PAUTA EXAMEN IIC2133 2019-1

1. Con respecto a los árboles rojo-negros:

a) **[2pt]** Justifica que es válido que un árbol rojo-negro con más de un nodo esté formado sólo por nodos negros.

Solución:

La justificación es posible hacerla directamente desde las propiedades de los **ARN**, o desde las propiedades de los **2-3 / 2-4.**

2-3 / 2-4:

Recordar que un nodo negro en un **ARN** representa un *nodo-2* en un árbol **2-3 / 2-4 [1pt]**, y un árbol **2-3 / 2-4** que sólo contiene *nodos-2* es válido. **[1pt]**

ARN:

De las propiedades de los **ARN:**

1. Un nodo puede ser rojo o negro
2. La raíz del árbol es negra
3. Los nodos rojos no pueden tener hijos rojos
4. La cantidad de nodos negros camino a cada una de las hojas debe ser la misma

[1pt]

Ninguna de estas propiedades menciona nada sobre los nodos negros con hijos negros, por lo que en principio no debería haber problema. La única propiedad que podría violarse en el caso que se pide es la 4, por lo que para que se cumpla la propiedad debe ser un árbol completo (con todos sus niveles llenos) **[1pt]**

b) **[2pt]** Justifica que un árbol rojo-negro construido sólo mediante la inserción de *n* nodos tiene al menos un nodo rojo.

Solución:

El primer paso de la inserción en un ARN es insertar el elemento como se haría en un ABB. El nodo recién insertado siempre es rojo. **[0.5pt]** Si este nodo genera conflictos estos se solucionan mediante rotaciones y cambio de color de sus ancestros, pero nunca se cambia de color el nodo recién insertado. Por lo tanto

al menos este nodo es rojo. **[1pt]** Eso sí, si el nodo queda como la raíz (*n = 1*), se debe pintar de negro. Es decir, esta propiedad solo se cumple para *n > 1*. **[0.5pt]**

c) Supongamos que todo nodo rojo en un árbol rojo-negro es "absorbido" por su padre, de modo que los hijos del nodo rojo pasan ahora a ser hijos de su abuelo (olvídate de lo que ocurre con las claves).

- **[1pt]** ¿Cuáles son los posibles números de hijos de un nodo negro del árbol después de que todos los nodos rojos han sido absorbidos?

  Solución:

  El mínimo posible es para los nodos negros que son hoja: 0 **[0.5pt]**

  El máximo posible es para los nodos negros que tienen dos hijos rojos, los cuales a su vez tienen dos hijos negros cada uno: 4 **[0.5pt]**

  [Formalidad] Por lo tanto el número de hijos de un nodo negro en este nuevo árbol puede ir entre 0 y 4.

- **[1pt]** ¿Qué puedes afirmar y ~~por qué~~ sobre las profundidades de las hojas del árbol resultante?

Solución:

[No tiene sentido preguntar por qué. Cualquier conclusión surge directamente de la definición del proceso.]

Cualquiera de estas explicaciones se considera como correcta:

- Para una hoja negra cualquiera a profundidad *P*, si existen *R* nodos rojos camino a la raíz del árbol, la profundidad de esta hoja pasa a ser *P - R,* ya que todos los nodos rojos en el camino desaparecen. **[1pt]**

- La profundidad del árbol puede verse reducida hasta en un 50% dependiendo de cómo estén distribuidos los nodos rojos. **[0.5pt]** Dar un ejemplo del mejor **[0.25pt]** y peor caso **[0.25pt]**.

2. Supongamos que tienes tres códigos compilados (es decir, códigos producidos por el compilador de C) S_1, S_2 y S_3, correspondientes a tres algoritmos de ordenación: *insertion sort*, *quick sort*, y *heap sort*. **Solo que no sabemos cuál código corresponde a cuál algoritmo**. Explica cómo identificar experimentalmente cuál código corresponde a cuál algoritmo, justificando tus decisiones en base a las propiedades de los algoritmos.

[Durante el examen se dijo que QuickSort escoge como pivote siempre al primer elemento de la secuencia, y que para hacer sus experimentos pueden ejecutar el programa con un input dado y medir cuánto tiempo toma]

Solución:

Para identificar los algoritmos experimentalmente podemos darles inputs específicos y medir cuánto demoran en resolverlos para determinar empíricamente su complejidad.

Las complejidades que buscamos son:

	Peor Caso	Caso Promedio	Mejor Caso
Insertion Sort	$O(n^2)$	$O(n^2)$	O(n)
Quick Sort	$O(n^2)$	O(n log n)	O(n log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)

[2pt por mencionar las complejidades. El caso promedio no es necesario]

El mejor caso de insertion sort se da cuando los datos vienen en orden. **[1pt]**

El peor caso de quick sort (con el primer elemento como pivote) también es cuando los datos vienen en orden. **[1pt]**

Heap sort demora lo mismo en todos los escenarios: es insensible al orden del input.

Complejidad con los datos ordenados:

Insertion Sort	O(n)
Heap Sort	O(n log n)
Quick Sort	$O(n^2)$

Una Posible estrategia:

Hay que medir cuánto demora cada algoritmo en ordenar un set de *n* datos **ordenados**, probando con distintos *n* para poder ver graficar los tiempos para cada *n*, trazando las curvas correspondientes. Es decir, si graficamos los 3 algoritmos juntos, la curva superior corresponde a QuickSort, la de al medio a Heap Sort, y la de abajo a Insertion Sort. **[2pts]**

Otra Posible estrategia:

Es posible identificar los tiempos de cada algoritmo comparándolos consigo mismos. En este caso basta con identificar dos y el tercero sale por descarte.

Para todo el procedimiento es similar al anterior, pero para cada *n* hay que probar con datos ordenados y aleatorios. Nuevamente graficando los resultados, pero esta vez tenemos un gráfico para cada algoritmo, y las curvas para cada gráfico son las de datos ordenados y datos aleatorios. **[1.5pt por razonamiento]**

- Quicksort es el que se demora más con los ordenados que con los aleatorios.
- Insertionsort es el que se demora más con los aleatorios que con los ordenados.
- Heapsort es el que se demora lo mismo con los aleatorios y los ordenados.

[0.5 pts por identificarlos]

3. Se tiene un *stream* de largo indefinido, que termina con el dato *d**. Cada dato *d* = (*u*, *v*) está formado por los nombres de dos ciudades, *u* y *v*, y representa la existencia de un vuelo directo de la ciudad *u* a la ciudad *v*. La idea es interpretar cada dato como una arista direccional que se agrega a un grafo inicialmente vacío.

a) [**2 pts.**] Explica cómo guardas el grafo en memoria; es decir, describe las estructuras de datos necesarias (tales como tablas de hash, listas de adyacencia, matriz de adyacencia, etc.), en lo posible ayudado por un dibujo. Ten presente que los datos originales son nombres de ciudades y no números. Define el **struct nodo** que usarías en C y qué cosas guarda.

Solución:

En primer lugar, el largo del stream es indefinido, y además no conocemos la lista de todas las ciudades. Eso significa que si o si es necesario el uso de un diccionario (tabla de hash) para guardar los datos, ya que no podemos definir de antemano una forma de guardarlos. **[0.75pt]** Por otro lado, es necesario usar el modelo de listas de adyacencia usando listas ligadas, ya que podemos ir agregándole vecinos a un nodo en O(1) sin costo adicional de memoria (a diferencia de la matriz de adyacencia que ocupa mucha más memoria de la necesaria y además no conocemos el tamaño de que debería tener o un arreglo de largo fijo para los vecinos) **[0.75pt]**

En esta tabla se guardan los pares <key, value> donde key son los nombres de las ciudades y value es un contenedor con toda la información de ese nodo.

Este contenedor tiene la siguiente forma:

```
struct nodo
{
   id; // El nombre de la ciudad
   vecinos; // la lista de adyacencia (nodos vecinos) de ese nodo
}
```

La lista de adyacencia tiene punteros directamente a los nodos vecinos

[0.5pt]

[Los tipos de C no son relevantes mientras quede claro qué se está guardando]

Considera ahora que cada dato en realidad es un trío de la forma *d = (u, v, w)* en que *w* es la duración del vuelo, en minutos (es decir, un número entero no negativo), entre *u* y *v*.

b) [**1 pt**] ¿Cómo cambia tu modelo del grafo en a)?

Solución:

La única diferencia es que ahora hay que guardar el peso *w* de las aristas del grafo en algún lado. **[0.5pt]** Ese lugar puede ser en la lista de vecinos del struct nodo, donde ahora en lugar de guardar punteros a los vecinos se guarden tuplas `(nodo*, peso)`. **[0.5pt]**

Finalmente, queremos aprovechar el hecho de que tenemos los datos representados como un grafo direccional con costos para buscar las rutas más cortas (rápidas) desde la ciudad *a* "a" todas las otras ciudades.

c) **[1pt]** ¿Qué algoritmo usarías para lograr esto? ¿Por qué?

Solución:

Dijkstra. **[0.5pt]** Porque el algoritmo de Dijkstra partiendo desde un nodo "*a*" genera un **árbol de rutas más cortas** desde ese nodo a todos los otros nodos, que es precisamente lo que queremos. **[0.5pt]**

d) **[1pt]** ¿Qué estructuras de datos adicionales necesitas para ejecutar eficientemente el algoritmo de c)? ¿Por qué?

Solución:

Dijkstra es un algoritmo codicioso que **en cada paso explora el siguiente nodo al que es más barato llegar desde "a" dado los nodos que se han explorado**. **[0.5pt]** Para esto necesita una **cola de prioridades** (heap) **[0.5pt]**

e) **[1pt]** ¿Es necesario modificar tu modelación en b) para usar este algoritmo? ¿Por qué?

Solución:

Sí. Cuando el algoritmo de Dijkstra explora un nuevo nodo lo hace junto con una referencia de "desde donde" se exploró ese nodo (también conocido como "el padre del nodo"), creando así el árbol. Para esto es necesario agregar un puntero al nodo padre en el struct. **[1pt]** *El algoritmo también debe poder identificar si un nodo ha sido o no explorado, aunque esto es posible hacerlo revisando si el nodo padre es nulo. Si se menciona un atributo de este estilo sin mencionar el nodo padre solo se da* **[0.25pt]**

4. Tienes que realizar un conjunto de tareas en un computador y queremos determinar el orden en que deben realizarse, teniendo en cuenta que existe una lista de dependencias (*a*, *b*), que representan el hecho de que la tarea *a* debe hacerse antes que la tarea *b*. De esta manera, se forma un grafo direccional que, sin embargo, podría tener ciclos; interpretamos estos ciclos como que todas las tareas que forman un determinado ciclo pueden ser realizadas al mismo tiempo (en paralelo).

Explica cómo puedes determinar el orden en que deben realizarse las tareas de manera que se cumplan todas las dependencias, indicando además los subconjuntos de tareas que se pueden hacerse en paralelo.

Solución:

Si no hubiera ciclos, el orden de las tareas es simplemente el orden topológico del grafo. Pero como el orden topológico no existe si el grafo tiene ciclos, debemos hacer algo al respecto. **[0.5pt]**

Lo primero que hay que hacer es identificar cuales son las tareas que deben ser realizadas al mismo tiempo. El detalle es que la cantidad de ciclos puede ser exponencial, por lo que no sirve "encontrar todos los ciclos". Lo que hay que hacer en lugar de eso es determinar la cantidad de ejecuciones en paralelo que hay que hacer. Llamaremos a cada una de esas una **super-tarea (ST)**: un conjunto de tareas que deben ser ejecutadas en paralelo.

La propiedad que nos interesa es que las STs corresponden a las componentes fuertemente conectadas del grafo con más de un elemento. [2.5pt]

Demostración:

Sabemos que las tareas de un ciclo corresponden a una ST, pero una ST podría contener a más de un ciclo.

Tomamos la siguiente definición de ciclo: si "a" pertenece a un ciclo, significa que existe una ruta de "a" hasta "a" pasando por al menos un nodo distinto de "a".

Si "x" pertenece a la misma ST que "a", significa que existe un ciclo que los contiene a ambos. Esto a su vez significa que existe una ruta de "a" a si misma pasando por "x".

Sea $T = \{a, x_1, \dots, x_n\}$ una ST que contiene a este nodo arbitrario "a"

$\{x_1, \dots, x_n\}$ corresponden a todas las tareas que están el la misma ST que "a".

Esto significa que para cada "x_i" existe una ruta de "a" a "x_i", y una ruta de "x_i" a "a".

Esto significa que para cada $i \neq j$, existe una ruta de "x_i" a "x_j", pasando por "a".

Es decir, T es un conjunto de nodos donde todos los nodos son alcanzables entre ellos.

Esa es la definición de componente fuertemente conectada.

$\therefore$ T es una CFC.

[1.5pt por demostrar o justificar correctamente esta propiedad]

Podemos usar el algoritmo para encontrar las CFC que se vió en clases para encontrar todas las CFC del grafo. Sea $\{T_1, \dots, T_n\}$ el conjunto de CFCs con más de un elemento, es decir un conjunto de STs.

Una vez encontradas todas las STs podemos eliminar del grafo los nodos que pertenecen a alguna ST, así como las dependencias entre elementos que pertenecen a la misma ST. Luego agregamos las STs como nuevas tareas al grafo, de la siguiente manera:

Si un nodo "u" pertenece a la ST T_i , y un nodo "v" pertenece a la ST T_j , y existe la dependencia (u, v), esta se elimina del problema y se agrega la dependencia entre STs, (T_i , T_j). Luego cada dependencia de la forma (a, u), donde a no pertenece a una ST, se elimina del problema y se agrega la dependencia (a, T_i). Por otro lado, cada dependencia de la forma (u, a), donde a no pertenece a una ST, se elimina del problema y se agrega la dependencia (T_i , a).

[1pt por la construcción de este nuevo grafo]

Ahora que tenemos un grafo acíclico podemos determinar su orden topológico usando el algoritmo visto en clases, que es el orden de tareas que se pide. **[0.5pt]**

5. Tenemos una tabla con las flechas de descubrimiento de *n* estrellas, donde cada fecha tiene el formato (*A, M, D, h, m, s*), que corresponde al año, mes, día, hora, minutos y segundos del descubrimiento. Queremos ordenar esta tabla en orden creciente, es decir, a partir de la estrella descubierta hace más tiempo.

a) **[4 pt.]** ¿Qué algoritmo deberías usar para ordenarlos en un tiempo mejor que *O(n log n)*; explica detalladamente cómo lo implementarías para este caso.

El algoritmo que es preferible en este caso es utilizar *radix sort* (1 pto).

Para implementar el algoritmo, se ordenará primero por la parte de la fecha más significativa y después se seguirá con las menos significativas, es decir, año, mes, día, hora, minutos, segundos (3 pt.) (también se puede realizar al revés). Para ordenar cada dígito se puede utilizar counting sort.

b) **[1 pt.]** ¿Cuál es su complejidad y de qué depende?

Si a contiene *n* estrellas, *d* es el número de dígitos y cada dígito puede tomar *k* valores distintos:

- ***O(d(n+k))*** en ordenar los n números.
- Cómo k es ***O(n)*** y d es constante obtenemos que el algoritmo toma ***O(n).***

c) **[1 pt.]** ¿Cuánta memoria adicional necesita este algoritmo?

Dado que Radix Sort debe guardar arrays auxiliares con los datos gasta una memoria adicional igual a *O(k + n)*, por ende, *O(n)*.