Facultad Politécnica – Ingeniería Informática
Algoritmos y Estructura de Datos III – 2025 – 2do. Periodo – Sección TQ/TR

Tarea 4-U3 - 35p

Trabajo a entregar por GRUPO vía EDUCA. Indique claramente los integrantes de su grupo. El código debe poder probarse sin usar ningún IDE desde la línea de comandos. No incluya ninguna clase en package.

Ejercicio 1 (10p) (Puede responder en un archivo de texto)

a) Definir el concepto de estabilidad de un algoritmo de ordenación. Luego indique claramente su utilidad práctica. Indicar referencia.
b) Indicar si estos algoritmos de ordenación son estables o inestables: InsertSort, MergeSort, HeapSort, QuickSort, BubleSort, ShellSort, RadixSort
c) Considere el siguiente algoritmo de ordenación el arreglo A de **n** elementos:

```
public static <T extends Comparable<T>> void ordenar( T [] A ) {
  int n = A.length;
  for ( int i = 0 ; i < n ; i++) {
      int min = i;
      for ( int j=i+1; j < n; j++) {
          if ( A[j].compareTo(A[min]) < 0 ) {
               min = j;
          }
      }
      //intercambia A[i] con A[min]
      T tmp  = A[i];
      A[i]   = A[min];
      A[min] = tmp;
  }
}
```

c.1) ¿Es este algoritmo estable o inestable? Fundamente.
c.2) Si es inestable, indique exactamente la forma de cambiarlo para volverlo estable. Justificar la respuesta.

Rúbrica	Puntaje
a) Conceptualización correcta y utilidad práctica	2
b) Indicar si cada algoritmo es estable o no.	4
c.1) Es el algoritmo dado estable o no, fundamentación	2
c.2) Si es inestable, indicar forma de cambiarlo	2

Ejercicio 2 (10p)

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)
```

Las respuestas a las preguntas a y b pueden dejarse en el mismo archivo del código del ítem c.

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 está 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 $O(n \log n)$.

Rúbrica	Puntaje
Respuesta correcta para (a)	3
Respuesta correcta para (b)	2
Respuesta correcta para (c)	5

Ejercicio 3 (15p)

Las respuestas a las preguntas se pueden dejar como comentario en el propio código (mencionado en el punto c) indicando a qué ítem corresponde la respuesta.

a) Dado el código de `RadixSort` en el lenguaje SL (https://www.cnc.una.py/sl/) presentado aquí:
   a.1) Mencione que restricciones impone esta implementación (en SL).
   a.2) Indique la $T(n)$ y la O de este algoritmo en el peor caso.

```
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) basado el algoritmo de a).

c) ¿En qué caso el algoritmo *RadixSort* puede tardar más que un tiempo lineal? Muestre un ejemplo.

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 ítem a) ?

Rúbrica	Puntaje
Respuesta Correcta para (a.1)	2
Respuesta Correcta para (a.2)	1
Implementación correcta en (b)	3
Comentarios aclaratorios en (b)	2
Respuesta correcta en (c)	2
Implementación correcta en (d)	3
Comentarios aclaratorios en (d) y respuesta correcta a la pregunta.	2