Ejercicio Parcial Mediciones

E3. Sorting (30 pts)

Se cuenta con un sistema de seguimiento de contaminación ambiental que cubre toda la ciudad de Buenos Aires.

Así, regularmente se registra información de diferentes sensores en diferentes momentos y se quiere saber en qué zonas se registra mayor contaminación, ya que cada sensor está asociado a una zona particular de la urbe.

De esta forma, dados ciertos registros realizados, que constan del identificador del sensor, que es un valor alfanumérico de máximo 64 caracteres, el momento en que se tomó el registro y el valor medido, donde ambos son naturales no acotados, se quiere obtener el acumulado de cada sensor para las últimas k mediciones (se puede considerar cualquier valor de k, en tanto se tiene una gran cantidad de registros por cada sensor). Con esta información se desea hacer un ranking, donde aparezcan primero los sensores con mayor contaminación registrada acumulada. En caso de empate, se mostrarán primero los sensores con registros más recientes.

Se quiere implementar la función MÁSCONTAMINADOS, que recibe un arreglo que contiene mediciones de los sensores y la cantidad de mediciones que se desean considerar para obtener un acumulado, y se desea obtener un *ranking*, donde primero aparecen los sensores que tienen mayor valor total:

```
MásContaminados(in a: Array<struct<sensor: string, in t:int, in v:int>>, in k: int): Array<string>
```

La función debe tener cota de peor caso $O(n.\log n)$, siendo n la cantidad de mediciones registradas entre todos los sensores.

Por ejemplo, dados el A indicado abajo y k igual a 2, `MásContaminados(A, 2)` retornará `[48A, 1AB, 1C]`, ya que tanto el sensor `1AB` como el `48A` registran un acumulado de 120 en las últimas 2 mediciones, pero el `48A` tiene un registro más reciente, por lo que queda primero, y luego queda el `1C`, que sólo acumula 55.

```
A = [ (1AB, 8, 100), (48A, 10, 100), (1AB, 9, 25), (48A, 9, 25),
    (1AB, 14, 95), (48A, 15, 20), (1C, 12, 5), (1C, 17, 50) ]
```

a) Se pide escribir el algoritmo de `MásContaminados`, justificando **detalladamente** la complejidad.
b) ¿Cuál sería el *mejor caso* para este algoritmo? ¿Cuál sería la cota de complejidad más ajustada?

Figura 1: Enunciado Mediciones

SensorId = *string*
Sensor = Struct⟨sensor: *SensorId*, t: *int*, v: *int*⟩
Medicion = Struct⟨sensorId: *SensorId*, t: *int*, valorAcmulado: *int*, mediciones: *int*⟩

proc MasContaminados (in A: Array<Sensor>, in k: int) : Array<SensorId>

```
mergeSort(A) // Ordenar por mas recientes "t" mayor a menor. O(n*log(n))

// El diccionario digital funciona en O(1)
// por el largo maximo es de 64 caracteres
// y el alfabeto está acotado.
var mediciones: DiccionarioDigital<SensorId, Medicion>
var sensores: ListaEnlazada<SensorId>
    mediciones:= diccionarioVacio()
    sensores:= listaVacia()

var j: int
    j:= 0

while (j < A.length) do // O(n)
  var sensorId: SensorId
      sensorId:= A[j].sensor

  if (!mediciones.esta(sensorId)) then
    // Nos quedamos con la fecha mas reciente
    mediciones.definir( // O(1)
      sensorId,
      new Medicion(
        sensorId=sensorId,
        mediciones=1
        t=A[j].t
        valorAcumlado=A[j].v,
      ) // O(1)
      sensores.agregarAtras(sensorId)
    )
  else
    var medicion: Medicion
        medicion:= sensores.obtener(sensorId) // O(1)

    if (medicion.mediciones < k) then
      medicion.mediciones:= medicion.mediciones + 1 // O(1)
      medicion.valorAcumulado:= medicion.valorAcumulado + A[j].v // O(1)
      // Nos vamos quedando con la fecha mas reciente dentro de las "k" mediciones
      medicion.t:= A[j].t // O(1)
    endif
  endif
endwhile

// Ahora que tenemos el diccionario armado, con los ultimos "k" valores
// acumulados, solo tenemos que pasarlo a un array y ordenarlo primero
// por el desempate (t menor a mayor) y luego por valorAcmulado (mayor a menor)
var resultadoMedicionesLista: ListaEnlazada<Medicion>
var resultadoMediciones: Array<Medicion>

while !(sensores.vacia()) do // O(n)
  resultadoMedicionesLista.agregarAtras(mediciones.obtener(sensores.primero())) // O(1)
  sensores.fin() // O(1)
endwhile

resultadoMediciones:= toArray(resultadoMediciones) // O(n)

mergeSort(resultadoMediciones) // ordenar por "t" menor a mayor O(n*log(n))
mergeSort(resultadoMediciones) // ordenar por "valorAcumulado" mayor a menor O(n*log(n))

var res: Array<SensorId>
    res:= new Array(resultadoMediciones.length) // O(n)
    j:= 0 // O(1)

while (j < res.length) do // O(n)
  res[j]:= resultadoMediciones[j].sensorId
endwhile

return res
/**
* Analisis complejidad: O(n*log(n))
*/
```

Ejercicio 14

Para poder resolver este problema con la complejidad que nos piden vamos a tener que utilizar una combinacion entre heap sort y listas enlazadas.
Fijémonos por qué merge no sirve:
Supongamos que tenemos el arreglo [2,3,4] y k=2
Si pudieramos armar el siguiente arreglo en n*k: [2, 4, 3, 6, 4, 8]
Nos damos cuenta que podríamos ir haciendo merge de a partes.
Paso 1: merge([2,4], [3,6]) $--> $ [2,3,4,6]
Paso 2: merge([2,3,4,6], [4, 8]) $--> $ [2,3,4,4,6,8]
Esto no funciona, la complejidad de hacer merge en n listas ordenadas es cuadratica.
(es una sumatoria de gauss)
Así que: Heap sort al rescat. Vamos a usar listas enlazadas para que remover el primero sea O(1)

`proc ordenarMultiplos` (in A: `Array<int>`, in k: int) : `Array<int>`

```
var res: Array<int>
    res:= new Array(A.length * k) // O(n*k)

var arregloDeListas: Array<ListaEnlazada<int>>
    arregloDeListas:= new Array(A.length)
var j: int
var i: int
    i:= 1
    j:= 0

while (j < A.length) do
  arregloDeListas[j] = listaVacia()

  while (i <= k) do
    arregloDeListas[j].agregarAtras(A[j] * i)
    i:= i + 1
  endwhile

  i:= 1
  j:= j + 1
endwhile

// Cola de prioridad min, por el primer elemento de la lista enlazada
var colaDePrioridad: ColaDePrioridadLog<ListaEnlazada<int>>
    colaDePrioridad:= colaDePriodidadVacia()
    j:= 0

while (j < arregloDeListas.length) do
  colaDePrioridad.encolar(arregloDeListas[j])
  j:= j + 1
endwhile

var res: Array<int>
var indiceActual: int
    res:= new Array(A.length * k)
    indiceActual:= 0

while (!colaDePriodidad.estaVacia())
  var listaPrioritaria: ListaEnlazada<int>
      listaPrioritaria:= colaDePriodidad.desencolarMax()

  res[i]:= listaEnlazada(listaPrioritaria.primero())

  listaPrioritaria.fin()
  colaDePriodidad.encolar(listaPrioritaria)

  i:= i + 1
endwhile

return res
```

Ejercicio 17

`Planta` $= string$
`Stock` $= int$
`Prioridad` $= int$
`Nodo` $=$ `Struct`$\langle$`planta:` $Planta$`, stock:` $Stock$`, prioridad:` $Prioridad\rangle$

```
proc Recolectar (
    in s: Array<Tupla<Planta, Stock>>,
    in u: DiccionarioDigital<Planta, Prioridad>
) : Array<Planta>
```

```
var stockTotal: DiccionarioDigital<Planta, Stock>
    stockTotal:= diccionarioVacio()

// Inicializamos stockTotal
var j:= 0
while (j < s.length) do
  var planta: Planta
  var stock: Stock
  var stockAcumulado: Stock
      planta:= s[j][0]
      stock:= s[j][1]
      stockAcumulado:= 0

  if (stockTotal.pertenece(planta)) then
      stockAcumulado:= stockTotal.obtener(planta)
  endif

  stockTotal.definir(Planta, stockAcumulado + stock)
  j:= j + 1
endwhile

// Evitemos a toda costa iterar sobre un dict
// justificar que se amortiza es siempre complicado
// TRUCAZO
var nodos: ListaEnlazada<Nodo>
    nodos:= listaVacia()
    j:= 0

while (j < s.length) do
  var planta: Planta
      planta:= s[j][0]
  if (stockTotal.pertenece(planta)) then
    var prioridad: Prioridad
    var stock: Stock
        prioridad:= u.obtener(planta)
        stock:= stockTotal.obtener(planta)

    nodos.agregarAtras(new Nodo(planta=planta, stock=stock, prioridad=prioridad))
    stockTotal.borrar(planta)
  endif
  j+= j + 1
endwhile

var arreglo_nodos: Array<Nodo>
    arreglo_nodos:= toArray(nodos)

mergeSort(arreglo_nodos) // ordenar de menor a mayor por stock
mergeSort(arreglo_nodos) // ordenar de mayor a menor por prioridad


var res: Array<Planta>
    res:= new Array(arreglo_nodos.longitud())
    j:=0

while (j < arreglo_nodos.longitud()) do
  res[j]:= arreglo_nodos[j].planta
  j:= j + 1
endwhile

return res
```

Ejercicio 18

`Libreta` $= int$
`Nota` $= int$
`Promedio` $= float$
`Contador` $= totalNotas : int, notasAcumuladas : Nota$

`proc OrdenarPorLibretaYPromedios` (in s: `Array<Tupla<Libreta, Nota>>`) : `Array<Tupla<Libreta, Promedio>>`

```
var promedios: DiccionarioDigital<Libreta, Contador>
promedios:= diccionarioVacio()

var j: int
    j:= 0

// Inicializamos el dict.
while (j < s.length) do
  var libreta: Libreta
      libreta:= s[j][0]
  promedios.definir(
    libreta,
    new Contador(s[j], new Contador(totalNotas=0, notasAcumuladas=0))
  )
  j += 1
endwhile

// Insertamos las notas acumuladas.
j:= 0
while (j < s.length) do
  var libreta: Libreta, nota: Nota
      libreta:= s[j][0], nota:= s[j][1]

  var contador: contador
      contador:= promedios.obtener(libreta)

  contador.totalNotas:= contador.totalNotas + 1
  contador.notasAcumuladas:= nota

  // No hace falta re-definir, hay aliasing
  j += 1
endwhile

// Creamos res
var res: Array<Tupla<Libreta, Promedio>>
    res:= new Array<promedios.tamaño()

// Vamos a zafar de iterar promedios, nunca queremos
// iterar y tener que justificar que las operaciones
// se amortizan, TRUCAZO:
var i: int
    i:= 0
    j:= 0
while (j < s.length) do
  var notaAcumulada: Nota
  var libreta: Libreta
      nota:= s[j][1]
      libreta:= s[j][0]

  if (promedios.pertenece(libreta))
    var contador: contador
        contador:= promedios.obtener(libreta)

    res[i]:= <libreta, contador.notasAcumuladas / contador.totalNotas>
    promedios.borrar(libreta)

    i:= i + 1
  endif

  j:= j + 1
endwhile

mergeSort(res) // ordenar res por la primera coordenada: libreta. (menor a mayor)
mergeSort(res) // ordenar res por la segunda coordenada: promedio. (mayor a menor)

return res
```

Ejercicio 19

`Prioridad` $= int$
`Nodo` $=$ `Struct`$\langle$`prioridad:` $Prioridad$, `valor:` $int\rangle$

`proc OrdenarSegunCriterio` (in s: `Array<int>`, in crit: `Array<Prioridad>`) : `Array<int>`

```
var s1: ListaEnlazada<Nodo>
var s2: ListaEnlazada<int>
var critDict: DiccionarioLog<int, Prioridad>
    s1:= listaVacia()
    s2:= listaVacia()
    critDict:= diccionarioVacio()

var prioridad: int
    prioridad:= 0

// Inidicalizamos crit dict.
while (prioridad < crit.length) do
  critDict.definir(crit[prioridad], prioridad)
  prioridad:= prioridad + 1
endwhile

var j: int
    j:= 0

// Insertamos en las listas s1 y s2.
while (j < s.length) do
  if critDict.pertenece(s[j]) then
    s1.agregarAtras(s[j], new Nodo(prioridad=critDict.obtener(s[j]), valor=s[j])
  else
    s2.agregarAtras(s[j])
  endif
endwhile

// Ordenar la primera parte del arreglo
var s1Arreglo: Array<Nodo>
var s1Valores: Array<int>
    s1Arreglo:= toArray(s1)
    s1Valores:= new Array(s1Arreglo.length)

mergeSort(s1Arreglo) // ordenar s1Arreglo por prioridad.
// No hay criterio de desempate pues crit no tiene repetidos

j:= 0
while (j < s1Arreglo.length) do
  s1Valores[j]:= s1Arreglo[j].valor
  j:= j + 1
endwhile

// Ordenar la segunda parte del arrego
var s2Valores: Array<int>
    s2Valores:= toArray(s2)

mergeSort(s2Valores) // ordenar s2Arreglo de menor a mayor

// Concatenar en res.
var res: Array<int>
    res:= concat(s1Valores, s2Valores)

// Devolver res.
return res
```