Asmad Murga Victor Manuel
LU: 760/19
--------------------------Ejercicio de Ordenamiento--------------------------
Aclaraciones dadas durante el parcial por los docentes:
La gravedad de un paciente es un Real (i.e., puede tener cualquier cantidad de decimales, por default)
Comparar dos apellidos que tiene costo 0(1)
Copiar los datos de una persona y su gravedad es O(1)
Comparar dos relaes es O(1)

Se que tengo n/log(n) valores con mas de 2 decimales
Tambien se que, como el arreglo tiene n valores. Tengo n - n/log(n) valores con menos de dos decimales

Me **importa** el orden de los apellidos para la estabilidad, entonces pueso hacer algo
Armo un arreglo de triplas, donde cada tripla sea una copia de las tuplas que vienen en el arreglo
Asumo que vienen tuplas como ⟨ gravedad, apellido ⟩
Entonces, otra tupla que será del estilo (⟨ gravedad, apellido ⟩, indice)
De forma que el indice va aumentando en 1 cada vez que paso
Asi mantengo estable. Como me dijeron que copiar los datos de una persona es O(1) entonces no tengo problema armarla
De esta manera puedo hacer la siguiente idea

-> Se que mi tengo n/log n personas con gravedad de mas de 2 decimales
-> Por ende, tengo n - n/log n personas con gravedad de menos de dos decimales

Puedo armar dos buckets uno donde esten las personas con gravedad de menos de dos decimales y otro donde
esten las personas con mas de dos decimales
Como tengo la funcion tieneDosDecimales(num) la utilizo para saber cuales tienen dos decimales y cuales no, en ese caso
divido las cosas internamente en los buckets

Se que los buckets tienen m valores siendo m < n
En el caso del bucket de menos decimales, esto es que tiene n - n/log (n) valores
Bueno, fuera de eso, se que tengo una cantidad acotada de cifras, a lo sumo 4 pues tengo 10.0
Luego, puedo usar un radixSort para ordenar
LA complejidad de RadixSort es O(d*n) siendo la cantidad de los digitos a ordenar
Como estos digitos estan acotados por 3, entones tengo O(3*n) = O(n)
Finalmente, tengo ordenado el primer bucket

Ahora, vamos por el segundo bucket. Este es mas complicado porque tengo muchos numeros con decimales indeterminados
Se como mucho que comparar dos reales tiene coso O(1) por las observaciones
Se tambien que tengo n/log n valores con mas de dos decimales. Esto lo puedo usar para el calculo de la complejidad
del algoritmo pues, sea s = n/log n. Con s < n pues no puedo tener mas pacientes con gravedad que los que tengo registrados
(Por lo menos en el registro, no puedo tener 500 pacientes con gravedad si solo me registro el sistema 300 como ejemplo)
Entonces, necesito un algoritmo que ordene en O(s) como mucho, pues esta es la cantidad de personas que tengo

No puedo usar radixSort porque mi cantidad de decimales no está acotada. Pero, puedo usar una idea
Supongamos que ordeno con un algoritmo de complejidad logatirmica*lineal
Puedo usar mergesort, y ahora muestro el porque

Esto debe ser O(n/log(n) * (log(n/log(n))))
Quiero ver que O(todo eso) ∈ O(n)

Por temas de tiempo no puedo terminar la demostracion
Me ahorro aplicar la explciacion del teorema entonces esto
me queda como n/log(n) * log(n/log(n)) ≤ c*n
Como n > 0 "simplifico"
Luego tengo 1/log(n) * log(n/log(n)) = 1/log(n) * (log(n) - log(log(n)))
Simplifico queda 1 - log(log(n))/log(n) que eso es por una cota (agresiva) menor que n
Luego tengo 1 - n ≤ c
1 - c ≤ n
Tomo c = 1, entonces 0 ≤ n ∀ n

Finalmente, esto me queda perteneciendo a O(n)

Entonces, puedo usar un mergeSort, para poder ordenar el segundo bucket. de forma que tenga el orden en O(n)

Una vez completado esto, tengo 2 buckets. uno con n - n/log(n) valores y otro con n/log(n) valores
Si sumo estos tengo n valores
COmo tengo dos buckets, solo me falta la parte del merge. Entonces, puedo hacer un merge de todos estos valores en un arreglo
final a devolver y es aca donde entra en juego la tupla armada, porque yo al insertar un valor en el merge, como esta ordenado, fijo los indices
De esta forma, se cual indice es menor
Ej: ⟨ (9.5, Alonzo), 5 ⟩, ⟨ (9.5, Alfonso), 2 ⟩ primero ira alfonso y luego alonzo por indices
Entonces, puedo asi mantener una estabilidad en el algoritmo
Finalmente, la complejidad del Conquistar es O(n)

Entonces tengo:
Costo de crear 2 buckets O(n) *// Asumo como peor caso*
Costo de crear el arreglo con indices O(n) (pues uso una lista para agregar al final siempre en costo O(1) por copia de datos es O(1)
Ordenar el bucket de hasta 2 decimales es O(n)
Ordenar el bucket de mas de 2 decimales es O(n)
Hacer el concat es O(n)
Finalmente, la complejidad de todo el algoritmo es O(n)

```
// ---- PSEUDOCODIGO
ordenarArreglo(in A: arreglo( ⟨ gravedad: ℝ, apellido: string⟩) ) -> res: arreglo( ⟨ gravedad: ℝ, apellido: string⟩)
  res <- vacio(tam(A))
  lista <- vacio()
  i <- 0
  for persona in A do
    agregarAtras(⟨persona, i⟩)
    i <- i + 1
  end for

  personas2Decimales <- vacio()
  personasMas2Decimales <- vacio()

  for persona in lista do
    if TieneDosDecimales(π₁(π₁(persona))) then
      agregarAtras(personas2Decimales, persona)
    else
      agregarAtras(personasMas2Decimales, persona)
    endif
  end for

  // Se pasa por referencia modificable para que no tenga que copiar
  RadixSort(personas2Decimales)
  MergeSort(personasMas2Decimales)

  // Con el criterio del menor indice va primero
  concat(personas2Decimales, personasMas2Decimales)
  i <- 0
  for p in lista do
    res[i] = π₁(p)
    i <- i + 1
  end for
  return res
end function

El algoritmo tendria una pinta asi. Complejidad dada mas arriba

b) Si tenemos en cuenta la tupla que cree inicialmente, con ⟨ datoPersona, indice ⟩
    Entonces solo hay que ver el indice. Si el indice es menor, es porque va segundo, ya que en el
    arreglo original vienen los apellidos de menor a mayor. Entonces si quiero de mayor a menor, solo veo
    el valor del indice e ira primero el que sea mayor numero
    La complejidad no cambia pues ya está incluida en este caso. Ver y comparar ese indice es O(1)
    Esta idea se aplica en la funcion concat

c) Una vez ordenado, yo no se el orden anterior, pero se los apellidos. Puedo aplicar la idea del punto a
   Armo un arreglo con indices como tercer item de cada tupla, entonces, ahora armo un bucket
   Bucket => O(n)
   Estos son varios buckets donde yo voy a meter a todas las personas, especialmente aquellas que tengan
   la misma gravedad
   Pero ahora el criterio de ordenamiento se hace sobre los que tengan la misma gravedad, sobre el apellido
   Como tengo un indice, lo unico que hago es invertir este indice, esto lo puedo hacer en O(n) con algun algoritmo conocido
   (Ingresa Cocktail Sort (recuerdos de Algo 1)) Toma el primer elemento lo manda al final, el del final lo trae al primer
   elemento el que le sigue lo manda al ante ultimo y así. Es como un insertion sort solo que yo ya se donde tienen que ir los nuevos elementos)

   Ahora se que esta ordenado de menor a mayor, todo gracias al indice, y me ahorra tener que comparar los apellidos (aunque me dijeran que es O(1))
   Esto tambien lo asumo porque inicialmente la lista, en el item a entraba con apellidos de menor a mayor, al devolverla
   estaran de mayor a menor por item b del ejercicio, luego quiero de menor a mayor otra vez, el orden solo se invierte
   Se mantiene estable el algoritmo
   Finalmente, cuando tengo ordenado, hago un concat y esto será en O(n)
   Este concat toma como criterio unir todos los buckets y el orden será por la gravedad y ,cuando llegue a un bucket ordenado por apellido, simplemente voy
   insertando los valores en el orden nuevo.
   Tambien, para evitar que las cosas se desordenen, busco este indice, por cada bucket me fijo si el ultimo elemento de ese bucket coincide con el primero
   del siguiente bucket. Si es asi sigo concatenand, si no es asi busco el ultimo elemento, como coinciden, entonces concateno de inicio a final

   La complejidad final es O(n)
```