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

IIC 2133 — Estructuras de Datos y Algoritmos
Pauta Interrogación 1
Segundo Semestre, 2014

Duración: 2 hrs.

1. **(Colas)** La implementación de colas en arreglos vista en clases usa un arreglo Q y los indicadores $head[Q]$ y $tail[Q]$ para ayudar al manejo de la cola.

   Suponga que el sistema operativo incurre en un *overhead* de T_m unidades de tiempo cada vez que se pide memoria, independiente de la cantidad de memoria que se pida. (T_m, en la práctica, es mucho mayor que el tiempo que demora una asignación o una comparación.) Proponga una implementación de colas, lo más simple que pueda, que requiera pedir memoria sólo después de k o más operaciones de EnQueue, donde k es un parámetro. Específicamente, dé un pseudocódigo para las operaciones:

   a) (1/6) $\mathrm{Init}(Q, k)$: que inicializa la cola Q con el parámetro k.

   Respuesta: Lo que haremos sera hacer una especie de lista ligada de arreglos en donde el ultimo elemento del arreglo, de posición $k+1$, sea un puntero hacia el siguiente arreglo, si es que este existe.

```
function INIT(Q, k)
    Q ← array(k + 1)
    Q.length ← k
    head[Q] ← 1
    tail[Q] ← 1
    check ← false
```

   b) (3/6) $\mathrm{EnQueue}(Q, x)$: agrega un elemento x al final de la cola.

   Respuesta:

```
function ENQUEUE(Q, x)
    if tail[Q] ≤ Q.length then
        tail[Q] ← tail[Q] + 1
        Q[tail[Q]] ← x
    else
        if Q.check = false then
            Q' ← newCola()
            Init(Q', k)                    ▷ Usamos la función Init
            Q[tail[Q]] ← Q'                ▷ Asignamos a la última posición de Q el siguiente arreglo
            Q'[tail[Q']] ← x
            tail[Q'] ← tail[Q'] + 1
            Q.check ← true
        else
            Q' ← Q[tail[Q]]
```

```
            Enqueue(Q', x)
```

c) (2/6) $\mathrm{DeQueue}(Q)$: retorna y elimina el elemento al principio de la cola si ésta no está vacía y un error si lo está.

Respuesta:

```
function DEQUEUE(Q)
    if head[Q] = tail[Q] and Q.check = false then
        return error
    else
        if Q.check = false and tail[Q] > head[Q] then
            aux ← Q[head[Q]]
            head[Q] ← head[Q] + 1
            return aux
        if Q.check = true and tail[Q] = head[Q] then
            Q' ← Q[tail[Q]]
            return Dequeue(Q')
```

2. Desafortunadamente, un arreglo que contiene elementos con un misma clave repetida muchas veces lleva a QuickSort a desempeñarse cercano a su peor caso. En el extremo, con un arreglo con un único valor para la clave de ordenación, el algoritmo es cuadrático.

a) Muestre cómo modificar $\mathrm{Partition}(A, p, r)$ visto en clases **sin agregar** un *keyword* de iteración adicional[1], de tal forma de evitar malos casos con muchas claves repetidas.

Respuesta: Una modificación correcta es la que se describe a continuación.

```
function PARTITION(A, p, r)
    x ← A[r]
    i ← p − 1
    mid ← ⌊(p + r)/2⌋
    for j ← p to mid do
        if A[j] ≤ x then
            i ← i + 1
            SWAP(A[i], A[j])
    for j ← mid + 1 to r − 1 do
        if A[j] < x then
            i ← i + 1
            SWAP(A[i], A[j])
    SWAP(A[i + 1], A[r])
    return i + 1
```

Existen muchas otras soluciones correctas.

b) Argumente por qué su algoritmo resuelve el problema y por qué es correcto.

Respuesta: Este algoritmo resuelve el problema ya que cuando hay muchas claves repetidas, tenderá a hacer particiones más equitativas que el algoritmo tradicional.

[1] Esta restricción tiene sentido hacerla por eficiencia.

Consideremos el caso en que todas las claves son iguales. Cuando particionamos un arreglo de tamaño n, el algoritmo visto en clases lo separará en arreglos de tamaño 1 y $n-1$. Esto causará que QuickSort tome tiempo en $\Theta(n^2)$.

El algoritmo presentado, en cambio, particionará tal arreglo en particiones de tamaño $\lceil n/2 \rceil$ y $\lfloor n/2 \rfloor$. En este caso, QuickSort tomará tiempo en $\Theta(n \log n)$.

3. Un *heap flojo* es uno en donde $A[Parent(Parent(i))] \geq A[i]$ cuando $i \geq 4$, y tal que $A[Parent(i)] \geq A[i]$, cuando $1 \leq i < 4$. Así, se cumple que todo heap también es un heap flojo, pero no al revés.

   a) Dé un pseudocódigo para la inserción de un elemento en un heap flojo y argumente que es correcta y más eficiente en la práctica que la misma operación en heaps tradicionales.

   Respuesta: Una posible solución es la siguiente.

   function INSERT(A, n)
       $A.heap\text{-}size \leftarrow A.heap\text{-}size + 1$
       $A[heap\text{-}size] \leftarrow n$
       SIFTUP($heap\text{-}size$)

   function SIFTUP(A, i)
       if $i = 1$ **then return** ▷ Llegamos a la raíz
       $next \leftarrow Parent(Parent(i))$
       if $next < 1$ **then** ▷ Este nodo no tiene abuelo
           $next \leftarrow Parent(i)$
       if $A[next] > A[i]$ **then**
           $swap(A[next], A[i])$
           SIFTUP($next$)

   El pseudocódigo anterior preservará la propiedad del *heap flojo* de forma similar que un heap tradicional. Es más eficiente ya que realizará $\lceil \lfloor \lg n \rfloor / 2 \rceil$ llamadas recursivas en lugar de $\lfloor \lg n \rfloor$.

   b) Explique en detalle (no es necesario un pseudocódigo) cómo se debe implementar la operación *SiftDown*. ¿Es esta operación más eficiente que su homónima en heaps? Justifique.

   Respuesta: La implementación de *SiftDown* seguirá la misma idea que la implementación en (a). Sin embargo, hay que considerar que cuando hacemos *SiftDown* en la raíz, tenemos que considerar tanto a los hijos como los nietos. Es decir, deberemos comparar la raíz con sus 2 hijos y 4 nietos y intercambiarla con el máximo de todos. Luego, continuamos recursivamente comparando el nodo que cambiamos con sus 4 nietos e intercambiándolos hasta que no sea necesario o alcancemos el final del heap.

   Dado lo anterior, no es claro si esta versión de *SiftDown* será más eficiente que la tradicional, ya que a pesar de que realizaremos aproximadamente la mitad de las llamadas recursivas, en cada una de ellas debemos realizar cuatro comparaciones en lugar de dos.

4. *a*) (2/3) Deduzca una cota lo más ajustada posible (usando notación Θ) para el tiempo de ejecución de un algoritmo cuyo tiempo de ejecución promedio está dado por

$$T(n) = \begin{cases} 1 & \text{cuando } n = 1 \\ n^2 + \frac{2}{n} \sum_{i=2}^{n-1} T(i) & \text{cuando } n > 1 \end{cases}$$

Respuesta: Debemos calcular el Θ de la ecuación de recurrencia del algoritmo, para esto debemos calcular el Ω del algoritmo y el O del algoritmo y demostrar que estas cotas son iguales. Primero calculamos la cota inferior. Vemos que en la expresión,

$$n^2 + \frac{2}{n}\sum_{i=2}^{n-1} T(i)$$

esta acotado inferiormente por n^2 ya que la función $T(n)$ es siempre positiva, luego su suma es positiva. Por lo tanto, podemos decir que:

$$n^2 \leq T(n)$$

La expresión anterior hace que el algoritmo tenga un $\Omega(n^2)$. Ahora debemos calcular una cota asintótica superior para el algoritmo. Para encontrar una cota superior haremos lo siguiente:

$$nT(n) = n^3 + 2\sum_{i=2}^{n-1} T(i)$$

Si evaluamos en $n-1$ obtenemos lo siguiente.

$$(n-1)T(n-1) = (n-1)^3 + 2\sum_{i=2}^{n-2} T(i)$$

. Si restamos estas dos ecuaciones obtenemos:

$$nT(n) - (n-1)T(n-1) = 3n^2 - 3n + 1 + 2T(n-1)$$

Si sumamos $(n-1)T(n-1)$ y dividimos por n a ambos lados obtenemos:

$$T(n) = 3n - 3 + \frac{1}{n} + \frac{(n+1)}{n}T(n-1)$$

La cual es una ecuacion de la forma

$$T(n) = \Theta(n) + cT(n-1)$$

En particular si $c = 1$, la ecuación tiene por solución

$$T(n) = c_1 + \frac{1}{2}n(n+1)$$

La cúal es $O(n^2)$. Por lo tanto como

$$\Omega(n^2) \leq T(n) \leq O(n^2)$$

Entonces

$$T(n) \in \Theta(n^2)$$

Es posible resolver este problema por el método de sustitución o el método iterativo, pero sus resultados se dejan como ejercicio para el lector.

b) (1/3) ¿Es verdadero que el tiempo de ejecución de *bubble sort*, en el mejor caso es $\Omega(n^2)$? Argumente usando el pseudocódigo del algoritmo.

Respuesta: Falso, en clases se vió una modificación del algoritmo que permite que el mejor caso sea $\Omega(n)$.

function BSORT(arr, n)
    $changed \leftarrow true$
    while $i < n$ and $changed = true$ **do**
        $changed \leftarrow false$
        for $(j = 1; j \geq i; j - -)$ **do**
            if $arr[j] < arr[j-1]$ **then**
                $swap(arr, j, j-1)$
                $changed \leftarrow false$
        $i \leftarrow i + 1$