Sorting

Algoritmos y Estructuras de Datos II

Jonathan Seijo

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

4 de junio de 2018

Sorting

Sobre resolver ejercicios

- Usar algoritmos ya conocidos.
- Prestar atención a las características del problema.
- Justificar todas las complejidades (Importante!).
- Practicar (y consultar dudas!).

Algoritmos conocidos - I

- Insertion Sort, $O(n^2)$

  Mantiene una parte ordenada y va insertando cada nuevo elemento en el lugar correspondiente. Son n elementos e insertar ordenado cuesta $O(n)$

Algoritmos conocidos - I

- Insertion Sort, $O(n^2)$

  Mantiene una parte ordenada y va insertando cada nuevo elemento en el lugar correspondiente. Son n elementos e insertar ordenado cuesta $O(n)$

- Selection Sort, $O(n^2)$

  Busca el mínimo elemento entre una posición i y el final, lo intercambia con el elemento de la posición i e incrementa i. Buscar el menor cuesta $O(n)$ y son n elementos.

Algoritmos conocidos - I

- Insertion Sort, $O(n^2)$

  Mantiene una parte ordenada y va insertando cada nuevo elemento en el lugar correspondiente. Son n elementos e insertar ordenado cuesta $O(n)$

- Selection Sort, $O(n^2)$

  Busca el mínimo elemento entre una posición i y el final, lo intercambia con el elemento de la posición i e incrementa i. Buscar el menor cuesta $O(n)$ y son n elementos.

- Bubble Sort, $O(n^2)$

  Itera n veces, comparando dos elementos contiguos y swapeando si no respeta el orden. En cada iteración, el máximo elemento *burbujea* hacia el final.

Algoritmos conocidos - II

- MergeSort, $O(n \log n)$

  Divide la entrada en dos partes iguales y ordena recursivamente las dos mitades. Luego, las combina de forma ordenada. (*Merge*)

Algoritmos conocidos - II

- MergeSort, $O(n \log n)$

  Divide la entrada en dos partes iguales y ordena recursivamente las dos mitades. Luego, las combina de forma ordenada. (*Merge*)

- QuickSort, $O(n^2)$ (En caso promedio: $O(n \log n)$)

  Elige un pivot y separa los elementos entre los menores y los mayores a él. Luego, se ejecuta el mismo procedimiento sobre cada una de las partes.

Algoritmos conocidos - II

- MergeSort, $O(n \log n)$

  Divide la entrada en dos partes iguales y ordena recursivamente las dos mitades. Luego, las combina de forma ordenada. (*Merge*)
- QuickSort, $O(n^2)$ (En caso promedio: $O(n \log n)$)

  Elige un pivot y separa los elementos entre los menores y los mayores a él. Luego, se ejecuta el mismo procedimiento sobre cada una de las partes.
- AVL-Sort, $O(n \log n)$

  Insertar los elementos en un AVL, $O(\log n)$ cada inserción. Quitar el mínimo en $O(\log n)$ hasta terminar con todos los elementos.

Algoritmos conocidos - II

- MergeSort, $O(n \log n)$

  Divide la entrada en dos partes iguales y ordena recursivamente las dos mitades. Luego, las combina de forma ordenada. (*Merge*)

- QuickSort, $O(n^2)$ (En caso promedio: $O(n \log n)$)

  Elige un pivot y separa los elementos entre los menores y los mayores a él. Luego, se ejecuta el mismo procedimiento sobre cada una de las partes.

- AVL-Sort, $O(n \log n)$

  Insertar los elementos en un AVL, $O(\log n)$ cada inserción. Quitar el mínimo en $O(\log n)$ hasta terminar con todos los elementos.

- HeapSort, $O(n + n \log n)) = O(n \log n)$

  Armar el Heap en $O(n)$ y quitar cada uno de los elementos ordenados en $O(\log n)$.

Definiciones

- Sorting inplace

  Utiliza $O(1)$ espacio adicional.

Definiciones

- Sorting inplace

  Utiliza $O(1)$ espacio adicional.

- Sorting estable

  Mantiene el orden relativo entre los elementos.

Definiciones

- Sorting inplace

  Utiliza $O(1)$ espacio adicional.

- Sorting estable

  Mantiene el orden relativo entre los elementos.

  Todo sorting de comparación puede hacerse estable a costo de memoria adicional (Usar una tupla <Elem, PosOriginal>)

Cotas inferiores

Teorema

Todo algoritmo de sorting basado en comparaciones requiere $\Omega(n \log n)$ comparaciones en peor caso.

Counting Sort

K representa el valor máximo del arreglo.

Counting-Sort(A, k)

```
1:  B = Arreglo(k+1) — Creo un arreglo de tamaño K                    ▷ O(k)
2:  for i = 0 to k do — Lleno el arreglo con ceros                    ▷ O(k)
3:      B[i] ← 0

4:  for j = 0 to length[A]-1 do — Cuento apariciones de cada elemento ▷ O(n)
5:      B[A[j]] ← B[A[j]] + 1

6:  indexA ← 0
7:  for i = 0 to k do — Recorro todo elemento de A (estan entre 0 y k) ▷ O(n + k)
8:      for j = 0 to B[i]-1 do — Inserto la cantidad de apariciones
9:          A[indexA] ← i
10:         indexA++
```

Counting Sort

K representa el valor máximo del arreglo.

Counting-Sort(A, k)

```
1:  B = Arreglo(k+1) — Creo un arreglo de tamaño K                    ▷ O(k)
2:  for i = 0 to k do — Lleno el arreglo con ceros                    ▷ O(k)
3:      B[i] ← 0

4:  for j = 0 to length[A]-1 do — Cuento apariciones de cada elemento ▷ O(n)
5:      B[A[j]] ← B[A[j]] + 1

6:  indexA ← 0
7:  for i = 0 to k do — Recorro todo elemento de A (estan entre 0 y k) ▷ O(n + k)
8:      for j = 0 to B[i]-1 do — Inserto la cantidad de apariciones
9:          A[indexA] ← i
10:         indexA++
```

- Complejidad: O(n+k) con $n = length(A)$

Counting Sort

- Counting Sort tiene complejidad lineal, ¿*Rompe* el teorema?

Counting Sort

- Counting Sort tiene complejidad lineal, ¿*Rompe* el teorema?

  No. Counting-Sort no está basado en comparaciones. Nunca compara de a pares, usamos información extra (cota superior de K).

Counting Sort

- Counting Sort **puede implementarse estable!**
- La versión que vimos se la conoce también como *Rapid-Sort*

Ejercicio

Practica 5 - Ejercicio 12

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, para 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.

Solución

Ejercicio12(A: Arreglo(nat))

```
1: losEnRango ← BuscarEnRango(A)                          ▷ O(n)
2: losFueraDeRango ← BuscarFueraDeRango(A)                ▷ O(n)

3: Counting-Sort(losEnRango, 40)                          ▷ O(n+40) = O(n)
4: Insertion-Sort(losFueraDeRango)                        ▷ O(tam^2) = O((√n)^2) = O(n)

5: A ← Merge(losEnRango, losFueraDeRango)                 ▷ O(n)
```

- ¿Por qué el Insertion-Sort costó O(n)?
  Porque la secuencia tenía como mucho $\sqrt{n}$ elementos.

Ejercicio 2

Se tiene información sobre las materias que actualmente se dictan en la facultad en un arreglo con la estructuras de la forma:

$$\text{info: tupla} \langle \textit{materia}\text{: string}, \textit{turno}\text{: MN}, \textit{aula}\text{: nat} \rangle$$

- Donde MN es enum{mañana, noche}
- *aula* es un nat ≤ 300.
- El string de la materia es acotado.

Ejercicio 2

Se tiene información sobre las materias que actualmente se dictan en la facultad en un arreglo con la estructuras de la forma:

$$\text{info: tupla} \langle \textit{materia}\text{: string}, \textit{turno}\text{: MN}, \textit{aula}\text{: nat} \rangle$$

- Donde MN es enum{mañana, noche}
- *aula* es un nat ≤ 300.
- El string de la materia es acotado.

Ejercicio 2

Se quiere ordenar en orden creciente de aulas, pero que primero aparezcan las de turno mañana y luego las de turno noche. Además, queremos mantener el orden relativo entre materias.

Siendo n la cantidad de elementos del arreglo, proponer un algoritmo que sea O(n) en peor caso.

Solución 2

Ejercicio(A : Arreglo(*info*))

```
1:  Arreglo(Lista(info)) deTurnoMañana ← CrearArreglo(300)        ▷ O(1)
2:  Arreglo(Lista(info)) deTurnoNoche  ← CrearArreglo(300)        ▷ O(1)

3:  for e in A do                                                 ▷ O(n)
4:      if e.turno = mañana then
5:          deTurnoMañana[ e.aula ].AgregarAtras( e )             ▷ O(1)
6:      else
7:          deTurnoNoche[ e.aula ].AgregarAtras( e )              ▷ O(1)

8:  mañanaOrdenada ← Rearmar(deTurnoMañana)                       ▷ O(n)
9:  nocheOrdenada ← Rearmar(deTurnoNoche)                         ▷ O(n)

10: A ← Concatenar(mañanaOrdenada, nocheOrdenada)                 ▷ O(n)
```

Solución 2

```
Ejercicio(A : Arreglo(info))
 1: Arreglo(Lista(info)) deTurnoMañana ← CrearArreglo(300)        ▷ O(1)
 2: Arreglo(Lista(info)) deTurnoNoche  ← CrearArreglo(300)        ▷ O(1)

 3: for e in A do                                                 ▷ O(n)
 4:     if e.turno = mañana then
 5:         deTurnoMañana[ e.aula ].AgregarAtras( e )            ▷ O(1)
 6:     else
 7:         deTurnoNoche[ e.aula ].AgregarAtras( e )             ▷ O(1)

 8: mañanaOrdenada ← Rearmar(deTurnoMañana)                       ▷ O(n)
 9: nocheOrdenada ← Rearmar(deTurnoNoche)                         ▷ O(n)

10: A ← Concatenar(mañanaOrdenada, nocheOrdenada)                 ▷ O(n)
```

- Complejidad: O(n)

Solución 2

```
Ejercicio(A : Arreglo(info))
 1: Arreglo(Lista(info)) deTurnoMañana ← CrearArreglo(300)        ▷ O(1)
 2: Arreglo(Lista(info)) deTurnoNoche  ← CrearArreglo(300)        ▷ O(1)

 3: for e in A do                                                 ▷ O(n)
 4:     if e.turno = mañana then
 5:         deTurnoMañana[ e.aula ].AgregarAtras( e )            ▷ O(1)
 6:     else
 7:         deTurnoNoche[ e.aula ].AgregarAtras( e )             ▷ O(1)

 8: mañanaOrdenada ← Rearmar(deTurnoMañana)                       ▷ O(n)
 9: nocheOrdenada ← Rearmar(deTurnoNoche)                         ▷ O(n)

10: A ← Concatenar(mañanaOrdenada, nocheOrdenada)                 ▷ O(n)
```

- Complejidad: O(n)
- Esto es CountingSort estable: al usar listas enlazadas estamos preservando el orden original de los elementos iguales.

Bucket Sort

- Clasificar los elementos en M clases (*buckets*).
- Ordenar cada bucket.
- Concatenar los resultados.

Bucket-Sort(A : Arreglo(nat), M : nat)

1: Arreglo(lista(nat)) B $\leftarrow$ CrearArreglo(M)
   // B es el arreglo que contiene los buckets

2: **for** i = 0 to n **do**
3: Insertar $A[i]$ en la lista $B[H(A[i])]$
   //*H manda* $A[i]$ *a su bucket correspondiente*.

4: **for** $j \leftarrow [0..M-1]$ **do**
5: Ordenar bucket $B[j]$

6: Concatenar buckets ya ordenados $B[0]$, $B[1]$, ..., $B[M-1]$

Bucket Sort - Complejidad

Complejidad:

- La complejidad depende de la cantidad de buckets, de cómo ordenemos y de cuántos elementos hay en cada bucket.
- Si todos los valores están acotados, podríamos usar counting sort en cada bucket y obtener complejidad lineal.

Bucket Sort - Complejidad

Complejidad:

- La complejidad depende de la cantidad de buckets, de cómo ordenemos y de cuántos elementos hay en cada bucket.
- Si todos los valores están acotados, podríamos usar counting sort en cada bucket y obtener complejidad lineal.
- Si los valores **no** están acotados, pero $M = O(n)$, las claves están **distribuidas uniformemente** entre los buckets y usamos InsertionSort en cada bucket, se puede demostrar que es $O(M + n)$. (*Demostración en el Cormen*)

Ejercicio 3

Tenemos todas las fechas de los finales de Análisis de los últimos 50 años, en tuplas de la forma:

$$\text{fecha: tupla} \langle \textit{dia} : \text{nat}, \textit{mes} : \text{nat}, \textit{año} : \text{nat} \rangle$$

Ejercicio 3

Queremos ordenar las fechas cronológicamente: según el año, en segundo lugar según el mes y por último según el día.
Si n es la cantidad de elementos del arreglo, ordenar las fechas en O(n) en peor caso.

Queremos ordenar *tuplas*
siguiendo una cierta prioridad de sus componentes,
sabiendo que todos los valores están acotados,
en tiempo **lineal**.

Radix Sort

A = Secuencia de tuplas
d = Cantidad de componentes de la tupla (o dígitos de un número)

Radix-Sort(A, d)

---

for $i \leftarrow [1..d]$ **do**
    Sort-Estable de A usando componente i

Radix Sort

A = Secuencia de tuplas
d = Cantidad de componentes de la tupla (o dígitos de un número)

Radix-Sort(A, d)

---

for $i \leftarrow [1..d]$ **do**
    Sort-Estable de A usando componente i

IMPORTANTE

- El Sorting tiene que ser **estable**.
- Ordenar desde la componente menos significativa a la más significativa, de otro modo estaríamos "arruinando" lo que ya está ordenado.
- Notar que podríamos querer cambiar el orden de iteración.

Radix Sort

- RadixSort nos sirve para ordenar cualquier estructura que pueda *interpretarse como tupla*, por ejemplo, números. Se puede pensar que cada dígito es la componente de una tupla.

Radix Sort

- RadixSort nos sirve para ordenar cualquier estructura que pueda *interpretarse como tupla*, por ejemplo, números. Se puede pensar que cada dígito es la componente de una tupla.
- La complejidad depende de la cantidad de componentes d (o "dígitos") y del algoritmos de sorting utilizado. Si utilizamos Counting-Sort, la complejidad es O(d*n).

Radix Sort

- RadixSort nos sirve para ordenar cualquier estructura que pueda *interpretarse como tupla*, por ejemplo, números. Se puede pensar que cada dígito es la componente de una tupla.
- La complejidad depende de la cantidad de componentes d (o "dígitos") y del algoritmos de sorting utilizado. Si utilizamos Counting-Sort, la complejidad es O(d*n).
- Si la cantidad de dígitos está acotada y usamos un algoritmo O(n) de ordenamiento, RadixSort es O(n).

Radix Sort

- RadixSort nos sirve para ordenar cualquier estructura que pueda *interpretarse como tupla*, por ejemplo, números. Se puede pensar que cada dígito es la componente de una tupla.
- La complejidad depende de la cantidad de componentes d (o "dígitos") y del algoritmos de sorting utilizado. Si utilizamos Counting-Sort, la complejidad es O(d*n).
- Si la cantidad de dígitos está acotada y usamos un algoritmo O(n) de ordenamiento, RadixSort es O(n).
- RadixSort es estable, porque ordena usando sortings estables.

Radix Sort

- RadixSort nos sirve para ordenar cualquier estructura que pueda *interpretarse como tupla*, por ejemplo, números. Se puede pensar que cada dígito es la componente de una tupla.
- La complejidad depende de la cantidad de componentes d (o "dígitos") y del algoritmos de sorting utilizado. Si utilizamos Counting-Sort, la complejidad es O(d*n).
- Si la cantidad de dígitos está acotada y usamos un algoritmo O(n) de ordenamiento, RadixSort es O(n).
- RadixSort es estable, porque ordena usando sortings estables.
- Al ordenar números usando sus dígitos, podría ser útil utilizar una base numérica que no sea base 2 o base 10 (*Pista para ejercicio 18*)

Solución - Ejercicio 3

- Tenemos tuplas donde los valores están acotados.
- Queremos que nuestra prioridad sea: Año, Mes, Día.
- Utilizar RadixSort, ordenando primero por día, luego por mes, y finalmente por año. (Recordar que es en orden inverso, para no arruinar lo más prioritario).

Ejercicio de parcial

1C - 2016 - Enunciado

Enunciado

Se tiene un arreglo de n colas de prioridad. Cada cola de prioridad almacena n números naturales. Supondremos que los números más chicos representan mayor prioridad (de manera que, por ejemplo, cada cola de prioridad podría estar representada con un *MinHeap*).

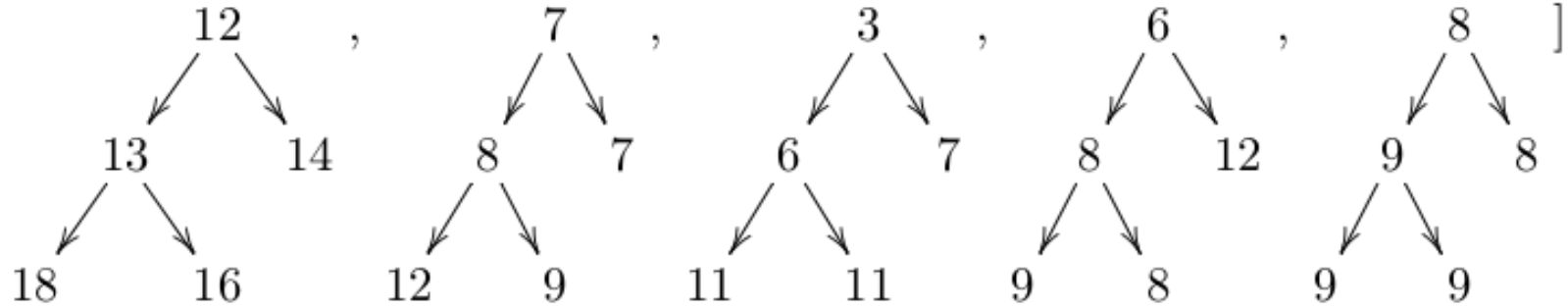
Se quiere calcular el arreglo ordenado de los n números más chicos que figuran en la totalidad de la estructura.

Diseñar un algoritmo con complejidad **O(n log n)** que calcule lo pedido.

másPrioritarios(in Q : arreglo(colaPrior(nat))) $\rightarrow$ res : arreglo(nat)

1C - 2016 - Enunciado

Ejemplo (con $n = 5$, mostrando las colas de prioridad como si fueran *MinHeaps*):

$Q = [$  $]$

$\text{másPrioritarios}(Q) = [3, 6, 6, 7, 7]$

1C - 2016 - Solución

másPrioritarios(in Q : Arreglo(ColaPrior(nat))) $\rightarrow$ res : Arreglo(nat)

```
1:  n ← tam(Q)                                                    ▷ O(1)
2:  res ← Arreglo(n)                                              ▷ O(n)
3:  primeros ← MinHeap()                                          ▷ O(1)

4:  for i = 0 to n do                                             ▷ O(n log n)
5:      minimo ← Q[i].Desencolar()                                  ▷ O(log n)
6:      primeros.Encolar(⟨ minimo, i ⟩)                             ▷ O(log n)

7:  for i = 0 to n do                                             ▷ O(n log n)
8:      minimo ← primeros.Desencolar()                              ▷ O(log n)
9:      proximoMinimo ← Q[ Π2(minimo) ].Desencolar()                ▷ O(log n)
10:     primeros.Encolar(⟨ proximoMinimo, Π2(minimo) ⟩)             ▷ O(log n)
11:     res[ i ] ← Π1(minimo)
```

Preguntas

Referencias y links útiles

- T. H. Cormen, C. E. Leiserson, R.L Rivest, and C. Stein. **"Introduction to Algorithms"**. 3rd edition.
  (También conocido como *CLRS* o *"El Cormen"* para los amigos)

- Clase de Erik Demaine, MIT.
  https://www.youtube.com/watch?v=Nz1KZXbghj8/

- Implementacion de `std::sort` de **GCC**.
  Advertencia: complicado de leer. Utiliza una mezcla de QuickSort, HeapSort e InsertionSort. Función principal en línea 1960.
  https://gcc.gnu.org/svn/gcc/trunk/libstdc++-v3/include/bits/stl_algo.h

- ¿Qué tan *malo* puede ser un algoritmo de sorting? Spoiler: muy malo.
  http://www.math.northwestern.edu/~mlerma/papers-and-preprints/inefficient_algorithms.pdf

:)