Algoritmos y Estructura de datos III - Año 2018 – Sección TQ – 1er Semestre
Prof. Cristian Cappo **- Ordenación interna – Tarea #6 –** ***A entregar el día Viernes 20-abril-2018 23:55 - TP=65***
Marzo-2018

Ejercicio 1 (5p)
Dado un conjunto de ***n*** puntos en el plano, el punto (x_i,y_i) domina al punto (x_j, y_j) si $x_i > x_j$ e $y_i > y_j$. Un punto máximo es un punto que no es dominado por ningún otro punto en el conjunto.

Diseñe un algoritmo ***O(n log n)*** en Java que encuentre todos los puntos máximos. Fundamente el tiempo solicitado. Incluya un archivo de puntos de prueba.

Ejercicio 2 (10p)
Considere la siguiente implementación de QuickSort:

```
public static void extranhoQuicksort(double[] a) {
        int N = a.length;                // Nro. de elementos
        int hi = 0;                      // Nro. de elemenos mayores
        int lo = 0;                      // Nro. de elementos  menores
        double[] H = new double[N];      // Elementos mayores
        double[] L = new double[N];      // Elementos menores

        for (int i = 1; i < N; i++) { // Partición en a[0]
             if (a[i] > a[0])
               H[hi++] = a[i]; // Mayores
             else
               L[lo++] = a[i]; // Menores
        }
        if (lo > 0) { // Si no es vacio
              double[] temp = new double[lo-1];                    // Redimensionar
              for (int i = 0; i < lo-1; i++) temp[i] = L[i];       // Copiar elementos menores
              extranhoQuicksort(temp);                             // Ordenar elementos menores
              a[lo] = a[0];                                                // Partición
              for (int i = 0; i < lo-1; i++) a[i] = temp[i];       // Copiar
        }
        if (hi > 0) {
              double[] temp = new double[hi-1];
              for (int i = 0; i < hi-1; i++) temp[i] = H[i];
              extranhoQuicksort(temp);
              for (int i = 0; i < hi-1; i++) a[lo+1+i] = temp[i];
        }
}
```

Critique esta versión de Quicksort según las siguientes consideraciones:
a) ¿El algoritmo es correcto? es decir, ¿Ordena? Si no ordena, indique porque (2p)
b) Análisis asintótico para el peor caso y el caso medio. (4p)
c) Uso de memoria para el peor caso y el caso medio. (4p)

Ejercicio 3 (15p)
a) Dado el código de RadixSort en SL presentado aquí:

- Mencione que restricciones impone esta implementación (en SL). *(2p)*
- Indique la *T(n)* y la *O* de este algoritmo en el peor caso. *(1p)*

```
sub  RadixSort(ref V:vector[*] numerico )
tipos
     colaitem : registro {
                elems : vector[1500] numerico
                cont  : numerico
     }
var
  i,j,k,h,digito :numerico
  iter           :numerico
  continuar      :logico
  colas          : vector[10] colaitem
inicio
  iter = 1
  repetir
     colas = {{{0,...},1},...} /* Inicializar colas */
     continuar = NO
     desde i=1 hasta alen(V) {
        continuar = ( int(V[i]/iter)  > 10 ) or continuar
        digito    = int(V[i]/iter)  % 10 + 1
        colas[digito].elems[colas[digito].cont] = V[i]
        inc(colas[digito].cont)
     }
     iter = iter * 10
     j = 1
     desde i=1 hasta 10 {
        h = colas[i].cont -1
        desde k=1 hasta h {
            V[j] = colas[i].elems[k]
            inc(j)
        }
     }
  hasta ( not continuar )
fin
```

b) Implemente en Java un algoritmo RadixSort que pueda ordenar cadenas alfabéticas (base 26). (5p)
c) En que caso el algoritmo RadixSort puede tardar más que un tiempo lineal. (3p)
d) Implemente una versión de *RadixSort* similar al mostrado en la diapositiva de clase utilizando el algoritmo countingSort, también mostrado en clase, como el algoritmo a ser utilizado por RadixSort para la ordenación en cada dígito. ¿Qué ventaja poseería sobre la versión de RadixSort presentado en el item a) ? *(5p)*

Ejercicio 4 (10p)

a) Defina el concepto de estabilidad de un algoritmo de ordenación. Luego indique claramente su utilidad práctica. *(2p)*

b) Indique si estos algoritmos de ordenación son estables o inestables: InsertSort, MergeSort, HeapSort, QuickSort, BubleSort, ShellSort, RadixSort. *(3p)*

c) Considere el algoritmo de ordenación por selección presentado aquí en SL para el arreglo *A[1..n]*:

```
sub SELECTSORT( ref A: vector[*] numerico )
var
  i, j, min : numerico
inicio
  desde i=1 hasta alen(A) - 1 {
      min = i
      desde j=i+1 hasta alen(A) {
          si ( A[j] < A[min] ) {
               min = j
          }
      }
      intercambiar(A[i],A[min])
  }
fin
```

c.1) ¿Es este algoritmo estable o inestable? ¿Porqué? *(2p)*

c.2) Si es inestable, indique exactamente la forma de cambiarlo para volverlo estable. Justifique su respuesta. *(3p)*

Ejercicio 5 (15p)

El algoritmo de *QuickSort* presentado en clase contiene dos llamadas recursivas como se muestra en el código de abajo :

```
QUICKSORT ( A, p, r)

if p < r do
      q ← PARTITION (A,p,r)
      QUICKSORT(A,p,q-1)
      QUICKSORT(A,q+1,r)

Llamada inicial: QUICKSORT ( A, 1, n)
```

La segunda llamada no es realmente necesaria, puede ser evitada utilizando una estructura iterativa. Esta técnica, denominada Recursión de Cola (*Tail Recursion*), es proveída muchas veces por algunos buenos compiladores. Es utilizada básicamente para evitar desbordamiento de pilas y hacer más eficiente el cómputo. Considere el siguiente código de quicksort que simula la recursión de cola.

```
QUICKSORT_RC( A, p, r)

while p <= r do
      q ← PARTITION (A,p,r)
      QUICKSORT_RC(A,p,q-1)
      p ← q + 1

Llamada inicial: QUICKSORT_RC( A, 1, n)
```

a) Muestre que el algoritmo `QUICKSORT_RC(A,1,n)` ordena correctamente el arreglo `A[1..n]`.

Los compiladores usualmente ejecutan las funciones utilizando una pila que contiene la información necesaria, incluyendo los valores de los parámetros, en cada llamada recursiva. La información de la llamada más reciente esta en el tope de la pila, y la información de la llamada inicial en la base de la pila. Cuando una función o procedimiento es invocado entonces la información es apilada y cuando termina la misma es desapilada de la pila de ejecución. Asumiendo que los parámetros que son arreglos son representados como punteros (como en C o Java), la información requerida en cada llamada requiere un espacio proporcional a O(1). La profundidad de la pila es la máxima cantidad de espacio utilizado durante el cómputo.

b) Describa un escenario en el cual la profundidad de la pila de `QUICKSORT_RC` es $\Theta(n)$ para un arreglo de n elementos.

c) Modifique el código de `QUICKSORT_RC` tal que en el peor caso la profundidad de la pila sea $\Theta(\log n)$, manteniendo el tiempo esperado en $O(n \log n)$.

Ejercicio 6 (10p)

Considere un arreglo *a* de N elementos comparables $a_0, a_1, a_2, .., a_{N-1}$. Se dice que un arreglo ***b*** es circular de ***a*** si este consiste en el subarreglo $a_k, a_{k+1}, .., a_{N-1}$ seguido del subarreglo $a_0, a_1, a_2, ... , a_{k-1}$ para algún entero *k*. En el ejemplo de abajo, ***b*** es circular de ***a*** con $k=7$ y $N=10$:

arreglo ordenado ***a*** *[ ]*

0	1	2	3	4	5	6	7	8	9
1	2	3	5	6	8	9	34	55	89

arreglo circular ***b*** *[ ]*

0	1	2	3	4	5	6	7	8	9
34	55	89	1	2	3	5	6	8	9

Suponga que usted tiene como entrada un arreglo circular ***b*** de algún arreglo ordenado (no tiene el valor de *k* ni el arreglo ordenado). Asuma que el arreglo ***b*** consiste en N elementos comparables, todos diferentes.

Diseñar una función **genérica** en **Java** que reciba un arreglo circular ***b*** de elementos comparables y un valor ***t*** de tipo genérico y retorne si ***t*** existe o no en el arreglo.

El algoritmo debe estar en $O(\log N)$ en el peor caso. N es el tamaño del arreglo. *Fundamentar el tiempo*.