# Examen de Programación 3

## 12 de diciembre de 2016

En recuadros con este formato aparecerán aclaraciones que cumplen una función explicativa pero que no eran requeridos como parte de la solución.

# Parte obligatoria

Esta parte es eliminatoria. Para la aprobación del examen debe obtenerse un mínimo del **50 % de esta parte (20 puntos)**. En caso de no llegar a dicho mínimo, **NO** se corregirán los problemas.

**Pregunta 1 (12 puntos)**

Asuma que $\forall n \geq n_0, f(n) \geq g(n)$ y que $f(n), g(n) \in \Theta(h(n))$.

(a) Utilizando la regla del limíte indique el/los órden(es) posibles de: $f(n) - g(n)$.

**Solución:**

(a) Considerando los siguientes limites:

$$Lim_{n\to+\infty}\frac{f(n)}{h(n)} = a$$

$$Lim_{n\to+\infty}\frac{g(n)}{h(n)} = b$$

Observamos que:

$$Lim_{n\to+\infty}\frac{f(n)-g(n)}{h(n)} = Lim_{n\to+\infty}\frac{f(n)}{h(n)} - Lim_{n\to+\infty}\frac{g(n)}{h(n)} = a - b$$

Por lo que:

Si $a = b \Rightarrow f(n) - g(n) \in O(h(n))$
si $a > b \Rightarrow f(n) - g(n) \in \Theta(h(n))$

**Pregunta 2 (14 puntos)**

Considere el algoritmo mostrado a la derecha.

(a) ¿Qué problema resuelve?

(b) Si se toma como operación básica la comparación con elementos del arreglo $A$, ¿cuándo se da el peor caso y por qué?

(c) Explique si el algoritmo es estable o no. En caso de no ser estable modifíquelo para que sea estable.

```
1 Procedimiento(A[1..n])
2     for i = 2 to n
3         dato ← A[i]
4         j ← 1
5         while dato > A[j]
6             j++
7         for h = i downto j + 1
8             A[h] ← A[h-1]
9         A[j] ← dato
```

**Solución:**

(a) El algoritmo ordena $A$ de manera creciente.

Es una variante de `InsertionSort` en la que la búsqueda del punto de inserción se hace desde el inicio del arreglo.

(b) El peor caso se da cuando $A$ está ordenado de manera creciente estricta. Esto es porque en ese caso la comparación de la línea 5 se hace $i$ veces.

Los ciclos anidados de las líneas 2 y 5 provocan que la cantidad de comparaciones pertenezca a $\Theta(n^2)$. La cantidad exacta es $\sum_{i=2}^{n} i = (n+2)(n-1)/2$.

(c) Un métdo de ordenación es estable si al terminar mantiene los elementos de igual valor en el mismo orden relativo en que se encontraban originalmente.

Este algoritmo no es estable porque la iteración de la línea 5 se interrumpe si $A[i] = A[j]$. En ese caso $A[j]$ queda en la posición $j+1$, y $A[i]$ queda en la posición $j$.

Supongamos que en $A[1..i-1]$ hay algún elemento igual a $A[i]$. Sea $k$ el mínimo $k < i$ para el que se cumple $A[k] = A[i]$. La iteración de la línea 5 se interrumpe en $j = k$. En la línea 8 el elemento $A[k]$ es movido a la posición $k+1$. Y en la línea 9 se mueve a la posición $k$ el elemento $A[i]$, quedando en orden inverso los elementos que originalente estaban en $A[k]$ y en $A[i]$.

Para que sea estable hay que cambiar la línea 5:

```
5     while (j < i) AND (dato >= A[j])
```

Si en $A[1..i-1]$ hay algún elemento igual a $A[i]$, el ciclo va a terminar cuando $j = i$. De esta manera el siguiente ciclo no se ejecuta. Finalmente en la línea 9 se copia $A[i]$ en la posición $i$ de $A$, dejando todo como estaba.

**Pregunta 3 (14 puntos)**

Dada una recorrida (DFS o BFS) en un grafo no dirigido.

(a) Defina arista *forward*.

(b) ¿Es posible tener más aristas *back* que *forward* si la recorrida es DFS? Justifique.

(c) ¿Es posible tener más aristas *cross* que *back* si la recorrida es BFS? Justifique.

(d) Escriba una condición para una artista $(v, w)$ en función de los prenums y posnums de una recorrida DFS que sea verdadera si y solo si la artista es *back*.

**Solución:**

(a) Se denomina *forward* a una arista no *tree* que va hacia un descendiente en el árbol inducido por la recorrida.

(b) Falso. No es posible por que en un grafo no dirigido las aristas *forward* y *back* coinciden.

(c) Verdadero, pues en una recorrida BFS sobre un grafo no dirigido solo puede haber aristas *cross*.

(d) $(prenum(w) < prenum(v)) \wedge (posnum(w) > posnum(v))$

# Problemas

**Problema A (30 puntos)**

Kailor se propone determinar los recorridos con los que más rápidamente puede llegar a cada una de las paradas de ómnibus de su barrio. Para ello identifica cada una de las esquinas de la ciudad (algunas de las cuales son las paradas de ómnibus de su barrio) con un número entre 1 y $n$ y mantiene en el arreglo bidimensional $C$ el tiempo que le lleva recorrer cada una de las cuadras. Asigna $C[i,i] = 0$ y, para $i \neq j$, $C[i,j] = \infty$ si no hay una cuadra desde $i$ hasta $j$. Se denota con $P$ el conjunto que contiene los identificadores de las esquinas en que se encuentran cada una de las paradas y con $K$ la esquina de la casa de Kailor. Se asume que desde $K$ se puede llegar a cada una de las esquinas en $P$.

Se quiere construir una estructura $E$ mediante la cual dada una esquina cualquiera $p \in P$ se pueda obtener en tiempo $O(r)$ el recorrido más rápido desde $K$ hasta $p$, siendo $r$ la cantidad de cuadras del recorrido.

(I) Escriba el seudocódigo de `Estructura(C, n, K, P)` que devuelve la estructura $E$. El tiempo de ejecución debe ser polinómico en $n$ y el algoritmo debe terminar tan pronto como en $E$ se disponga de la información necesaria para obtener el recorrido desde $K$ hasta cada esquina en $P$.

(II) Escriba el seducódigo de `Recorrido(E, p)` que dadas la estructura $E$ y una esquina $p \in P$ imprime las esquinas del recorrido más rápido desde $K$ hasta $p$, en el orden en que deben visitarse. El tiempo de ejecución debe ser $O(r)$, siendo $r$ la cantidad de cuadras del rcorrido.

**Solución:**

(I) El problema se modela con un grafo simple, dirigido (ya que no se dice que $C[i,j] = C[j,i]$) y con aristas ponderadas. Lo que se busca son los caminos más cortos desde un vértice hasta otros vértices. Esto es resuelto por el algoritmo de Dijkstra que en su versión más sencilla, que es la que aquí se muestra, tiene tiempo de ejecución $\Theta(n^2)$.

```
Estructura(C, n, K, P)
    D, E: Arreglo de n elementos
    V ← {1...n}
    FORALL i ∈ V
        D[i], E[i]  ← ∞
    u ← K
    D[u], E[u] ← 0
    S ← {u}
    WHILE P ∩ S ≠ P
        FORALL w ∈ V − S
            D[w] = mín(D[w], D[u] + C[u, w])
        v ← arg mín_{w∈V−S}{D[w]}
        E[v] ← u
        S ← S ∪ {v}
        u ← v
    RETURN E
```

En $E$ se mantiene el árbol que resulta de la recorrida, con aristas desde cada nodo a su padre. Con $E[K] = 0$ se indica que $K$ es la raíz del árbol. El algoritmo termina cuando todos los vértices de $P$ están en $S$, lo cual va a ocurrir porque en cada paso del ciclo externo se agrega a $S$ un nuevo vértice hacia el cual hay camino desde $K$, y se sabe que todos los vértices de $P$ cumplen esa condición.

(II)

```
Recorrido(E, p)
    IF p ≠ 0
        Recorrido(E, E[p])
        imprimir(p)
```

**Problema B (30 puntos)**

Sea $G = (V, E)$ un grafo dirigido, se denomina *clique* a todo subgrafo de $G$ que sea completo. Encontrar un *clique maximal* en un grafo consiste en encontrar un clique con la mayor cantidad de vértices posibles. Sea pretende encontrar un *clique maximal* en un grafo mediante *backtracking* utilizando una formulación con tuplas de tamaño fijo.

(I) Defina la forma de la tupla.

(II) Defina restricciones explícitas, implícitas y función objetivo.

(III) Defina un predicado de poda.

(IV) Considere el siguiente algoritmo:

```
clique_maximal(G):
  conj_vertices = conjunto_de_partes(G.vertices) // subconjutos de V
  clique = {}

  For conj in conj_vertices:
    If es_clique(conj, G) and (tamanio(conj) > tamanio(clique)):
      clique = conj

  return clique
```

¿El algoritmo planteado es más eficiente que su propuesta de backtracking? Justifique

**Solución:**

(I) **Forma de la tupla** Tupla $t = \langle t_1, \cdots, t_n \rangle$, de largo fijo $n$, siendo $V = \{v_1, \cdots, v_n\}$ el conjunto de vértices del grafo. Cada componente indica la pertenencia de dicho vértice en el clique.

(II) **Restricciones Explícitas**

- Pertenencia de cada vértice de $G$ en el clique.
  $t_i \in \{0, 1\}$

**Restricciones Implícitas**

- Si dos vértices pertenecen al clique es por que hay una arista de ida y otra de vuelta entre ellos.
  Si $t_i = t_j = 1$ con $i \neq j \implies (v_i, v_j) \in E \wedge (v_j, v_i) \in E$

**Función Objetivo** Se busca maximizar la cantidad de vértices del clique.
La función objetivo es

$$f = \max_{t \in T} \sum_{i=1}^{n} t_i$$

siendo $T$ el conjunto de tuplas solución.

(III) **Predicado de poda** Sea $s = \langle s_1, \cdots, s_n \rangle$ la mejor solución hasta el momento y la tupla en construcción $t = \langle t_1, \cdots, t_k, \_, \cdots, \_ \rangle$, no se sigue construyendo $t$ si al considerar como parte del clique a los vértices faltantes de $t$ no se supera la cantidad de vértices de $s$.
Es decir, se poda si

$$\sum_{i=1}^{n} s_i > \sum_{i=1}^{k} t_i + (n - k)$$

(IV) El algoritmo planteado ejecuta $2^n$ veces la rutina $es\_clique$ cuyo tiempo de ejecución depende a su vez del tamaño del conjunto $conj$. El algoritmo basado en *backtracking* es más eficiente en tiempo de ejecución por lo siguiente:

- Considera menos subconjuntos que $2^n$, pues si una tupla no cumple con la restricción implícita o satisface el predicado de poda no se sigue con su construcción.
- Al verificar si el subconjunto especificado por $t$ en el paso k es clique se sabe que hasta el paso $k-1$ lo era. Esto disminuye la cantidad de cálculos en comparación con las ejecuciones independientes de la rutina $es\_clique$.