Ordenamiento - Sorting

Algoritmos y Estructuras de Datos

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

15 de noviembre de 2023

¿Por qué sorting?

- Es uno de los problemas clásicos de computación, del área de algoritmos.

Motivación

¿Por qué sorting?

- Es uno de los problemas clásicos de computación, del área de algoritmos.
- Buscar elementos en una estructura es mucho más eficiente si está ordenada.

Motivación

¿Por qué sorting?

- Es uno de los problemas clásicos de computación, del área de algoritmos.
- Buscar elementos en una estructura es mucho más eficiente si está ordenada.
- Es muy común que un algoritmo ordene algo como subrutina, y así su eficiencia está atada al algoritmo de sorting que use.

¿Por qué sorting?

- Es uno de los problemas clásicos de computación, del área de algoritmos.
- Buscar elementos en una estructura es mucho más eficiente si está ordenada.
- Es muy común que un algoritmo ordene algo como subrutina, y así su eficiencia está atada al algoritmo de sorting que use.
- Tener buenos algoritmos de sorting puede ayudar a resolver muchos problemas en forma eficiente.

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Trabaja in-place.

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Trabaja in-place.

- Selection Sort, $\Theta(n^2)$

  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. Trabaja in-place.

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Trabaja in-place.

- Selection Sort, $\Theta(n^2)$

  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. Trabaja in-place.

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

  Algoritmo recursivo que ordena sus dos mitades y luego las fusiona. No es in-place.

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Trabaja in-place.

- Selection Sort, $\Theta(n^2)$

  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. Trabaja in-place.

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

  Algoritmo recursivo que ordena sus dos mitades y luego las fusiona. No es in-place.

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

  Elige un pivote y separa los elementos entre los menores y los mayores a él. Luego, se ejecuta el mismo procedimiento sobre cada una de las partes. Es de lo mejor que hay en la práctica.

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$. Trabaja in-place.

- Selection Sort, $\Theta(n^2)$

  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. Trabaja in-place.

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

  Algoritmo recursivo que ordena sus dos mitades y luego las fusiona. No es in-place.

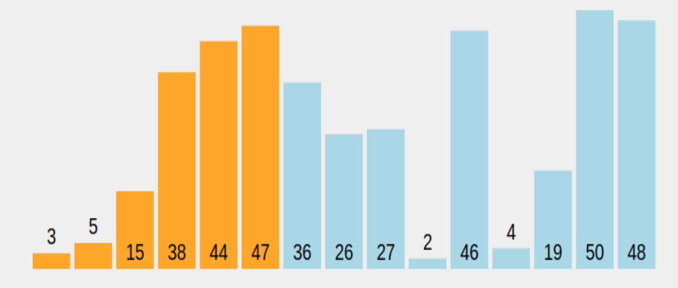
- QuickSort, $\Theta(n^2)$ en el peor caso. $\Theta(n \log n)$ en el caso promedio.

  Elige un pivote y separa los elementos entre los menores y los mayores a él. Luego, se ejecuta el mismo procedimiento sobre cada una de las partes. Es de lo mejor que hay en la práctica.

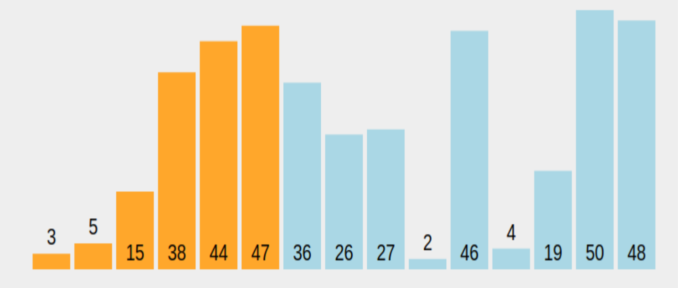
- HeapSort, $O(n + n \log n) = O(n \log n)$

  Arma el Heap en $O(n)$ y va sacando los elementos ordenados pagando $O(\log n)$.

Trivia - Adivinen qué algoritmo se está usando

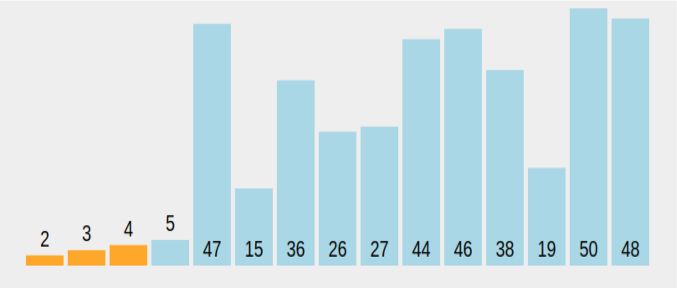


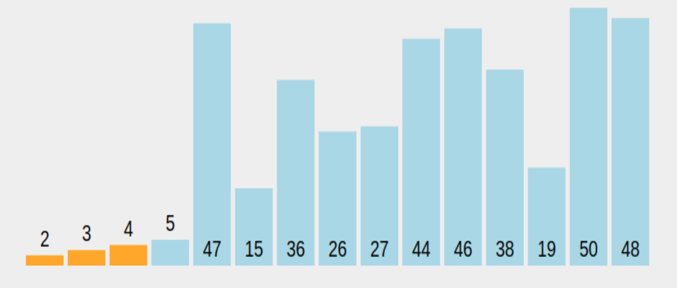
Trivia - Adivinen qué algoritmo se está usando



Insertion Sort

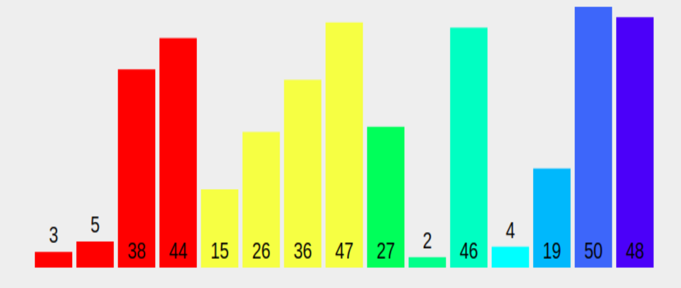
Trivia - Adivinen qué algoritmo se está usando



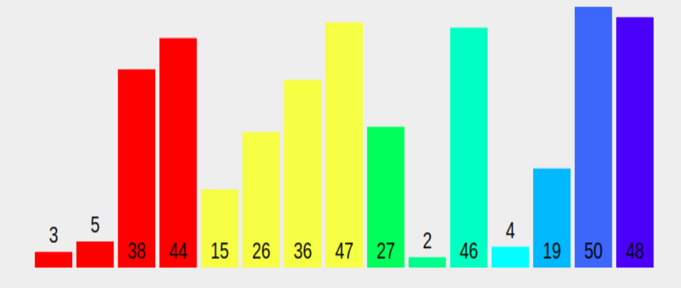


Selection Sort

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 cateoría 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 cateoría 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 $B[0], B[1], \ldots, B[M-1]$

- ¿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)

```
B ← arreglo de tamaño k
for i ← [0..k − 1] do
    B[i] ← 0
end for
//cuento la cantidad de apariciones de cada elemento.
for j ← [0..length(A) − 1] do
    B[A[j]] ← B[A[j]] + 1
end for
//inserto en A la cantidad de apariciones de cada elemento.
indexA ← 0
for i ← [0..k − 1] do
    for j ← [1..B[i]] do
        A[indexA] ← i
        indexA ← indexA + 1
    end for
end for
```

Counting Sort

Precondición: $k > \max_i\{A[i]\}$

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

$B \leftarrow$ arreglo de tamaño k
for $i \leftarrow [0..k-1]$ **do**
    $B[i] \leftarrow 0$
end for
//cuento la cantidad de apariciones de cada elemento.
for $j \leftarrow [0..length(A)-1]$ **do**
    $B[A[j]] \leftarrow B[A[j]] + 1$
end for
//inserto en A la cantidad de apariciones de cada elemento.
$indexA \leftarrow 0$
for $i \leftarrow [0..k-1]$ **do**
    for $j \leftarrow [1..B[i]]$ **do**
        $A[indexA] \leftarrow i$
        $indexA \leftarrow indexA + 1$
    end for
end for

- ¿Complejidad?

Counting Sort

Precondición: $k > \max_i\{A[i]\}$

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

```
B ← arreglo de tamaño k
for i ← [0..k − 1] do
    B[i] ← 0
end for
//cuento la cantidad de apariciones de cada elemento.
for j ← [0..length(A) − 1] do
    B[A[j]] ← B[A[j]] + 1
end for
//inserto en A la cantidad de apariciones de cada elemento.
indexA ← 0
for i ← [0..k − 1] do
    for j ← [1..B[i]] do
        A[indexA] ← i
        indexA ← indexA + 1
    end for
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)

```
B ← arreglo de tamaño k
for i ← [0..k − 1] do
    B[i] ← 0
end for
//cuento la cantidad de apariciones de cada elemento.
for j ← [0..length(A) − 1] do
    B[A[j]] ← B[A[j]] + 1
end for
//inserto en A la cantidad de apariciones de cada elemento.
indexA ← 0
for i ← [0..k − 1] do
    for j ← [1..B[i]] do
        A[indexA] ← i
        indexA ← indexA + 1
    end for
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		72**0**		7**2**0
457		35**5**		3**2**9
657		43**6**		4**3**6
839	$\longmapsto$	45**7**	$\longmapsto$	8**3**9
436		46**7**		3**5**5
720		32**9**		4**5**7
355		83**9**		6**5**7

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**		7**20**		**3**29
457		35**5**		3**29**		**3**55
657		43**6**		4**36**		**4**36
839	$\longmapsto$	45**7**	$\longmapsto$	8**39**	$\longmapsto$	**4**57
436		46**7**		3**55**		**6**57
720		32**9**		4**57**		**7**20
355		83**9**		6**57**		**8**39

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

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 $i \leftarrow [1..d]$ // 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)? $d \times O(n) = O(n \cdot d)$.
- 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 ← [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 $d \times O(\text{ordenar } A \text{ por un dígito})$.

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: Una planilla de notas

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

- alumno es $\langle$*nombre*: string, *turno*: Turno, *puntaje*: Nota$\rangle$
- donde Turno es *enum*$\{$M, T$\}$
- Nota es un nat no mayor que 10.

Se necesita ordenar un arreglo(alumno) para que todos los alumnos del turno mañana aparezcan al inicio según un orden creciente de notas y todos los del turno tarde al final con el mismo orden. Por ejemplo:

Entrada

Andrés	M	10
Clara	T	6
Rita	M	6
Paula	M	7
Jose	T	7
Pedro	T	8

Salida

Rita	M	6
Paula	M	7
Andrés	M	10
Clara	T	6
Jose	T	7
Pedro	T	8

Segundo ejercicio: Una planilla de notas

a) Proponer un algoritmo de ordenamiento ORDENAPLANILLA(IN/OUT 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.

Segundo ejercicio: Lo importante

alumno es tupla$\langle$*nombre*: string, *turno*: Turno, *puntaje*: Nota$\rangle$
donde Turno es *enum*{M, T} y Nota es un nat no mayor que 10.

Entrada

Andrés	M	10
Clara	T	6
Rita	M	6
Paula	M	7
Jose	T	7
Pedro	T	8

Salida

Rita	M	6
Paula	M	7
Andrés	M	10
Clara	T	6
Jose	T	7
Pedro	T	8

- Complejidad temporal O(n) en el peor caso, donde n es la cantidad de elementos del arreglo.
- Turno mañana primero.
- Hay que ordenar por puntaje.

Segundo ejercicio: Una planilla de notas

Entrada

Andrés	M	10
Jose	T	7
Clara	T	6
Marta	M	7
Paula	M	7
Pedro	T	8

$\overset{??}{\longmapsto}$

Salida?

Marta	M	7
Paula	M	7
Andrés	M	10
Clara	N	6
Jose	N	7
Pedro	M	8

Salida?

Paula	M	7
Marta	M	7
Andrés	M	10
Clara	N	6
Jose	N	7
Pedro	M	8

b) Si la planilla original estaba ordenada alfabéticamente por nombre. ¿Podemos asegurar que si existen dos o más alumnos del mismo turno y con igual nota, entonces éstos aparecerán en orden alfabético en la planilla ordenada?

Segundo ejercicio: Una planilla de notas

Entrada

Andrés	M	10
Jose	T	7
Clara	T	6
Marta	M	7
Paula	M	7
Pedro	T	8

?? $\longmapsto$

Salida?

Marta	M	7
Paula	M	7
Andrés	M	10
Clara	N	6
Jose	N	7
Pedro	M	8

Salida?

Paula	M	7
Marta	M	7
Andrés	M	10
Clara	N	6
Jose	N	7
Pedro	M	8

b) Si la planilla original estaba ordenada alfabéticamente por nombre. ¿Podemos asegurar que si existen dos o más alumnos del mismo turno y con igual nota, entonces éstos aparecerán en orden alfabético en la planilla ordenada?

¡Sí! ¡Nuestro algoritmo es estable!

Segundo ejercicio: Una planilla de notas

Entrada

Andrés	M	10
Jose	T	7
Clara	T	6
Marta	M	7
Paula	M	7
Pedro	T	8

$\overset{??}{\longmapsto}$

Salida?

Marta	M	7
Paula	M	7
Andrés	M	10
Clara	N	6
Jose	N	7
Pedro	M	8

Salida?

Paula	M	7
Marta	M	7
Andrés	M	10
Clara	N	6
Jose	N	7
Pedro	M	8

b) Si la planilla original estaba ordenada alfabéticamente por nombre. ¿Podemos asegurar que si existen dos o más alumnos del mismo turno y con igual nota, entonces éstos aparecerán en orden alfabético en la planilla ordenada?

¡Sí! ¡Nuestro algoritmo es estable!

c) ¿Qué habría que modificar de nuestro algoritmo para que ordene igual que antes pero ante igual turno y nota ordene por orden alfabético? ¿Cuál sería su complejidad?

Segundo ejercicio: Conclusiones

- Radix Sort ordena números, pero el concepto que usa es mucho más que eso cuando se puede pensar la entrada como una lista o tupla.
- “Sirve” cuando la cantidad de elementos a ordenar es muy grande en comparación al tamaño del rango de los elementos.
- Los algoritmos estables se llevan bastante bien entre sí.
- Si tenemos una jerarquía para ordenar tenemos que ordenar primero por el criterio menos importante e ir subiendo.

Radix Sort (versión con cambio de base)

- El Radix Sort que vimos trabaja en base 10
- Pero el esquema de Radix Sort puede aplicarse a cualquier base.

Algorithm 5 RadixSort(A, b)

$n \leftarrow length(A)$
$d \leftarrow \log_b Max(A)$
$B \leftarrow$ arreglo de n arreglos de tamaño d
for $i \leftarrow [0..n-1]$ **do**
    $B[i] \leftarrow$ descomposición en base b de $A[i]$
    // Los números con menos de d dígitos se completan a izquierda con 0.
end for
for $j \leftarrow [1..d]$ **do**
    Ordenar el arreglo B según el dígito j, en forma estable
end for
return volverNumerosASuBase(B, b)

Radix Sort (versión con cambio de base)

- El Radix Sort que vimos trabaja en base 10
- Pero el esquema de Radix Sort puede aplicarse a cualquier base.

Algorithm 5 RadixSort(A, b)

$n \leftarrow length(A)$
$d \leftarrow \log_b Max(A)$
$B \leftarrow$ arreglo de n arreglos de tamaño d
for $i \leftarrow [0..n-1]$ **do**
    $B[i] \leftarrow$ descomposición en base b de $A[i]$
    // Los números con menos de d dígitos se completan a izquierda con 0.
end for
for $j \leftarrow [1..d]$ **do**
    Ordenar el arreglo B según el dígito j, en forma estable
end for
return volverNumerosASuBase(B, b)

- ¿Complejidad?

Radix Sort (versión con cambio de base)

- El Radix Sort que vimos trabaja en base 10
- Pero el esquema de Radix Sort puede aplicarse a cualquier base.

Algorithm 5 RadixSort(A, b)

$n \leftarrow length(A)$
$d \leftarrow \log_b Max(A)$
$B \leftarrow$ arreglo de n arreglos de tamaño d
for $i \leftarrow [0..n-1]$ **do**
  $B[i] \leftarrow$ descomposición en base b de $A[i]$
  // Los números con menos de d dígitos se completan a izquierda con 0.
end for
for $j \leftarrow [1..d]$ **do**
  Ordenar el arreglo B según el dígito j, en forma estable
end for
return volverNumerosASuBase(B, b)

- ¿Complejidad? $O(n \cdot d) + d \cdot O(n+b) = O(d \cdot (n+b))$, con $d = \log_b(\max(A))$

Radix Sort (versión con cambio de base)

- El Radix Sort que vimos trabaja en base 10
- Pero el esquema de Radix Sort puede aplicarse a cualquier base.

Algorithm 5 RadixSort(A, b)

$n \leftarrow length(A)$
$d \leftarrow \log_b Max(A)$
$B \leftarrow$ arreglo de n arreglos de tamaño d
for $i \leftarrow [0..n-1]$ **do**
  $B[i] \leftarrow$ descomposición en base b de $A[i]$
  // Los números con menos de d dígitos se completan a izquierda con 0.
end for
for $j \leftarrow [1..d]$ **do**
  Ordenar el arreglo B según el dígito j, en forma estable
end for
return volverNumerosASuBase(B, b)

- ¿Complejidad? $O(n \cdot d) + d \cdot O(n + b) = O(d \cdot (n + b))$, con $d = \log_b(\max(A))$
- **Observación:** A mayor base, menor cantidad de dígitos.

Radix Sort (versión con cambio de base)

- El Radix Sort que vimos trabaja en base 10
- Pero el esquema de Radix Sort puede aplicarse a cualquier base.

Algorithm 5 RadixSort(A, b)

$n \leftarrow length(A)$
$d \leftarrow \log_b Max(A)$
$B \leftarrow$ arreglo de n arreglos de tamaño d
for $i \leftarrow [0..n-1]$ **do**
  $B[i] \leftarrow$ descomposición en base b de $A[i]$
  // Los números con menos de d dígitos se completan a izquierda con 0.
end for
for $j \leftarrow [1..d]$ **do**
  Ordenar el arreglo B según el dígito j, en forma estable
end for
return volverNumerosASuBase(B, b)

- ¿Complejidad? $O(n \cdot d) + d \cdot O(n+b) = O(d \cdot (n+b))$, con $d = \log_b(\max(A))$
- **Observación:** A mayor base, menor cantidad de dígitos.
- **Observación:** b puede no ser una constante (!).