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

IIC 2133 — Estructuras de Datos y Algoritmos
Interrogación 1
Primer Semestre, 2017

Duración: 2 hrs.

1. *a*) ¿Es correcto que si la rutina *Partition* siempre realiza el *mínimo* número de intercambios posibles, se garantiza que *QuickSort* ejecuta en el menor tiempo posible (es decir, está en el mejor caso)? Argumente.

   b) Escriba el pseudocódigo de una versión inestable (pero correcta) de Merge Sort. Ahora diga cómo transformarla en estable.

   c) Suponga que insertará los objetos $o_1, o_2, \ldots, o_n$, en orden, a un Min-Heap, usando la operación de inserción estándar. ¿Cómo podrían ser las claves de estos objetos para obtener el mejor caso? ¿Y para obtener el peor caso? Diga, con precisión, cuántos intercambios se producen en cada caso si se usan las rutinas vistas en clases.

Solución

a) Falso (**0.5 puntos**). Los peores casos de `QuickSort` son aquellos en los que ocurren muchas particiones desbalanceadas. Una partición desbalanceada es cuando el pivote, luego de `Partition`, queda al inicio o al final del arreglo, y esto no tiene relación alguna con el número de intercambios realizados (puede haber una partición desbalanceada con muchos intercambios así como una sin ningún intercambio). Si ocurren muchas particiones desbalanceadas en `QuickSort`, entonces, cada llamada recursiva procesará un arreglo de $n-1$ elementos (arreglo original menos el pivote), por lo que se llamará n-veces hasta llegar a un arreglo de largo 1 (el Heap de llamadas se vuelve lineal) (**1.5 puntos** por argumentar).

Otra forma sería mostrar un contraejemplo. Por ejemplo, ordenar un arreglo ordenado con `QuickSort` (**2 puntos** si el contraejemplo está correcto y bien fundamentado).

b) Un algoritmo estable es aquel que mantiene el orden del input original cuando el algoritmo de comparación no es capaz de distinguir dos o más elementos. Va a garantizar que cuando dos objetos son iguales, los ordenará de la misma forma en la que estaban originalmente.

En el algoritmo de `Merge`, se comparan los objetos del arreglo de la derecha y de la izquierda, poniendo primero en el arreglo original el que sea mas pequeño. Ahora, si cambiamos la comparación de $\leq$ por solo $<$ en el algoritmo, podemos obtener una versión inestable de `MergeSort`, pues cuando hayan dos objetos iguales no entrará en la primera condición del `If` y entrará en el `Else`, que es donde se ubica el objeto del arreglo de la derecha primero (**1 punto** por escribir el pseudocódigo inestable).

El cambio se encuentra en la linea 9 del siguiente pseudocodigo:

1: **function** MERGESORT(A, p, q, r)
2: $i \leftarrow 0, j \leftarrow 0$
3: L nuevo arreglo de tamaño $q - p + 2$
4: R nuevo arreglo de tamaño $r - q + 2$
5: $L[0...q-p] \leftarrow A[p...q]$
6: $R[0...r-q-1] \leftarrow A[q+1...r]$
7: $L[q-p+1] \leftarrow R[r-q] \leftarrow \infty$

```
8:      for k ← p to r do
9:          if L[i] < R[j] then
10:             A[k] ← L[i]
11:             i+ = 1
12:         else
13:             A[k] ← R[j]
14:             j+ = 1
15:         end if
16:     end for
17: end function
```

Para hacerlo estable, se cambia $<$ por $\leq$ en la linea 9 (**1 punto** por decir cómo hacerlo estable).

c) Al insertar un elemento en un Min-Heap, este se coloca al final del arreglo y luego "sube"hasta que sea mayor o igual que su Padre. `DecreaseKey`, toma en el peor caso una complejidad de $\mathcal{O}(log(n))$ (con n el tamaño del heap) y esto ocurre cuando el elemento insertado llega al inicio del arreglo (pues es el más pequeño). El algoritmo tomará el mejor caso cuando el elemento insertado es el mayor de todos, por lo que no debe modificar su posición y `DecreaseKey` se ejecuta en $\mathcal{O}(1)$.

Los intercambios se producen cuando el elemento insertado debe subir por el árbol (cuando es menor que su padre), por lo tanto, para obtener un máximo número de intercambios, todos los elementos insertados deben ser menores que todos los elementos sobre el, de modo de llegar hasta lo más arriba del heap a través de los intercambios. Por lo tanto, insertar n elementos de mayor a menor será el caso con mayor números de intercambios (**0.5 puntos**). El total será de (**0.5 puntos** por el número correcto):

$$\sum_{i=1}^{n}(\lfloor \log(i) \rfloor)$$

Para obtener un número mínimo de intercambios, todos los elementos insertados deben ser mayores que los elementos sobre el, de modo de mantenerse en el lugar donde fueron insertados y no subir por el heap a través de intercambios. Por lo tanto, insertar n elementos de menor a mayor será el caso con menor número de intercambios (**0.5 puntos**). El total será de cero intercambios (**0.5 puntos**).

2. En muchas localidades de Chile se debe usar un *ferry* para poder continuar un camino. Aquí nos interesa el problema de cargar un ferry con dos filas de autos. El encargado de cargar este ferry debe decidir, para cada auto que está en en fila, a cuál fila del ferry ingresarlo, con el objetivo es maximizar el número de autos cargados en él. Lo que hace a este problema no trivial es que el total de autos que se puede cargar depende de las decisiones tomadas por el encargado.

   El problema está descrito por el largo de ambas filas del ferry, L, y una secuencia de largos, $\ell_1, \ell_2, \ldots, \ell_n$, donde ℓ_i representa el largo del vehículo en la posición i de la fila.

   a) Escriba una ecuación recursiva que describa el número máximo de autos que se puede cargar en el ferry. ¿Por qué es su ecuación correcta? Argumente.

   b) Escriba un programa iterativo que imprima qué autos deben ir en qué columna del ferry de manera de obtener una solución óptima.

   c) Analice el tiempo de ejecución del programa que escribió en *b*). Diga si su solución es exponencial (pero no pseudo-polinomial), polinomial, o pseudo-polinomial, demostrando su respuesta.

   Respuesta:

   a) Definimos la función $F(L_1, L_2, i)$ con $i \in [1, n]$ y $L_1, L_2 \in [0, L]$, como el máximo número de autos que se puede colocar en total dado que la fila 1 tiene largo L_1, la fila 2 largo L_2 y que se quiere agregar los autos $A[i], \ldots, A[n]$.

$$F(L_1, L_2, i) = \begin{cases} 0 & \text{if } L_1 < l_i \text{ y } L_2 < l_i \\ -\infty & \text{if } L_1 < 0 \text{ o } L_2 < 0 \text{ o } i > n \\ max\{F(L_1 - l_i, L_2, i+1) + 1, F(L_1, L_2 - l_i, i+1)\} + 1 & \text{en otro caso} \end{cases}$$

   Este algoritmo es correcto ya que el problema tiene la propiedad de sub estructura óptima, es decir, se puede resolver como una instancia de otros problemas más pequeños, en este caso sería subdividirlo según la decisión de ponerlo en la fila 1, fila 2 o que no quepa (en cuyo caso retorna $-\infty$).

   Puntaje:

   - **0.5 ptos** por casos base y borde.
   - **1 pto** por caso recursivo.
   - **0.5 ptos** por justificación.

b) El algoritmo es el siguiente:

```
1: function OPTIMALCARS(L, [l_1, ... l_n])
2:     //Cálculo con programación dinámica
3:     DP ← arreglo de tamaño (L + 1) × (L + 1) × (n + 2) inicializado en −∞
4:     Choose ← arreglo de tamaño (L + 1) × (L + 1) × (n + 2) inicializado en 0
5:     for i ← n + 1 to 1 do
6:         for L_1 ← 0 to L do
7:             for L_2 ← 0 to L do
8:                 if i = n + 1 then
9:                     DP[L_1][L_2][i] = 0
10:                    DP[L_1][L_2][i] = 0
11:                else
12:                    if L_1 < l_i ∧ L_2 < l_i then
13:                        DP[L_1][L_2][i] ← 0
14:                        Choose[L_1][L_2][i] ← 0
15:                    else
16:                        d_1 ← −∞, d_2 ← −∞
17:                        if L_i ≥ l_i then
18:                            d_1 ← DP[L_1 − l_i][L_2][i + 1]
19:                        end if
20:                        if L_i ≥ l_i then
21:                            d_2 ← DP[L_1][L_2 − l_i][i + 1]
22:                        end if
23:                        if d_1 = −∞ ∧ d_2 = −∞ then
24:                            DP[L_1][L_2][i] = 0
25:                            Choose[L_1][L_2][i] = 0
26:                        else if d_1 ≥ d_2 then
27:                            DP[L_1][L_2][i] = d_1 + 1
28:                            Choose[L_1][L_2][i] = 1
29:                        else if d_2 > d_1 then
30:                            DP[L_1][L_2][i] = d_2 + 1
31:                            Choose[L_1][L_2][i] = 2
32:                        end if
33:                    end if
34:                end if
35:            end for
36:        end for
37:    end for
38:    // Impresión resultados
39:    L_1 ← L, L_2 ← L
40:    for i ← n to 1 do
41:        decision ← Choose[L_1][L_2][i]
42:        if decision = 0 then
43:            break
44:        else if decision = 1 then
45:            L_1 ← L_1 − l_i
46:            print(Auto i va en fila 1)
47:        else if decision = 2 then
48:            L_2 ← L_2 − l_i
49:            print(Auto i va en fila 2)
50:        end if
```

51: **end for**
52: **end function**

Básicamente genera los casos posibles de la función de una forma bottom-up, generando primero los casos base y luego aprovechando los resultados anteriores para poder hacer los cálculos siguientes. $DP[L_1, L_2, i]$ guarda el número máximo de autos de la manera definida en a), $Choose[L_1][L_2][i]$ guarda la decisión de en que fila se colocará el auto i.

Puntaje:

- **0.5 ptos** por impresión de decisiones.
- **1 pto** por algoritmo con programación dinámica.
- No se da puntaje por soluciones recursivas.

c) El algoritmo es $O(L^2n)$, que es pseudo-polinomial ya que es exponencial en la representación binaria del input en base al largo: $O((2^{2x})(2^y))$ **Puntaje:**

- **1 pto** por correcto calculo de complejidad, siempre y cuando el algoritmo haya sido correcto.
- **1 pto** por explicación de si es exponencial, polinomial o pseudo polinomial.

3. *a*) (2/3) Muestre que existe un algoritmo cuyo tiempo de ejecución es $\Theta(n)$ para ordenar n números cuyas claves van entre 2^n y $2^n + \log_2 n$. Suponga que el algoritmo recibe dos parámetros: A, el arreglo en donde están los números, y n. Describa el algoritmo con detalle (pero no es necesario un pseudocódigo). ¿Cuánta memoria adicional necesita su algoritmo?

   b) (1/3) Modifique mínimamente el algoritmo de *a*) de manera que funcione correctamente cuando el rango de claves del input va entre 0 y $2^n + \log_2 n$. Analice el tiempo de ejecución de la mejor configuración que se le ocurra del algoritmo resultante bajo este nuevo escenario de inputs.

   - Criterio de corrección:
     a) El alumno menciona o describe (con palabras o código) un algoritmo que sirva para ordenar en $\Theta(n)$. Se consideran correctos *counting sort*, *radix sort*, *bucket sort* o algún otro algoritmo existente o creado que cumpla la complejidad (**1 punto**). Describe con detalle qué hace el algoritmo y cómo lo utiliza en este caso (**1 punto**) y explica correctamente su complejidad (**1 punto**). Entrega un valor correcto de cuánta memoria necesita su algoritmo (depende del algoritmo) (**1 punto**)
     b) El alumno modifica mínimamente el algoritmo y aún sirve para ordenar la lista **A**, dada sus características (**1 punto**). El alumno explica correctamente la complejidad de este nuevo algoritmo para su mejor caso (**1 punto**).
   - Ejemplo de posible solución:
     a) Una forma de ordenar la lista **A** de n elementos que pertenecen a un conjunto de numeros enteros entre 2^n y $2^n + log_2 n$, es utilizando una variación de *counting sort*. Utilizamos una lista de tamaño $log_2 n - 1$, ya que solo hay $log_2 n - 1$ valores posibles para los n numeros. Cada valor de esta lista (**B**) lo dejamos en 0. Luego hacemos los pasos de *counting sort* (*aquí hay que detallar qé hace el algoritmo (por pasos)*) con diferencia de que restamos 2^n en el cálculo para la obtención del índice en **B**. Este paso que se agrega es despreciable para la complejidad, por lo que se mantiene en $\Theta(n + k)$ dado por *counting sort*.
     Como $k = log_2 n$, entonces $k < n$ y $\Theta(n + k) \in \Theta(n)$. Por ende, el algoritmo es $\Theta(n)$. Como este algoritmo utiliza una lista de largo $log_2 n - 1$ y una lista de tamaño n en la que se insertan los valores ordenados, la memoria utilizada tiene complejidad $\Theta(n)$
     b) En caso de tener ahora valores entre 0 y $2^n + log_2 n$, solo bastaría modificar la lista **B** a un tamaño de $2^n + log_2 n - 1$. En este caso, ahora $k = 2^n + log_2 n$. Por ende, al recorrer la lista **B** para hacer las respectivas sumas en cada indice, se requieren $\Theta(2^n)$ pasos. Ahora la complejidad, en su mejor caso es $\Theta(2^n)$, ya que en cualquier caso deberá recorrer toda la lista **B**.

4. Las siguientes preguntas son en el contexto del problema de la Tarea 1.

a) Discuta el posible impacto en el tiempo de ejecución de usar la siguiente poda: *"Si en algún momento alguna de las celdas queda sin posibles opciones de asignación, entonces se debe retornar (o retroceder)"*
Respuesta:
(1 pt) La poda es correcta ya que se deben completar todas las celdas para encontrar la solución.
(1 pt) En cuanto al aporte de la poda, logra detectar casos irresolubles en niveles bajos de la búsqueda, lo que elimina mucha ramificación del problema y disminuye considerablemente los pasos que realiza.
(1 pt) El costo de calcular la poda es $O(n)$ en cada paso, donde n es la cantidad de celdas sin piezas, ya que debe revisar para todas las celdad si no tiene opciones de asignación.

b) ¿Cuál de las siguientes heurísticas cree que es más efectiva? Justifique.

- Al asignar una pieza a una celda, revisar si alguna otra celda queda con una sola opción. De ser así, asignarla de inmediato, como parte de este mismo paso y repetir este proceso, de ser posible.
- Al elegir qué celda asignar, preferir la que tiene menos opciones de asignación.

Respuesta:
(1 pt) El costo de calcular ambas heurísticas es el mismo, por lo que su efectividad solo depende de su beneficio.
(2 pts) Cualquiera de la siguientes respuestas se considera correcta:

- Ambas heurísticas son correctas y ayudan a disminuir la cantidad de pasos, pero la segunda engloba a la primera, por lo que es mejor.
- La segunda heurística contiene a la primera, pero existe la posibilidad de que la segunda heurística lleve a que se hagan más pasos en algún puzle especísfico, mientras que la primera heurística solo puede hacer más rápida la búsqueda. Por lo tanto la primera es mejor.
- La primera heurística hace que el árbol de búsqueda tenga una altura menor, mientras que la segunda heurística ayuda a disminuir la ramificación del problema, por lo que ambas son muy efectivas (ambas son igual de buenas).