Ordenamiento - Sorting

Algoritmos y Estructuras de Datos

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

2° cuatrimestre 2024

Plan del día

- Repaso de los algoritmos de ordenamiento vistos en la teórica
- Repaso: Ejercicio 16 prática 9
- Ejercicio 6 práctica 9
- Ejercicio 19 práctica 9
- Ejercicio 18 prática 9

Un pequeño repaso

- Insertion Sort, $O(n^2)$ en el peor caso.

  En cada paso, el elemento se compara con los anteriores ya ordenados hasta encontrar su posición correcta. Son n elementos e insertar ordenado cuesta $O(n)$. Es **estable**.

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

  Busca el mínimo entre una posición i y el final de la lista intercambiando el mínimo con el elemento de la posición i. Buscar el menor cuesta $O(n)$ y son n. No es **estable**.

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

  Algoritmo recursivo que ordena sus dos mitades y luego las fusiona. Es **estable**

- QuickSort, $O(n^2)$ en el peor caso. $O(n \log n)$ en el caso promedio.

  Elige un pivote y separa los elementos entre los menores y los mayores a él. Y ejecuta el procedimiento sobre cada parte. Es de lo mejor en la práctica. No es **estable**.

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

  Arma el Heap en $O(n)$ y va sacando los elementos ordenados pagando $O(\log n)$. No es **estable**.

Repaso: Bucket Sort

- **Bucket Sort** ordena arreglos de cualquier tipo.
- Generalización de **Counting Sort**.
- Supone que los elementos pueden separarse (según algún criterio) en M categorías ordenadas.
- Es decir, para $i < j$, todo elemento de la categoría i es menor que todo elemento de la categoría j.
- Idea:

1. Construir un arreglo B de M listas y guardar los elementos de la categoría i en la i-ésima lista.
2. Ordenar las M listas por separado (si restara algo por ordenar).
3. Reconstruir el arreglo A, concatenando las listas en orden.

Repaso: Bucket Sort

- $h(x)$ es una función que indica la categoría de x (de 0 a $M-1$).
- Suponemos que se ejecuta en $O(1)$.

Algorithm 1 BUCKET-SORT(A, M)

$n \leftarrow length(A)$
$B \leftarrow$ arreglo de M listas vacías
for $i \leftarrow [0..n-1]$ **do**
    insertar $A[i]$ en la lista $B[h(A[i])]$
end for
for $j \leftarrow [0..M-1]$ **do**
    ordenar lista $B[j]$
end for
concatenar listas $B[0]$, $B[1]$, ..., $B[M-1]$

- ¿Complejidad? $O(n+M) + O(\text{ordenar buckets})$, con $n = length(A)$.
- Si se omite el paso 2... $O(n+M)$.

Repaso: Counting Sort

- **Counting Sort** asume que los elementos de un arreglo A de naturales son menores que un cierto valor k.

- Idea:

  1. Construir un arreglo B de tamaño k y guardar en la posición i, la cantidad de apariciones de i en A.

  2. Reconstruir el arreglo A, poniendo tantas copias de cada valor $j \in \{0..k\}$, como lo indique $B[j]$.

- **Observación:** Para un arreglo cualquiera de naturales, puede tomarse $k = \max_i\{A[i]\} + 1$.

- **Observación 2:** Se puede adaptar para el caso en que los valores están contenidos en un intervalo $[d, d + k)$.

Repaso: Counting Sort

Precondición: $k > \max_i\{A[i]\}$

Algorithm 2 COUNTING-SORT(A: arreglo(nat), k: nat)

```
1:  B ← arreglo de tamaño k
2:  for i ← [0..k − 1] do
3:      B[i] ← 0
4:  end for
5:  //cuento la cantidad de apariciones de cada elemento.
6:  for j ← [0..length(A) − 1] do
7:      B[A[j]] ← B[A[j]] + 1
8:  end for
9:  //inserto en A la cantidad de apariciones de cada elemento.
10: indexA ← 0
11: for i ← [0..k − 1] do
12:     for j ← [1..B[i]] do
13:         A[indexA] ← i
14:         indexA ← indexA + 1
15:     end for
16: end for
```

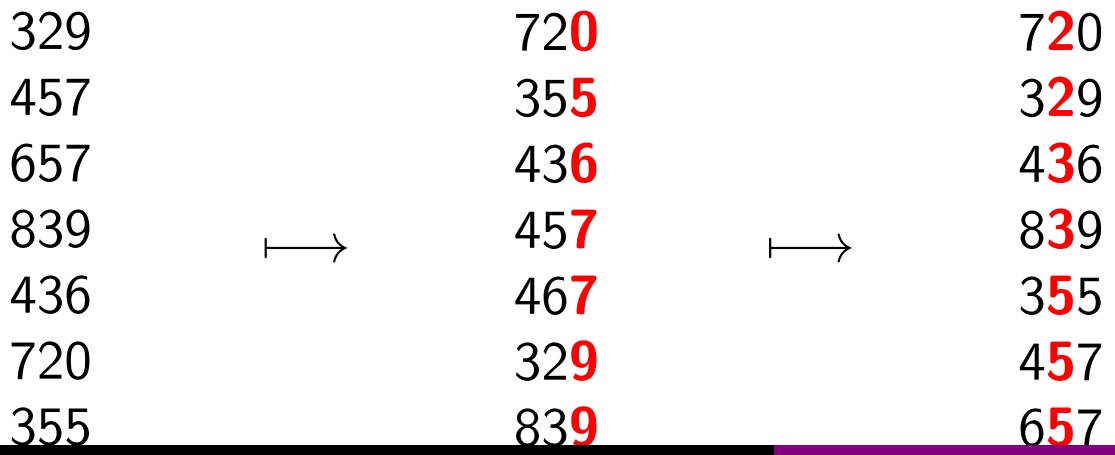
- ¿Complejidad? $O(n + k)$, con $n = length(A)$
- En el Cormen hay otra versión (más complicada de entender pero más fácil de analizar su complejidad)

Repaso: Radix Sort (versión para ordenar naturales)

- **Radix Sort**, en su versión más básica, ordena una lista de naturales.
- **Idea:**
  1. Ordenar los valores mirando sólo el dígito 1
  2. Ordenar los valores mirando sólo el dígito 2 (manteniendo el orden en caso de empate)
  3. Ordenar los valores mirando sólo el dígito 3 (manteniendo el orden en caso de empate)
  4. ...

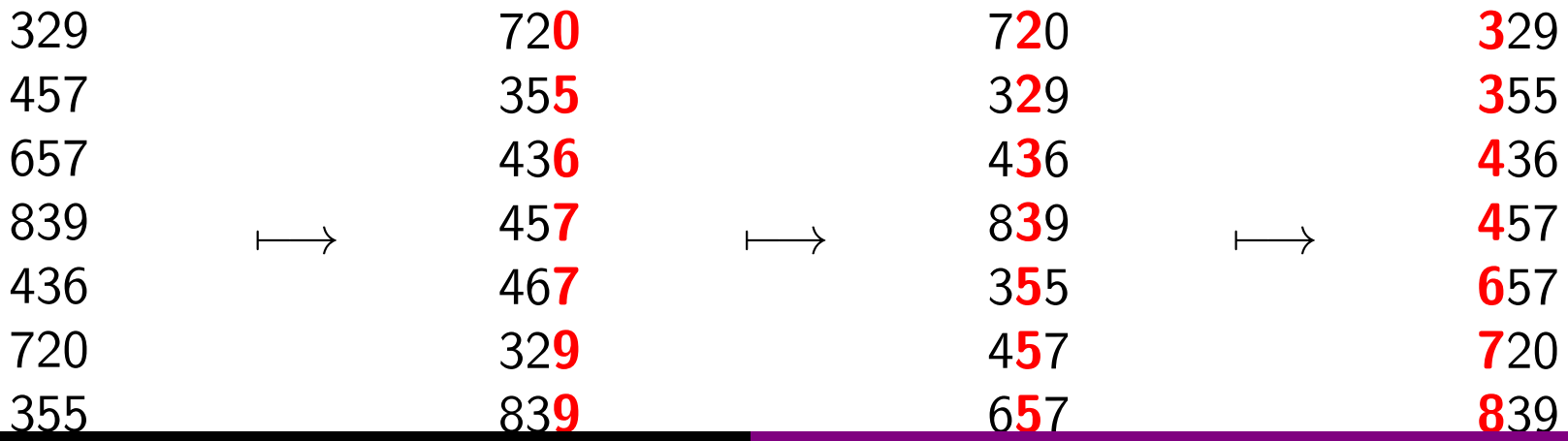
Repaso: Radix Sort (versión para ordenar naturales)

- **Radix Sort**, en su versión más básica, ordena una lista de naturales.
- **Idea:**
  1. Ordenar los valores mirando sólo el dígito 1
  2. Ordenar los valores mirando sólo el dígito 2 (manteniendo el orden en caso de empate)
  3. Ordenar los valores mirando sólo el dígito 3 (manteniendo el orden en caso de empate)
  4. ...
- Ejemplo:



Repaso: Radix Sort (versión para ordenar naturales)

- **Radix Sort**, en su versión más básica, ordena una lista de naturales.
- **Idea:**
  1. Ordenar los valores mirando sólo el dígito 1
  2. Ordenar los valores mirando sólo el dígito 2 (manteniendo el orden en caso de empate)
  3. Ordenar los valores mirando sólo el dígito 3 (manteniendo el orden en caso de empate)
  4. ...
- Ejemplo:



Repaso: Radix Sort (versión para ordenar naturales)

- Llamamos d a la cantidad máxima de dígitos de los elementos y suponemos que el dígito 1 es el menos significativo de cada valor.

Algorithm 3 RADIX-SORT(A, d)

```
for i ← [1..d] // del menos significativo al más significativo do
    Ordenar el arreglo A según el dígito i, en forma estable
end for
```

- Recordemos que un algoritmo es **estable** si preserva el orden original entre las cosas equivalentes.
- ¿Complejidad? $d \times O(\text{ordenar } A \text{ por un dígito})$.
- ¿Con Bucket Sort (sin ordenar los buckets)? $d \times O(n) = O(n \cdot d)$.
- O sea... $O(n \cdot \log(\max(A)))$.

Repaso: Radix Sort (generalizando)

- Radix Sort puede usarse también para ordenar cadenas de caracteres (la primera letra es la más significativa).
- El mismo esquema sirve para ordenar tuplas, donde el orden que se quiera dar depende principalmente de un "dígito" (i.e., un campo de la tupla) y en el caso de empate se tengan que revisar los otros.
- La idea en el fondo es la misma: usar un algoritmo estable e ir ordenado por "dígito" (del menos al más significativo).

Algorithm 4 RADIX-SORT(A, d)

for $i \leftarrow [1..d]$ // del menos significativo al más significativo **do**
    Ordenar el arreglo A según el dígito i, en forma estable
end for

- La complejidad en este caso es $d \times O(\text{ordenar } A \text{ por un dígito})$.

Sobre los ejercicios de sorting

Algunas aclaraciones:

- Requieren práctica y paciencia.
- También requieren saber un poco de estructuras de datos.
- NO requieren que diseñen dichas estructuras (a menos que no sean las de siempre; como cuando hacen los ejercicios de elección de estructuras).
- Cada línea de código que escriban debería estar acompañada de la correspondiente complejidad temporal.

Sobre los ejercicios de sorting

Algunas estrategias:

- Utilizar algoritmos ya conocidos (Merge Sort, Quick Sort, Counting Sort, Bucket Sort, Radix Sort, etc.)
- Utilizar estructuras de datos ya conocidas (AVL, Heap, Trie, listas enlazadas, etc.)
- Analizar la complejidad pedida e deducir algo de eso.
- Si conocemos algo de la entrada, ver cómo se puede usar para mejorar la complejidad.

Repaso: Ejercicio 16

1. Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n en tiempo $O(n)$.
2. Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n^2 en tiempo $O(n)$. *Pista: Usar varias llamadas al ítem anterior.*
3. Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n^k para una constante arbitraria pero fija k.
4. ¿Qué complejidad se obtiene si se generaliza la idea para arreglos de enteros no acotados?

Primer ejercicio: Ejercicio 6

Se tiene un arreglo de n números naturales que se quiere ordenar por frecuencia, y en caso de igual frecuencia, por su valor.

Describa un algoritmo que realice el ordenamiento descrito, utilizando las estructuras de datos intermedias que considere necesarias. Calcule el orden de complejidad temporal del algoritmo propuesto.

Por ejemplo, a partir del arreglo [1, 3, 1, 7, 2, 7, 1, 7, 3] se quiere obtener [1, 1, 1, 7, 7, 7, 3, 3, 2].

Segundo ejercicio: Ejercicio 19

Se tienen dos arreglos de enteros, el primero contiene los elementos a ordenar en la salida y el segundo determina un criterio de ordenamiento.

OrdenarSegunCriterio(In s: `Array<int>`, In crit: `Array<int>`) :`Array<int>`

El usuario quiere reordenar *s* teniendo en cuenta el orden de aparición de los elementos definidos en *crit*. Los elementos que están presentes en *s*, pero no están presentes en *crit*, deben encontrarse al final del arreglo *res* y estar en orden creciente. El arreglo *crit* es de tamaño menor o igual que *s*, no tiene repetidos y puede contener elementos extra que no son parte de *s*.

Segundo ejercicio: Ejercicio 19

A continuación se presenta un ejemplo:

`OrdenarSegunCriterio(s, crit) = [3,3,3,5,5,7,1,4,4,8,9]`

Como en *crit* encontramos primero a 3, luego 5 y detrás 7, todas las apariciones de 3 aparecen primeras, seguidas por las apariciones de 5 y finalmente las de 7. Como 2 y 6 no están en s, son ignorados. Luego, se encuentran todos los elementos de s que no sean 3, 5 o 7 en orden ascendente.

Se pide dar un algoritmo que haga lo requerido con complejidad de $O(n.log(n))$ siendo n la cantidad de elementos de s

Tercer ejercicio: Ejercicio 18

Dado un arreglo de pares `<estudiante, nota>`, queremos devolver un arreglo de pares `<estudiante, promedio>`, ordenado en forma descendente por promedio y, en caso de empate, por número de libreta.
Los estudiantes se representan con un número de libreta (puede considerar que es un número de a lo sumo 10 dígitos), y la nota es un número entre 1 y 100. La cantidad de estudiantes se considera no acotada.
Se pide dar un algoritmo que resuelva el problema con complejidad $O(n + m \log m)$, donde n es la cantidad total de notas (es decir, el tamaño del arreglo de entrada), y m es la cantidad total de estudiantes.

Tercer ejercicio: Ejercicio 18

Entrada: $< 12395, 72 >, < 45615, 81 >, < 12395, 94 >, < 45615, 100 >, < 78920, 81 >, < 12395, 90 >, < 45615, 71 >, < 78920, 50 >$

Salida:$< 12395, 85.3333 >, < 45615, 84 >, < 78920, 65.5 >$

1. Describa cada etapa del algoritmo con sus palabras, y justifique por qué cumple con las complejidades pedidas.
2. Escriba el algoritmo en pseudocódigo. Puede utilizar (sin reescribir) todos los algoritmos y estructuras de datos vistos en clase.