Ordenamiento

Algoritmos y Estructuras de Datos 2

Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

9 de Octubre de 2015

Funciones auxiliares

```
unirListasDeArreglo(in A: arreglo(lista(alumno)), out B : arreglo(alumno))
  lista(α) B ← Vacía()                                              O(1)
  for i ← 0 to tam(A − 1) do
    unirListas(B, A[i])                                             O(|A[i]|)
  end for
```

Complejidad del algoritmo: $O(1) + O(|A[0]|) + O(|A[1]|) + ... + O(|A[9]|) =$

$O(1) + \sum_{i=0}^{9} O(|A[i]|) = \sum_{i=0}^{9} O(|A[i]|) =$ **O(n)** (Como mucho hay n hombres o n mujeres)

```
unirListas(inout IZQ: lista(α), inout DER : lista(α) )
  for i ← 0 to Longitud(DER − 1) do
    AgregarAtras(IZQ, Copiar(A[0]))                                 O(copy(α))
    Fin(DER)                                                        O(1)
  end for
```

Complejidad del algoritmo: $O(|DER| * |\alpha|)$

```
deListaAArreglo(inout A: lista(α), out B : arreglo(α) )
  arreglo(α) B ← crearArreglo(Longitud(A))                          O(|A|)
  for i ← 0 to Longitud(A − 1) do
    B[i] ← A[0]                                                     O(copy(α))
    Fin(A)                                                          O(1)
  end for
```

Complejidad del algoritmo: $O(|A| * |\alpha|)$

Primer ejercicio

Considere la siguiente estructura para guardar las notas de un alumno de un curso:

alumno es tupla⟨nombre: string, sexo: FM, puntaje: Nota⟩
donde FM es $enum\{masc, fem\}$ y Nota es un nat no mayor que 10.

Se necesita ordenar un arreglo(alumno) de forma tal que todas las mujeres aparezcan al inicio de la tabla según un orden creciente de notas y todos los varones aparezcan al final de la tabla también ordenados de manera creciente respecto de su puntaje, como muestra en el siguiente ejemplo:

Entrada

Ana	*F*	10
Juan	*M*	6
Rita	*F*	6
Paula	*F*	7
Jose	*M*	7
Pedro	*M*	8

Salida

Rita	*F*	6
Paula	*F*	7
Ana	*F*	10
Juan	*M*	6
Jose	*M*	7
Pedro	*M*	8

- Proponer un algoritmo de ordenamiento ordenaPlanilla(inout p: arreglo(alumno)) para resolver el problema descripto anteriormente y cuya complejidad temporal sea O(n) en el peor caso, donde n es la cantidad de elementos del arreglo. Justificar.

```
ordenaPlanilla(inout A: arreglo(alumno))
    arreglo(lista(alumno)) Femenino ← crearArreglo(10)                          O(10) = O(1)
    arreglo(lista(alumno)) Masculino ← crearArreglo(10)                         O(10) = O(1)
    for i ← 0 to 9 do
        Femenino[i] ← Vacía()                                                   O(1)
        Masculino[i] ← Vacía()                                                  O(1)
    end for
    for i ← 0 to tam(A − 1) do
        if A[i].sexo = F then AgregarAtras(Femenino[A[i].nota − 1], A[i])       O(1)
        else AgregarAtras(Masculino[A[i].nota − 1], A[i]) end if                O(1)
    end for
    lista(alumno) Mujeres ← unirListasDeArreglo(Femenino)                       O(n)
    lista(alumno) Hombres ← unirListasDeArreglo(Masculino)                      O(n)
    A ← deListaAArreglo(unirListas(Mujeres, Hombres))                           O(n)
```

Complejidad del algoritmo: $O(n) + O(n) + O(n) + O(n) = O(n)$ donde $n = |A|$.

Segundo ejercicio

Se desea ordenar los datos generados por un sensor industrial que monitorea la presencia de una sustancia en un proceso químico. Cada una de estas mediciones es un número entero positivo. Dada la naturaleza del proceso se sabe que, dada una secuencia de n mediciones, a lo sumo $\lfloor \sqrt{n} \rfloor$ valores están fuera del rango $[20, 40]$.

Proponer un algoritmo $O(n)$ que permita ordenar ascendentemente una secuencia de mediciones y justificar la complejidad del algoritmo propuesto.

Opción 1

```
ordenarDatos(inout A: arreglo(nat))
    lista(nat) menoresA41 ← Vacía()                                   O(1)
    lista(nat) mayoresA40 ← Vacía()                                   O(1)
    for i ← 0 to tam(A − 1) do
      if A[i] < 41 then AgregarAtras(menoresA41, A[i])                O(1)
      else AgregarAtras(mayoresA40, A[i]) end if                      O(1)
    end for
    arreglo(nat) arrayX ← deListaAArreglo(menoresA41)                 O(n) = O(|menoresA41|)
    arreglo(nat) arrayY ← deListaAArreglo(mayoresA40)                 O(√n) = O(|mayoresA40|)
    CountingSort(arrayX)                                              O(n) = O(|arrayX| + 40) = O(|menoresA41| + 40))
    InsertionSort(arrayY)                                             O(n) = O(√|arrayY|)²)
    contenar(A, arrayX, arrayY)                                       O(n) = O(n + √n)
```

Como los números menores a 41 son a lo sumo n y los mayores o iguales a 41 son a lo sumo $\sqrt{n}$, $arrayX$ y X son de tamaño a lo sumo n (y su máximo **siempre** es 40), mientras que $arrayY$ y Y son de tamaño a lo sumo $\sqrt{n}$. CountingSort, InsertionSort y merge son los mismos algoritmos especificados en el módulo de algoritmos básicos.

CountingSort lo puedo aplicar a $arrayX$ porque su máximo siempre es 41 (y su mínimo siempre 1), lo que hace que ordenarlo cueste $O(n)$. InsertionSort resulta costar $O(n)$ porque el tamaño de $arrayY$ es a lo sumo $\sqrt{n}$, que elevado al cuadrado es n. Aclaración: $n = |A|$

Opción 2

```
ordenarDatos(inout A: arreglo(nat))
    lista(nat) enRango ← Vacía()                                      O(1)
    lista(nat) mayoresA40 ← Vacía()                                   O(1)
    lista(nat) menoresA20 ← Vacía()                                   O(1)
    for i ← 0 to tam(A − 1) do
      if A[i] < 41 and A[i] > 19 then AgregarAtras(enRango, A[i]) else  O(1)
      if A[i] > 40 then AgregarAtras(mayoresA40, A[i]) else AgregarAtras(menoresA20, A[i])  O(1)
      end if                                                          O(1)
    end for
    arreglo(nat) arrayZ ← deListaAArreglo(enRango)                    O(n) = O(|enRango|)
    arreglo(nat) arrayX ← deListaAArreglo(menoresA20)                 O(n) = O(|menoresA20|)
    arreglo(nat) arrayY ← deListaAArreglo(mayoresA40)                 O(√n) = O(|mayoresA40|)
    CountingSort(arrayZ)                                              O(n) = O(|arrayZ| + 20) = O(|enRango| + 20))
    InsertionSort(arrayX)                                             O(n) = O(√|arrayX|)²)
    InsertionSort(arrayY)                                             O(n) = O(√|arrayY|)²)
    contenar(A, arrayX, arrayZ, arrayY)                               O(n) = O(n + √n)
```

Tercer ejercicio

Se tienen k arreglos de naturales $A_0, ..., A_{k-1}$. Cada uno de ellos está ordenado de forma creciente. Se sabe que ningún natural está dos veces en el mismo arreglo y no aparece el mismo natural en dos arreglos distintos. Además, se sabe que para todo i el arreglo A_i tiene exactamente 2^i elementos.

Dar un algoritmo que devuelva un arreglo B de $n = \sum_{i=0}^{k-1} 2^i = 2^k - 1$ elementos, ordenado crecientemente, de manera que un natural está en B si y sólo si está en algún A_i (o sea, B es la unión de los A_i y está ordenado). El algoritmo debe tener complejidad $O(n)$, donde $n = |A|$

```
granMerge(in A: arreglo(arreglo(nat)), out B : arreglo(nat) )
    arreglo(nat) B ← crearArreglo(0)                                   O(1)
    for i ← 0 to tam(A − 1) do
      merge(B, B, A)                          O(2^i) = O(2^i + (2^i − 1)) = O(tam(B) + tam(A))
    end for
```

El merge utilizado es el especificado en el apunte de algoritmos básicos. El algoritmo tiene complejidad

$$O(n) = O(2^k - 1) = O(\sum_{i=0}^{k-1} 2^i) = \sum_{i=0}^{k-1} O(2^i)$$

que es la suma de las complejidades de cada iésima iteración, calculadas según la complejidad que tiene el merge del apunte.