado por la recorrida en profundidad (*DFS*) de **G**

Esto es falso. Dado el grafo:



Los vértices A, B, C, D forman un subgrafo completo. Haciendo DFS(A) se puede obtener el siguiente árbol:



Por lo tanto, no necesariamente todos los vértices aparecen consecutivos.

Ejercicio 2 (13 puntos)

a) Dado el siguiente algoritmo:

```
void algoritmo(int n, int* a) {
   int sum = 0;
   for (int i=1; i<n; i=i+2) {
        if (a[i-1] > a[i])
             sum++;
   }
   for (int i=1; i<n; i=i+2)
        swap(a, i-1, i);
   for (int i=1; i<n; i=i+2) {
        if (a[i-1] < a[i])
             sum++;
   }
}

void swap (int* a, int i, int j) {
   int aux = a[i];
   a[i] = a[j];
   a[j] = aux;
 }
```

que recibe como parámetro un arreglo **a** de tamaño **n** de enteros distintos, interesa calcular el costo en el mejor ($T_B(n)$) y peor ($T_W(n)$) caso, considerando simultáneamente las siguientes tres operaciones elementales:

1. sum++ tiene costo *c1*.
2. La comparación entre elementos de `a` con costo *c2*.
3. La asignación en el arreglo `a` con costo *c3*.

Se pide:

- Indique en forma genérica las entradas de tamaño n que maximizan el costo y luego calcule el costo del peor caso.
- Indique en forma genérica las entradas de tamaño n que minimizan el costo y luego calcule el costo del mejor caso.

Dominio de definición:
Para este problema en particular $D_n = \{(a_0, a_1, .., a_{n-1}) / a_i \in Z, i \in \{0, ..., n-1\}\}$.

Análisis del costo
Se observa que las líneas que se tienen que contar para el costo de una entrada $e \in D_n$ son:

1. `if (a[i-1]>a[i])` tiene costo c_2 y se ejecuta siempre $\left\lfloor \frac{n}{2} \right\rfloor$ veces.

2. `sum++` tiene costo c_1 y la cantidad de veces que se ejecuta puede ser entre 0 y $\left\lfloor \frac{n}{2} \right\rfloor$ veces.

3. `swap(a, i, j)` tiene costo $2c_3$ y la cantidad de veces que se ejecuta es $\left\lfloor \frac{n}{2} \right\rfloor$ veces.

4. `if (a[i-1] < a[i])` costo c_2 y se ejecuta siempre $\left\lfloor \frac{n}{2} \right\rfloor$ veces.

5. `sum++` tiene costo c_1 y la cantidad de veces que se ejecuta puede ser entre 0 y $\left\lfloor \frac{n}{2} \right\rfloor$ veces.

Peor caso:

Dado el análisis anterior, se verifica que las entradas que maximizan el costo son:

$(a_0, a_1, .., a_{n-1})$ tal que $a_0 > a_1$, $a_2 > a_3$, $a_4 > a_5$, ….
O sea: $a_{i-1} > a_i$ para *i* impar y $0 < i < n$.

Cabe destacar que para el análisis se tiene en cuenta que si `a[i-1]>a[i]` es verdadera si y solamente si, luego del `swap(a, i-1, i) a[i-1] < a[i]` es verdadera.

Por lo tanto:

$$T_W(n) = \left\lfloor \frac{n}{2} \right\rfloor (c_2 + c_1) + \left\lfloor \frac{n}{2} \right\rfloor 2c_3 + \left\lfloor \frac{n}{2} \right\rfloor (c_2 + c_1) = \left\lfloor \frac{n}{2} \right\rfloor 2(c_1 + c_2 + c_3)$$

Mejor caso:

Se verifica que las entradas que minimizan el costo son:

$(a_0, a_1, .., a_{n-1})$ tal que $a_0 < a_1$, $a_2 < a_3$, $a_4 < a_5$, ….
O sea: $a_{i-1} < a_i$ para *i* impar y $0 < i < n$.

Cabe destacar que para el análisis se tiene en cuenta que si `a[i-1]>a[i]` es falsa si y solamente si, luego del `swap(a, i-1, i) a[i-1] < a[i]` es falsa.

Por lo tanto:

$$T_B(n) = \left\lfloor \frac{n}{2} \right\rfloor c_2 + \left\lfloor \frac{n}{2} \right\rfloor 2c_3 + \left\lfloor \frac{n}{2} \right\rfloor c_2 = \left\lfloor \frac{n}{2} \right\rfloor 2(c_2 + c_3)$$

b) Determine si son correctas las siguientes afirmaciones, realizando una demostración detallada de la veracidad o falsedad de las mismas. Para esta parte puede utilizar propiedades vistas en el curso, enunciándolas correctamente antes de aplicarlas.

Regla del límite.

Dadas dos funciones arbitrarias f: $N \to R^*$ y g: $N \to R^*$

1. Si lím {n→+∞} (f(n) / g(n)) ∈ R^+ entonces f(n) ∈ O(g(n)) y g(n) ∈ O(f(n))
2. Si lím {n→+∞} (f(n) / g(n)) = 0 entonces f(n) ∈ O(g(n)) pero g(n) ∉ O(f(n))
3. Si lím {n→+∞} (f(n) / g(n)) = +∞ entonces f(n) ∉ O(g(n)) pero g(n) ∈ O(f(n))

1. $e^n 2^n \in O\left(e^{2n}\right)$

$\lim_{n\to+\infty} \frac{e^n 2^n}{e^{2n}} = \lim_{n\to+\infty} \frac{2^n}{e^n} = \lim_{n\to+\infty} \left(\frac{2}{e}\right)^n = 0$. Entonces por 2) $e^n 2^n \in O\left(e^{2n}\right)$.

2. $(n+n^2)! \in O((n^2)!)$

$\lim_{n\to+\infty} \frac{(n+n^2)!}{(n^2)!} = \lim_{n\to+\infty} \frac{(n+n^2)..(n^2)!}{(n^2)!} = \lim_{n\to+\infty} (n+n^2)..(n^2+1)! = +\infty$ entonces por 3) $(n+n^2)! \notin O((n^2)!)$.

3. $n! \in O(n^n)$. Para todo $n \geq 1$ se cumple que $n! \leq n^n$. Por lo tanto se cumple que $n! \in O(n^n)$ (considerando por ejemplo *K=1*).

Ejercicio 4 (13 puntos)

a) Defina el factor de carga de una tabla de dispersión abierta e indique si algún valor de éste evita el peor caso en futuras búsquedas.

Se denomina factor de carga (λ) de una tabla de dispersión, a la razón entre la cantidad de elementos de la tabla (N) y el tamaño de la misma.

$$\lambda = \frac{N}{TAMAÑO\_T}$$

No, ningún valor garantiza esto debido a que la dispersión y su comportamiento dependerán de los valores de datos de entrada y función de dispersión.

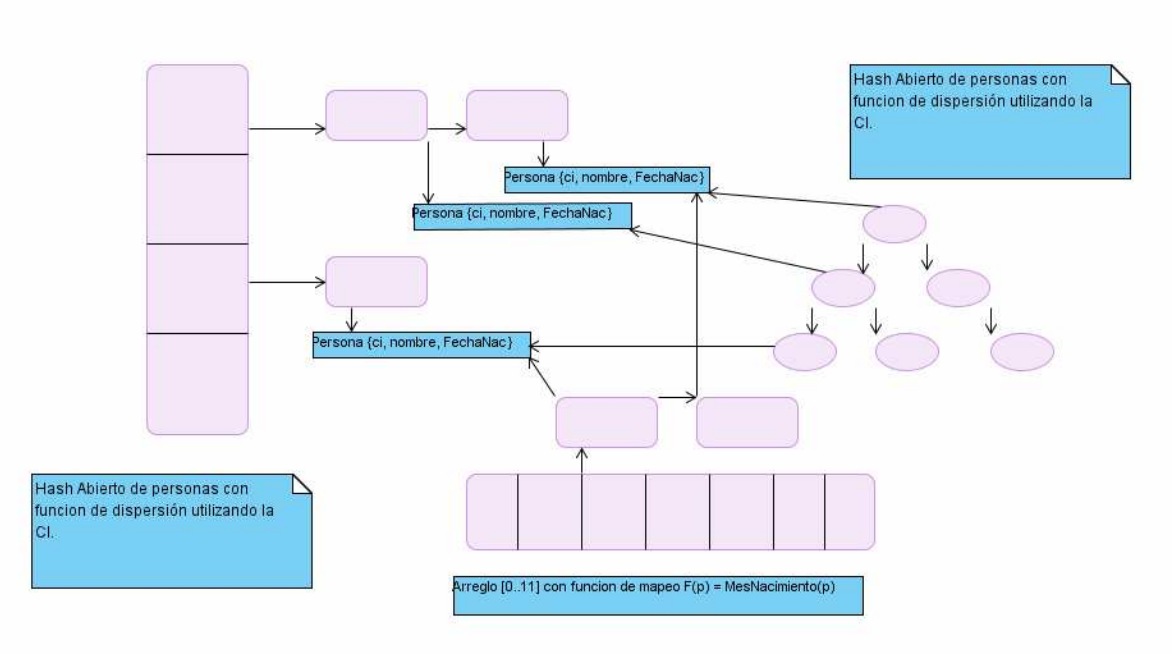
b) Considere un conjunto de N personas de las cuales se conoce su cédula de identidad, nombre y fecha de nacimiento (día, mes y año). Considerando las siguientes operaciones y sus respectivos órdenes:

1. *char * obtenerNombre(Estructura E, const char* cedula)* en O(1) promedio. Retorna el nombre de la persona con cédula de identidad *cedula*. Precondición: la persona con cédula de identidad *cedula* existe en *E*.
2. *ListaPersonas obtenerListaDePersonasPorMesDeNacimiento(Estructura E, int mes)* en Θ(1) peor caso. Retorna la lista de personas cuyo mes de fecha de nacimiento es *mes*.
3. *Persona obtenerPersonaMayorEdad(Estructura)* en Θ(1) peor caso. Retorna la persona de mayor edad en *E*. En caso de existir varias con la misma fecha de nacimiento retorna cualquiera.

Dibuje un diagrama detallando una estructura de datos que permita realizar las operaciones anteriores y respete los órdenes indicados de cada una de ellas, justificando porque se cumplen los órdenes de ejecución.

Solución

Hash dispersión abierta utilizando función de dispersión por CI. Los elementos son punteros a nodos persona que conteniendo la información de estas.
Arreglo[0..11] a lista de punteros a nodo persona nacidas. La inserción se realiza considerando en la lista correspondiente a Arreglo[i] donde i es el mes de nacimiento de la persona.
Heap de punteros a personas considerando el orden descendente por edad de estas para mantener y minimizar los costos de inserción y eliminación de personas.



c) Implemente la operación en C*:

```
void altaPersona(Estructura * E, const char * cedula, const char * nombre, fecha fechaNacimiento)
```

que da de alta en la estructura *E* de la parte b), la persona con cédula de identidad *cedula*, nombre *nombre* y fecha de nacimiento *fechaNacimiento*. Defina todos los tipos datos necesarios.

Puede utilizar:

- los tipos de datos *Persona* y *fecha* con operaciones de creación y selectoras.
- operaciones sobre los tipos de datos vistos en este curso o anteriores definiendo, como se pide, dichos tipos de datos y explicando qué hace cada operación.

La implementación dependerá de cada estructura, se evalúa la correcta definición de tipos: Persona, Hash de Personas, Lista de Personas, Heap de Personas y Personas por Mes; asi como la coherencia en algoritmo de inserción de personas en la estructura planteada.