PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC 2133 — Estructuras de Datos y Algoritmos
Examen
Primer Semestre, 2017

Duración: 3 hrs.

1. Para cada una de las siguientes afirmaciones, diga si es *verdadera* o *falsa*, **siempre justificando** su respuesta.

   a) Quick Sort es un algoritmo de ordenación estable. **Respuesta:** Falso. Basta con dar un contraejemplo.

   b) Sea e la segunda arista más barata de un grafo dirigido acíclico G con más de dos nodos y aristas con costos diferentes. Entonces e pertenece al árbol de cobertura de costo mínimo para G. **Respuesta:** Verdadero. Si usamos el algoritmo de Kruskal, la segunda arista más barata es siempre agregada al MST puesto que no puede formar un ciclo en el bosque construido hasta el momento.

   c) Radix Sort es un algoritmo de ordenación que puede ordenar n números enteros y cuyo tiempo de ejecución está siempre en $\Theta(n)$. **Respuesta:** El tiempo de ejecución de Radix Sort es $d(n + k)$, cuando los datos están en $[0, k]$. Basta entonces con que k sea suficientemente grande (por ejemplo, exponencial en n), para que el algoritmo no sea $\Theta(n)$

   d) Sea A un árbol rojo-negro en donde cada rama tiene n nodos negros y n nodos rojos. Si al insertar una clave nueva en A se obtiene el árbol A', entonces cada rama de A' tiene $n + 1$ nodos negros. **Respuesta:** Verdadero. Un árbol rojo negro como A corresponde a un árbol 2-4 "completamente saturado"; es decir, uno que contiene nodos con 3 claves. Al insertar una clave nueva, el árbol 2-4 aumentará su altura en 1. Esto significa que su equivalente rojo-negro debe tener un nodo negro más por cada rama.

   e) Si se tienen dos tablas de hash, una con direccionamiento abierto y otra cerrado, las dos del mismo tamaño m y el mismo factor de carga α, ambas ocupan la misma cantidad de memoria.

   f) Sea A un árbol AVL y ℓ_1 y ℓ_2 los largos de dos ramas de A. Entonces $|\ell_1 - \ell_2| \leq 1$. **Respuesta:** Falso. Basta dar un contraejemplo

   g) La operación de inserción en un árbol AVL con n datos realiza a lo más una operación *restructure* pero toma tiempo $O(\log n)$. **Respuesta:** Verdadero, puesto que la inserción debe revisar que el balance esté correcto a lo largo de la rama donde se insertó el dato. Y esa rama tiene tamaño $O(\log n)$.

   h) Si A es un ABB y n es un nodo de A, el *sucesor* de n no tiene un hijo izquierdo. **Respuesta:** Falso. La propiedad no se cumple en general cuando n no tiene un hijo derecho.

   i) Es posible modificar la implementación de la estructura de datos para conjuntos disjuntos vista en clases, de manera que permita des-unir dos conjuntos en $O(1)$.

   j) Como diccionario, una tabla de hash es más conveniente que un árbol rojo negro en cualquier aplicación. (Sin considerar la dificultad de implementación).

   k) Sea p una secuencia de dos o más números diferentes y sea q una permutación de p distinta de p. Adicionalmente, sean A_p y A_q los árboles binarios de búsqueda que resultan de, respectivamente, insertar en orden los elementos de p y q en árboles binarios de búsqueda vacíos. Entonces A_p y A_q son distintos. **Respuesta:** Falso.

   l) Re-hashing, el procedimiento que construye una nueva tabla de hash a partir de otra existente, toma tiempo $O(n)$ en una tabla con colisiones resueltas por encadenamiento de tamaño m que contiene n datos. **Respuesta:** Falso. El tiempo depende del tamaño de la tabla y del número de datos; específicamente es $O(m + n)$.

2. Dadas una secuencia de números enteros positivos no necesariamente distintos $\mathbf{x} = x_1, \ldots, x_n$ y una secuencia de bits $\mathbf{b} = b_1, \ldots, b_n$, se define la función:

$$\mathbf{x} \otimes \mathbf{b} = \sum_{i=1}^{n} (-1)^{b_i} x_i$$

a) Dé el pseudocódigo de una función que utilice backtracking y que reciba una secuencia de números $\mathbf{x}$ y un número N, y retorne *true* cuando existe un $\mathbf{b}$ tal que $\mathbf{x} \otimes \mathbf{b} = N$, y que retorne *false* en caso contrario.

Respuesta:
```
1 function f(i, s)
```

b) Dé un pseudocódigo iterativo para la función de a) que sea **mucho más** eficiente que su implementación de b). Analice su tiempo de ejecución.

3. Existen diversas maneras de implementar Quick Sort. Una de ellas es reemplazar *Partition* por una función *MedianPartition*, que es tal que *MedianPartition*(A, p, r) es el índice en donde se ubica la *mediana* del arreglo $A[p..r]$, una vez que éste está ordenado.

   a) Modifique el pseudo-código de QuickSort—mostrado abajo—de manera que use esta idea. Su pseudocódigo debe ser detallado y completo; es decir, no debe faltar ninguna función por implementar. Asegure, además, que el tiempo promedio de *MedianPartition* sea $O(n)$.

```
1  function Partition(A, p, r)
2      i ← p − 1
3      j ← p
4      while j ≤ r do
5          if A[j] ≤ A[r] then
6              i ← i + 1
7              A[i] ↔ A[j]
8          j ← j + 1
9      return i
10 procedure Quick-Sort(A, p, r)
11     if p < r then
12         q ← Partition(A, p, r)
13         Quick-Sort(A, p, q − 1)
14         Quick-Sort(A, q + 1, r)
```

   b) Argumente a favor de que el tiempo promedio de su función *MedianPartition* es $O(n)$.

   c) ¿Es posible alimentar a su versión de Quick Sort con un arreglo A para que tome tiempo $O(n^2)$? Argumente.

   d) ¿Cómo espera que su algoritmo funcione en la práctica? ¿Cree que una variante de esta idea podría funcionar mejor? ¿Cuál?

4. Dado un grafo $G = (V, E)$, una función de pesos $w : E \to \mathbb{R}^+$, y un nodo $s \in V$, nos interesa el problema de encontrar el costo del camino simple (sin ciclos) de *mayor* costo entre s y cada nodo del grafo.

   Un ex-alumno de IIC2133 argumenta que una modificación del algoritmo de Bellman-Ford (BF) puede resolver este problema. Su razonamiento es el siguiente:

   > *"Como el camino que buscamos no tiene ciclos, el camino más largo entre dos nodos tiene a lo más* $|V| - 1$ *aristas. De esta forma podemos primero buscar los caminos más largos de 1 arista, luego los de dos aristas, y así sucesivamente, tal como lo hace BF."*

   El algoritmo que propone es este:

```
1  procedure Init()
2      for each u ∈ V[G] do
3          d[u] ← −∞
4          π[u] ← nil
5  procedure CaminosMasCaros(G,s)
6      Init()
7      d[s] ← 0
8      for i ← 1 to |V| − 1 do
9          for each (u, v) ∈ E do
10             costo ← d[u] + w(u, v)
11             if costo > d[v] then
12                 d[v] ← costo
13                 π[v] ← u
```

Note que el único cambio respecto de la rutina de *CaminosMasCortos*, discutida en clases, son la línea 3 (cambio de ∞ por $-\infty$) y la línea 11 (cambio de $<$ por $>$).

En el resto de esta pregunta usted debe dar un argumento magistral que muestre por qué este algoritmo es **incorrecto**. Específicamente:

a) Entregue un contraejemplo que muestre que el algoritmo es incorrecto. **Respuesta:** El contrajemplo se construye con un grafo con un ciclo en el camino hacia un nodo.

b) Identifique una condición que debe cumplir el grafo G para que el algoritmo del alumno sea correcto. Demuéstrelo. (Ayuda: reduzca el problema de encontrar caminos más caros al de encontrar caminos más cortos.) **Respuesta:** Observamos primero que el algoritmo del alumno está basado en una subrutina de BF. Segundo, observamos que encontrar el camino más caro sin ciclos en un grafo $G = (V, E)$ con w es equivalente a encontrar el camino más corto sin ciclos en un grafo $G = (V, E)$ con $-w$. BF, ejecutado con $(G, -w)$, calculará el camino más corto sin ciclos exactamente cuando retorna "true". Y BF retorna true ssi no hay un ciclo negativo. $(G, -w)$ tiene un ciclo negativo ssi G tiene un ciclo, y por lo tanto esta es la propiedad que debe cumplir G para que el algoritmo del alumno sea correcta.

c) Hay parte del argumento del alumno que efectivamente está correcto: "los caminos más largos no pueden tener más de $|V| - 1$ aristas". Diga cómo, entonces, es posible modificar la idea de BF para encontrar caminos más largos. Analice el algoritmo resultante. **Respuesta:** La modificación no es sencilla ni eficiente. Esto es porque el camino más largo hasta U podría pasar por V, mientras que el camino más largo hasta V podría pasar por U.

La modificación de BF, cada vez que encuentra un nuevo camino hasta un nodo, lo guarda explícitamente, junto a su costo.

En la práctica esto significa que tenemos una matriz $d[v][i]$ que almacena el costo del i-ésimo camino sin ciclos hasta v que se ha encontrado. El camino, as su vez, se almacena en $\pi[v][i]$, por ejemplo, como una lista ligada.

Cuando en la línea 9 miramos (u, v) debemos usar esta arista completando cada camino hasta u para generar un nuevo camino hasta v. El costo de computar esto depende del tamaño de $d[v][]$, que, lamentablemente, en el peor caso debe llevar la cuenta de todos los caminos posibles desde la fuente hasta v, que es exponencial en $|V|$ ($O(|V|!)$, de hecho)

El algoritmo principal, entonces, realiza $|V| - 1$ iteraciones, pero cada iteración es $O(V!)$. El algoritmo es $O((|V| + 1)!)$.

5. En sus tiempos libres, Gianna Zecca, alumna de programación avanzada, ha estado estudiando algunos temas del libro de Cormen, Leiserson, Rivest y Stein. A pesar de que las estructuras de datos "Heap binario" y "Árbol Rojo-Negro" le parecen apasionantes, no logra entender la utilidad de los heaps. Según ella, ha encontrado una forma más o menos ingeniosa de usar árboles rojo-negro en vez de heaps dentro de los algoritmos de Prim y Dijkstra de tal manera de obtener *la misma complejidad asintótica*.

   a) ¿Está Zecca en lo correcto respecto a la complejidad asintótica de Prim y Dijkstra con árboles rojo-negro versus heaps? Para responder esta pregunta, piense en una forma "más o menos ingeniosa" de usar árboles dentro de esos algoritmos. **Respuesta:** Las dos operaciones que realizan Dijkstra/Prim sobre la cola son: extraer el mínimo y cambiar prioridad. Al implementar la cola con árboles, extraer el mínimo resulta simplemente de eliminar el mínimo, que es $O(\log n)$. Extraer el mínimo, en un heap, también es $O(\log n)$.

   Para el cambio de prioridad, en un árbol lo podemos hacer con una eliminación si (ingeniosamente) mantenemos un arreglo de punteros P tal que $P[v]$ contiene un puntero al nodo del árbol que representa al nodo v. Esta operación, en un árbol balanceado, toma $O(\log n)$. En el caso de un heap, la operación también toma $O(\log n)$. Concluimos que Zecca está en lo correcto.

   b) Escriba una batería de sólidos argumentos a favor de alguna de las dos afirmaciones:

      i. Es desventajoso implementar el algoritmo de Dijkstra (o Prim) usando árboles binarios de búsqueda balanceados.

ii. Jorge Baier no debiera complicarse en enseñar heaps, porque (1) los heaps no tienen ventaja alguna sobre los árboles (al menos en el caso de los algoritmos de Prim y Dijkstra) y (2) heapsort no tiene niguna ventaja práctica sobre Quick Sort.

Respuesta: Argumentamos en favor de i. Aunque la complejidad asintótica de ambas implementaciones es la misma, usar los heaps, para este algoritmo, tienen ventajas sobre árboles. Veamos el caso de la memoria. Un heap ocupa menos memoria que un árbol balanceado (cada nodo del árbol necesita almacenar punteros).

Respecto al tiempo, las operaciones de heap debieran ser más rápidas que con un árbol. En particular, la disminución de la prioridad necesita en el heap mover el elemento hacia arriba en el árbol; es decir, no se necesita revisar una rama completa. Esto no ocurre si usamos un ABB, porque la eliminación de un nodo requiere computar el sucesor (al menos), lo que implica en muchos casos revisar una rama completa. En la práctica esto se puede reflejar en los tiempos de ejecución de manera importante.

Para la extracción del más pequeño podemos dar un argumento similar.