Grado en Ingeniería Informática

Estructura de Datos y Algoritmos, Grupo 81M, 2015/2016

16 de Marzo de 2016

Nombre y Apellidos:

…….

**PROBLEMA 1 (1 punto)**

Dadas las clases:

```
public class SNodeInteger {

    public Integer elem;
    public SNodeInteger next;

    public SNodeInteger(Integer e) {
        elem = e;
    }

}
```

```
public class SListInteger implements IListInteger {

    public SNodeInteger first;

    ....

}
```

donde **IListInteger** es la especificación del TAD Lista de números enteros.

Se pide:

a) Añade un método en la clase SListInteger que reciba un número entero **x** como parámetro. El objetivo del método es borrar de la lista todos los elementos cuyo valor es múltiplo de **x**. Algunos ejemplos son:

| lista | Operación | lista |
|---|---|---|
| 5 4 3 10 15 2 5 | lista.removeMultiples(5) | 4 3 2 |
| 3 1 2 8 | lista.removeMultiples(5) | 3 1 2 8 |
| 3 1 3 5 8 | lista.removeMultiples(1) | |
| 3 1 3 5 8 | lista.removeMultiples(2) | 3 1 3 5 |

b) Razona sobre el caso peor y mejor del algoritmo. ¿Cuál es la complejidad del removeMultiples?.

c) Si la lista fuera una lista doblemente enlazada, ¿cuál sería la complejidad del método removeMultiples?

**Nota:** La solución no requiere que uséis los métodos de la clase IListInteger. Sin embargo, si decides utilizar alguno de sus métodos (por ejemplo, isEmpty()), deberás añadir su implementación en la solución.
En la solución no puedes usar arrays, ArrayList, LinkedList.

**Solución:**

```
/**
 * This method traverses the lists
 * removing those nodes
 * whose elem is multiple of x,
 * that is, elem % x == 0
 * @param x
 */
public void removeMultiples(int x) {
    SNodeInteger previousNode=null;
    for (SNodeInteger nodeIt=first; nodeIt!=null; nodeIt=nodeIt.next) {
        if (nodeIt.elem % x ==0) {
            if (previousNode == null) {
                first = nodeIt.next;
            } else {
                previousNode.next = nodeIt.next;
            }
        } else previousNode=nodeIt;
    }

}
```

b) El mejor caso se da cuando la lista está vacía (O(1)). No hay diferencia entre el peor caso y el caso promedio ya que siempre es necesario recorrer la lista completa. La complejidad del método es lineal (O(n)).

c) La complejidad del método seguiría siendo lineal (O(n)). En realidad, la implementación usando listas doblemente enlazada no proporciona ninguna ventaja ya que el método requiere que la lista se recorra de extremo a extremo.

**PROBLEMA 2 (1 punto)**

El siguiente método iterativo devuelve el elemento más pequeño de un array de enteros:

```
public static int findMinIte(int data[]) {
    if (data == null)
        throw new NullPointerException("Can't handle null arrays");

    if (data.length == 0)
        throw new IllegalArgumentException("Can't handle zero-length arrays.");


   int min=data[0];
   for (int i=1; i<=data.length-1;i++) {
       if (data[i]<min) {
           min=data[i];
       }
   }
   return min;

}
```

a) Razona sobre la complejidad temporal del algoritmo. Razona también sobre el caso promedio y el peor caso.

**Respuesta:** La complejidad del método es lineal (O(n)) ya que es necesario recorrer todo el array para obtener el mínimo del array.
No existe diferencia entre el caso promedio y el caso peor, ya que siempre es necesario recorrer toda el array para obtener el mínimo. El mejor caso es cuando la lista está vacía.

b) Desarrolla un **método RECURSIVO** que dado un array de enteros devuelva el elemento más pequeño del array.
**Nota:** Las soluciones iterativas (basadas en bucles) no serán evaluadas.
**Nota:** Por simplificar, puedes suponer que el array siempre tendrá datos. Dicho de otra forma, no tienes que preocuparte de comprobar si el array es null o su longitud es 0.

**Solución:**

```java
public static int findMin(int[] data, int left, int right) {

    //base case
    if (left==right) return data[left];


    //we calculate the position in the middle
    //This positions allows us to divide the problem into two subproblems
    int m=(left + right) / 2;

    int minLeft=findMin(data,left,m);    //both subproblems are separate
    int minRight=findMin(data,m+1,right);


    //finally, we combine the solutions of the subproblems
    //in order to obtain the global solution
    if (minLeft<minRight) return minLeft;
    else return minRight;
}
```