Ordenamiento - Sorting

Algoritmos y Estructuras de Datos

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

1° cuatrimestre 2024

Plan del día

- Mini repaso de los algoritmos de ordenamiento vistos en la teórica
- Bucket, Counting y Radix Sort
- Ejercicio 12 práctica 8
- Ejercicio 5 prática 8

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Es **estable**.

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Es **estable**.

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

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

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Es **estable**.

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

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

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

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

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Es **estable**.

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

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

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Es **estable**.

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

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

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

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

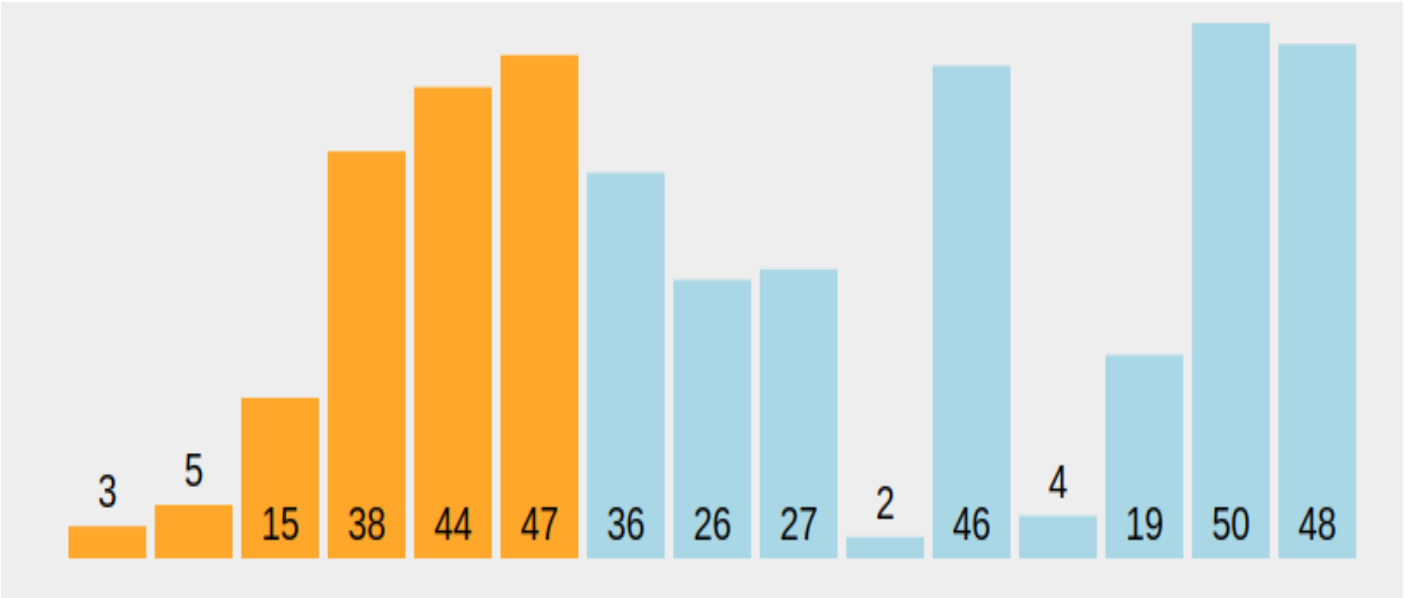
- QuickSort, $O(n^2)$ en el peor caso. $O(n \log n)$ en el caso promedio.

  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**.

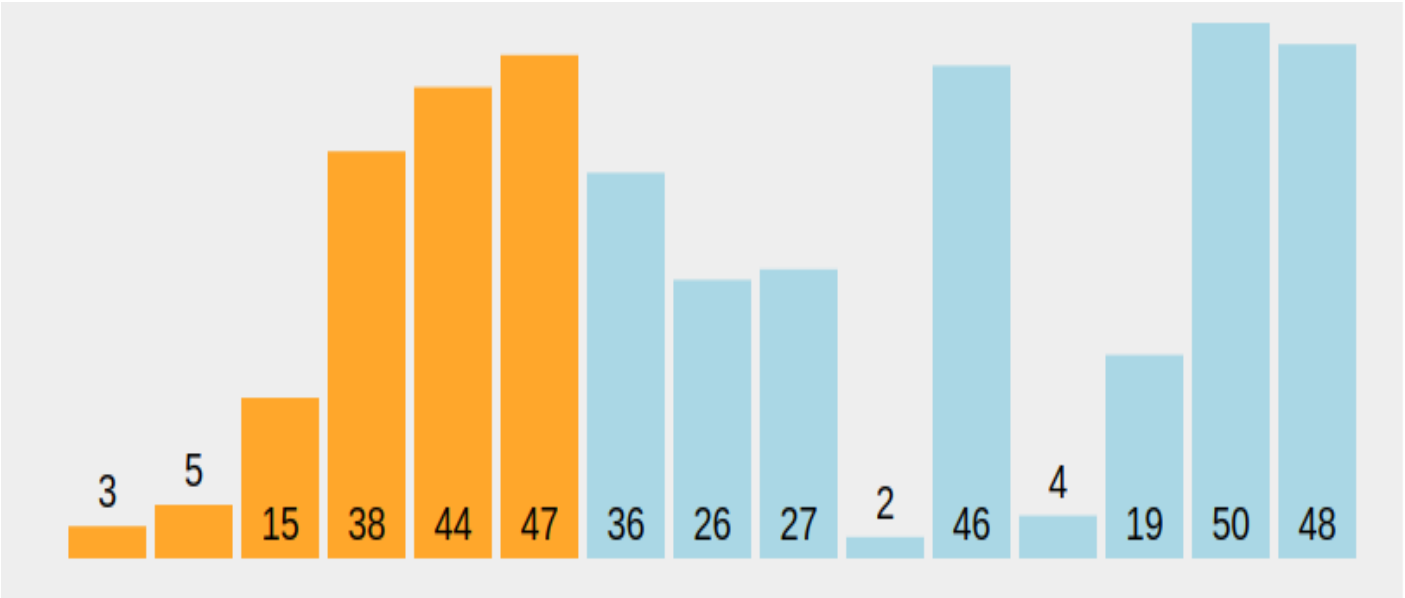
- HeapSort, $O(n + n \log n) = O(n \log n)$

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

Trivia - Adivinen qué algoritmo se está usando

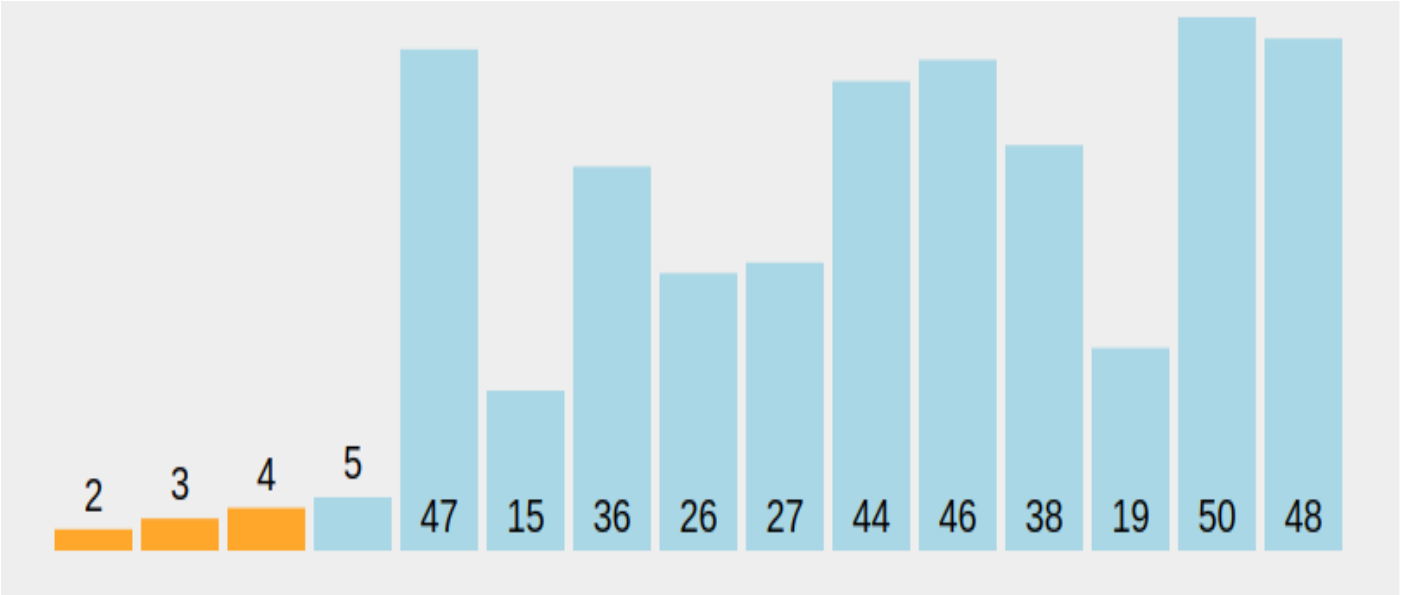


Trivia - Adivinen qué algoritmo se está usando

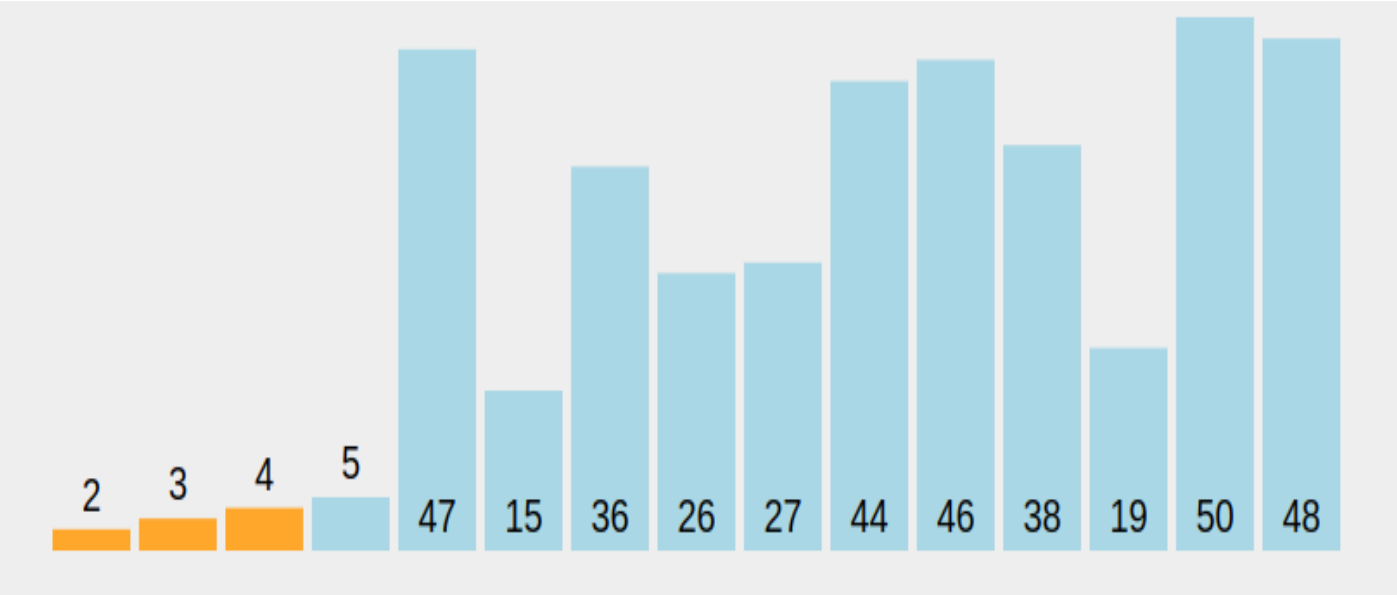


Insertion Sort

Trivia - Adivinen qué algoritmo se está usando

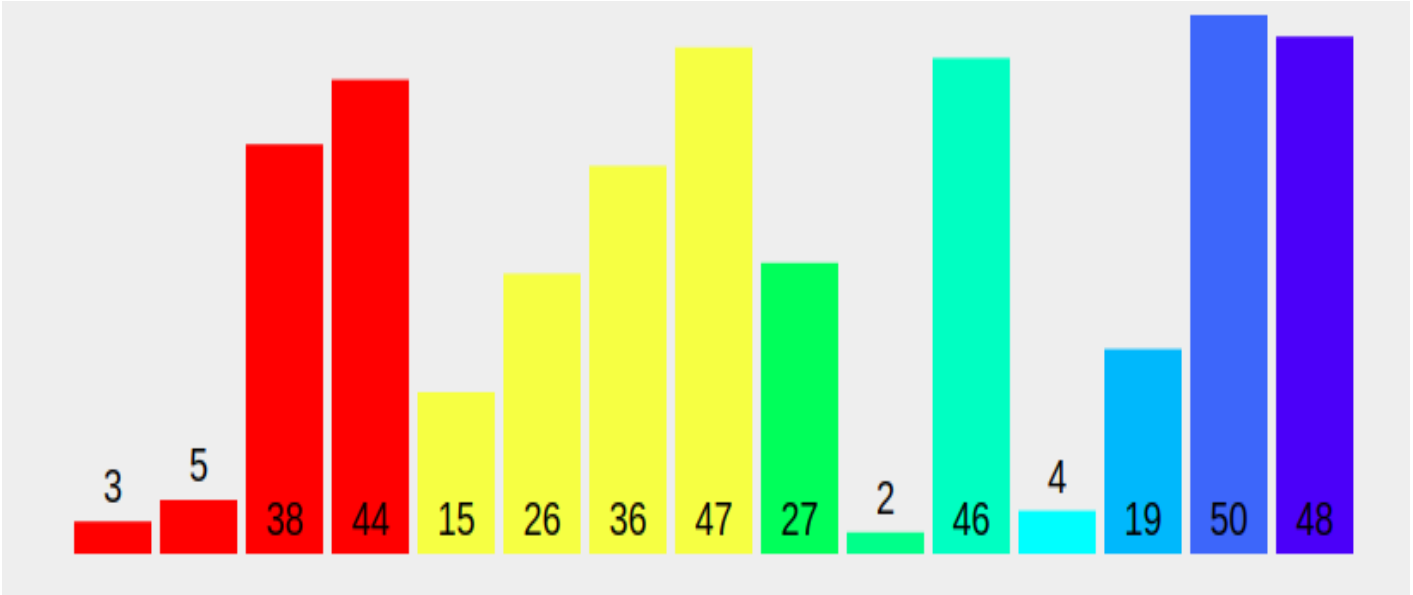


Trivia - Adivinen qué algoritmo se está usando

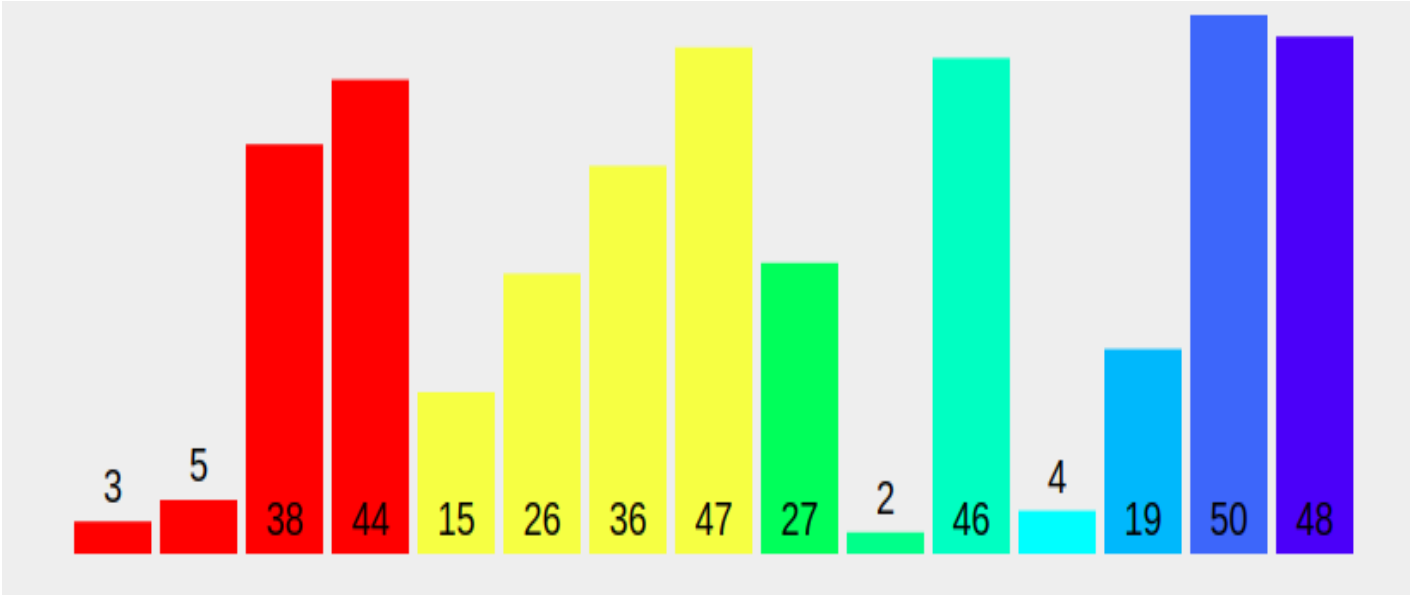


Selection Sort

Trivia - Adivinen qué algoritmo se está usando



Trivia - Adivinen qué algoritmo se está usando



Merge Sort

Reflexión

- Estos algoritmos ordenan arreglos comparando los elementos, es decir, son algoritmos de sorting por comparación.
- Vimos en la teórica que en el peor caso son $\Omega(n \log n)$.
- ¿Puede haber algo mejor?
- Sí, por ejemplo, si tenemos información de los elementos.
- O podríamos tener complejidades sujetas a otros factores, más allá del tamaño de entrada.

Bucket Sort

- **Bucket Sort** ordena arreglos de cualquier tipo.
- 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.

Bucket Sort

- **Bucket Sort** ordena arreglos de cualquier tipo.
- 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.

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?

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 $n = length(A)$.

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... $O(n+M)$.

Counting Sort

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

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 $B[j]$.

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 $k = \max_i\{A[i]\} + 1$.

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 $[d, d + k)$.

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
```

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?

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 $n = length(A)$

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)

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

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)

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)

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. ...

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:

  329
  457
  657
  839
  436
  720
  355

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:

329		72**0**
457		35**5**
657		43**6**
839	$\longmapsto$	45**7**
436		46**7**
720		32**9**
355		83**9**

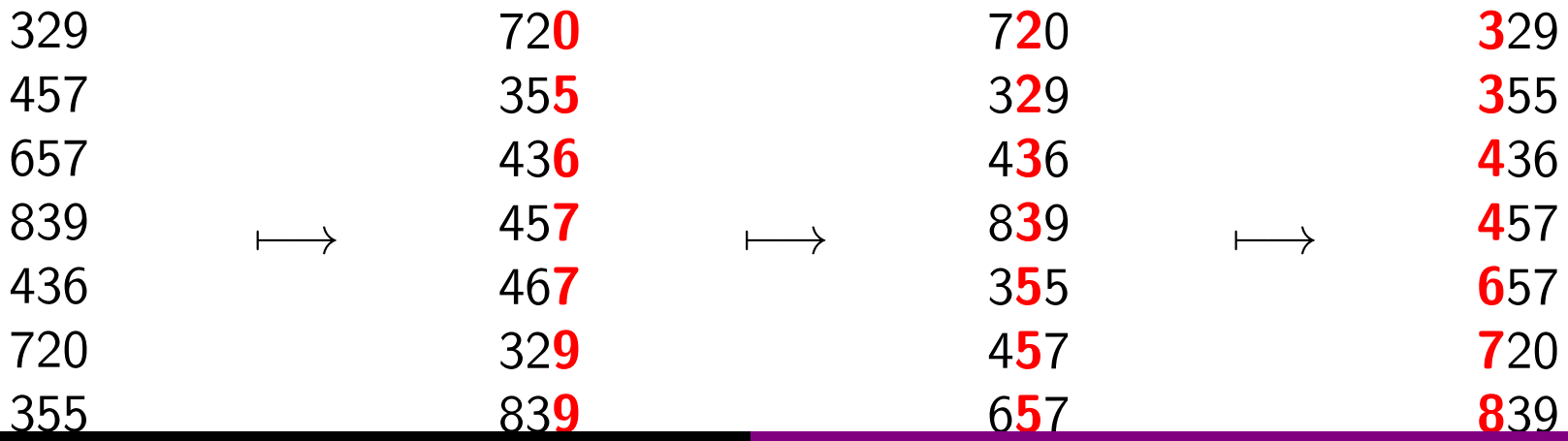
Radix Sort (versión para ordenar naturales)

- **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:

329		720		720
457		355		329
657		436		436
839	$\longmapsto$	457	$\longmapsto$	839
436		467		355
720		329		457
355		839		657

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:



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.

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?

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 $i \leftarrow [1..d]$ // 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)?

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 $i \leftarrow [1..d]$ // 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)? $d \times O(n) = O(n \cdot d)$.

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... $O(n \cdot \log(\max(A)))$.

Radix Sort (generalizando)

- Radix Sort puede usarse también para ordenar cadenas de caracteres (la primera letra es la más significativa).

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.

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.

Primer ejercicio: La distribución loca

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.

Primer ejercicio: Lo importante

- Arreglo de **n enteros positivos**
- De los n elementos, $\sqrt{n}$ **están afuera del rango** $[20, 40]$
- Quieren $O(n)$

Discutamoslo...

Primer ejercicio: Conclusiones

- Está bueno ver la "forma" que tiene la entrada en un problema de ordenamiento
- A veces se pueden combinar distintos algoritmos de ordenamiento para lograr los objetivos.

Segundo ejercicio: Ejercicio 5 Práctica 8

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: Conclusiones

En este ejercicio vimos estrategias que se pueden extrapolar a muchos ejercicios de la práctica tales como:

- Usar estructuras intermedias aprovechando sus propiedades, en caso de tenerlas
- Ordenar con diferentes criterios un mismo arreglo
- La importancia de la estabilidad de un algoritmo de ordenamiento en caso de necesitarla
- Reconstruir nuestra estructura intermedia para devolver la salida como se pide en el problema.