Ordenamiento - Sorting

Federico Hosen

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

27 de octubre del 2017

¿Por qué sorting?

- Es uno de los problemas clásicos de computación, del área de algoritmos.
- Cuando vemos un dropdown en una página web esperamos que esté ordenada.
- Es muy común que un algoritmo ordene algo como subrutina. Entonces su eficiencia está atada al algoritmo de sorting que use.
- Vimos en la teórica que hay una cota inferior para los algoritmos de sorting por *comparación*, entonces podemos demostrar que otros problemas también están acotados por abajo (era por abajo palacio!)

Algunas aclaraciones

- Este tipo de ejercicios requieren práctica y paciencia.
- También requieren que se sepa 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.

Algunas estrategias

- Utilizar algoritmos ya conocidos (Merge Sort, Quick Sort, etc.)
- Utilizar estructuras de datos ya conocidas.
- Intuir algo de la complejidad pedida.
- Revelación divina.

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

  Va *burbujeando* los elementos de a pares, llevando el más grande hacia adelante, cuando encuentra uno más grande burbujea ese.

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

  Va *burbujeando* los elementos de a pares, llevando el más grande hacia adelante, cuando encuentra uno más grande burbujea ese.
- 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.
- Bubble Sort, $\Theta(n^2)$ en el peor caso.

  Va *burbujeando* los elementos de a pares, llevando el más grande hacia adelante, cuando encuentra uno más grande burbujea ese.
- 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 el que mejor funciona 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.
- Bubble Sort, $\Theta(n^2)$ en el peor caso.

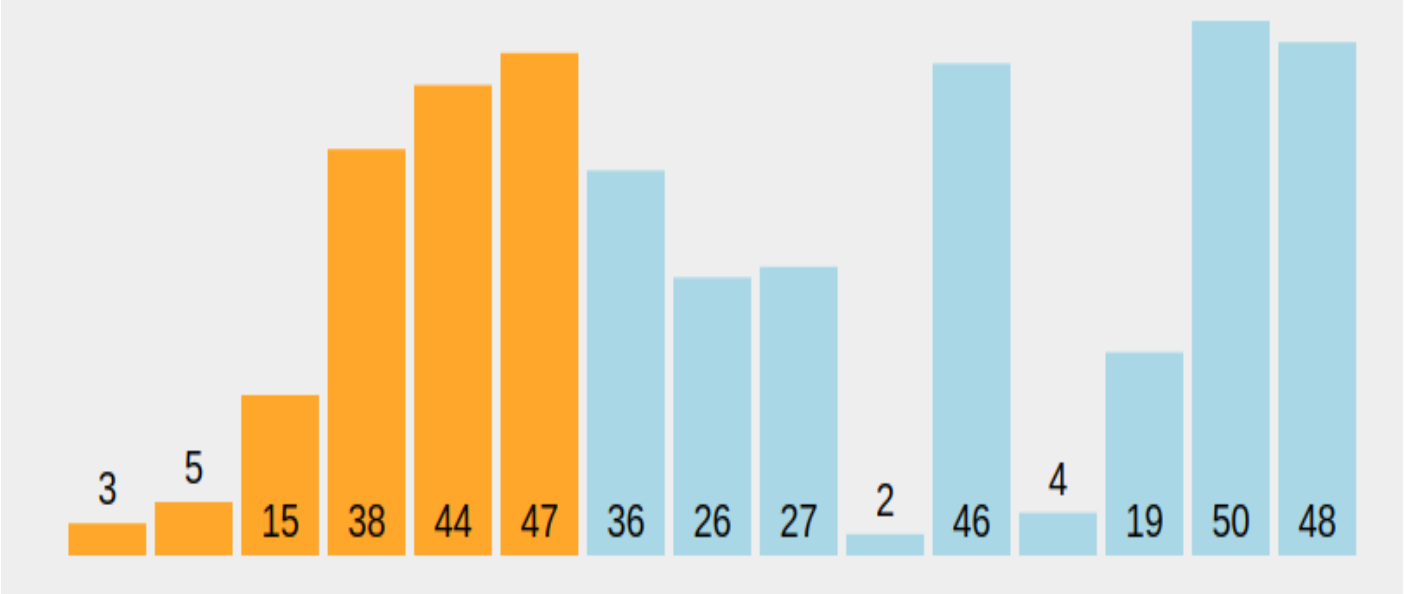
  Va *burbujeando* los elementos de a pares, llevando el más grande hacia adelante, cuando encuentra uno más grande burbujea ese.
- MergeSort, $\Theta(n \log n)$

  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.

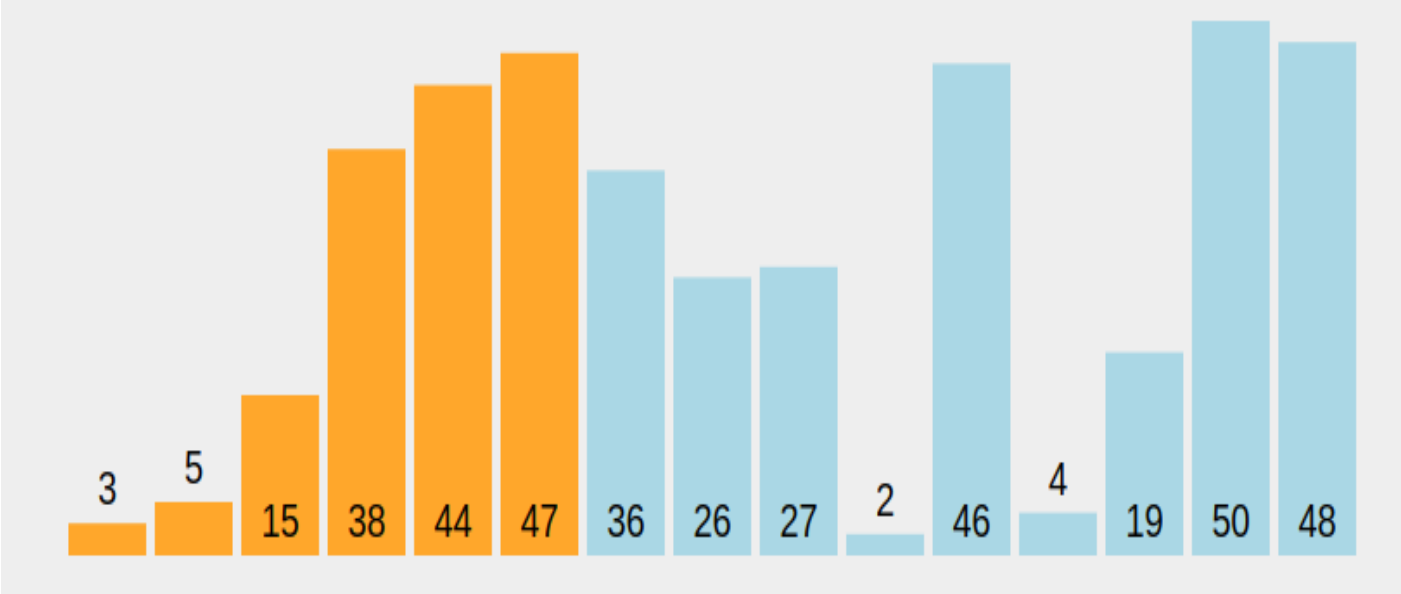
  Elige un pivote y separa los elementos entre los menores y los mayores a él. Luego, se ejecuta el mismo procedimiento sobre cada una de las partes. Es el que mejor funciona en la práctica.
- HeapSort, $\Theta(n + n \log n)) = O(n \log n)$

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

Trivia - Adivinen qué algoritmo se está usando

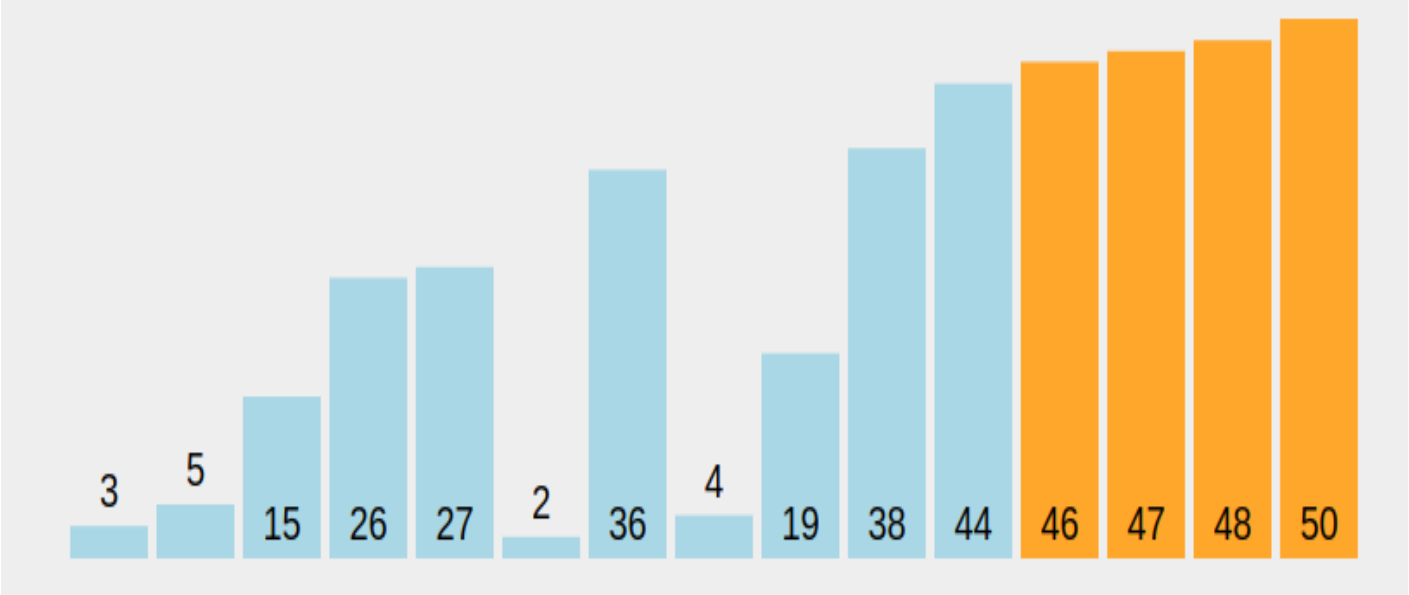


Trivia - Adivinen qué algoritmo se está usando

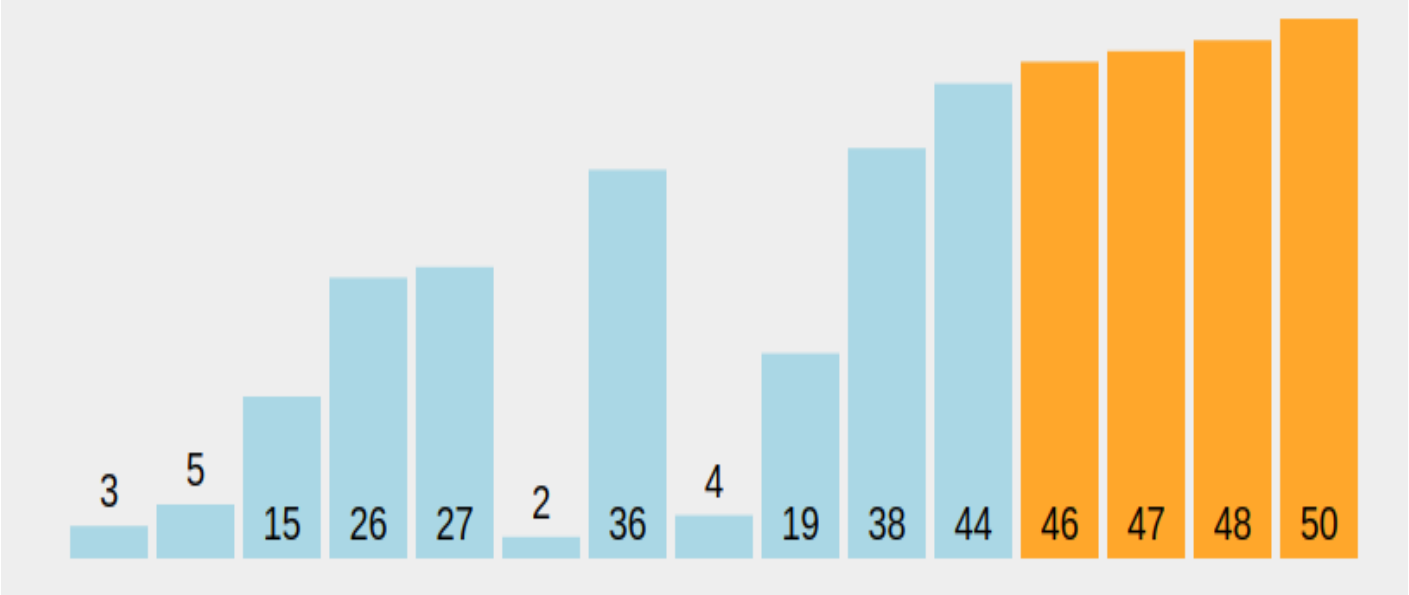


Insertion Sort

Trivia - Adivinen qué algoritmo se está usando

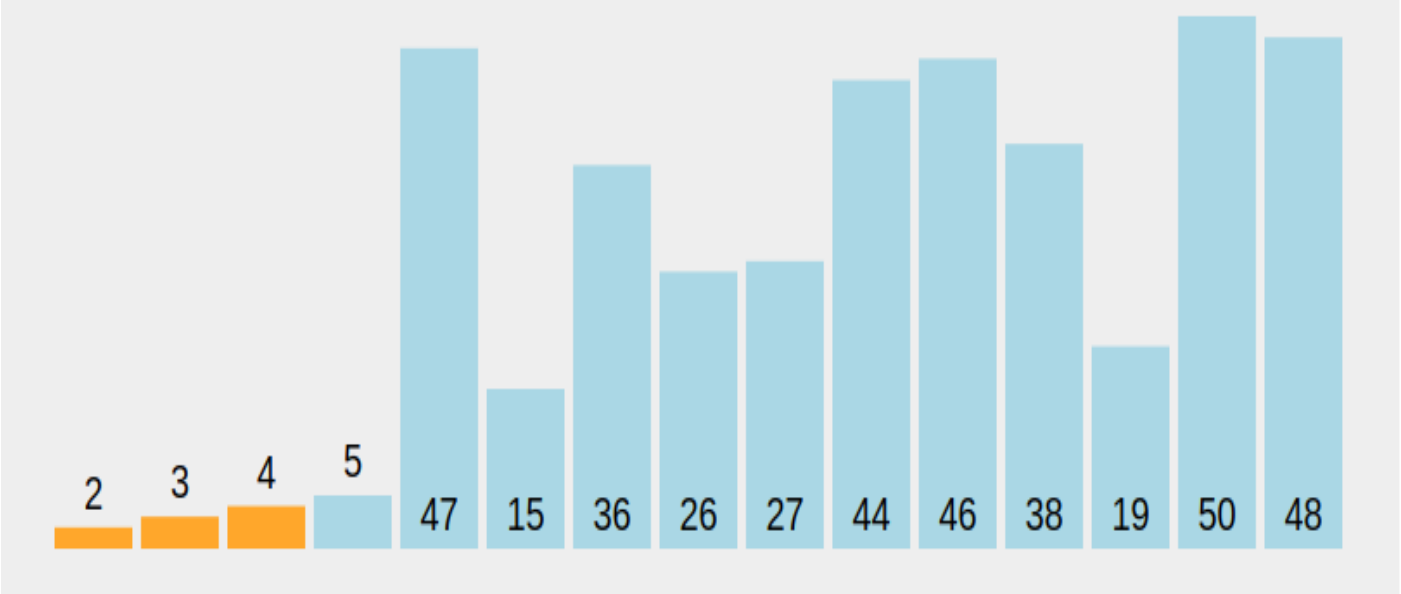


Trivia - Adivinen qué algoritmo se está usando

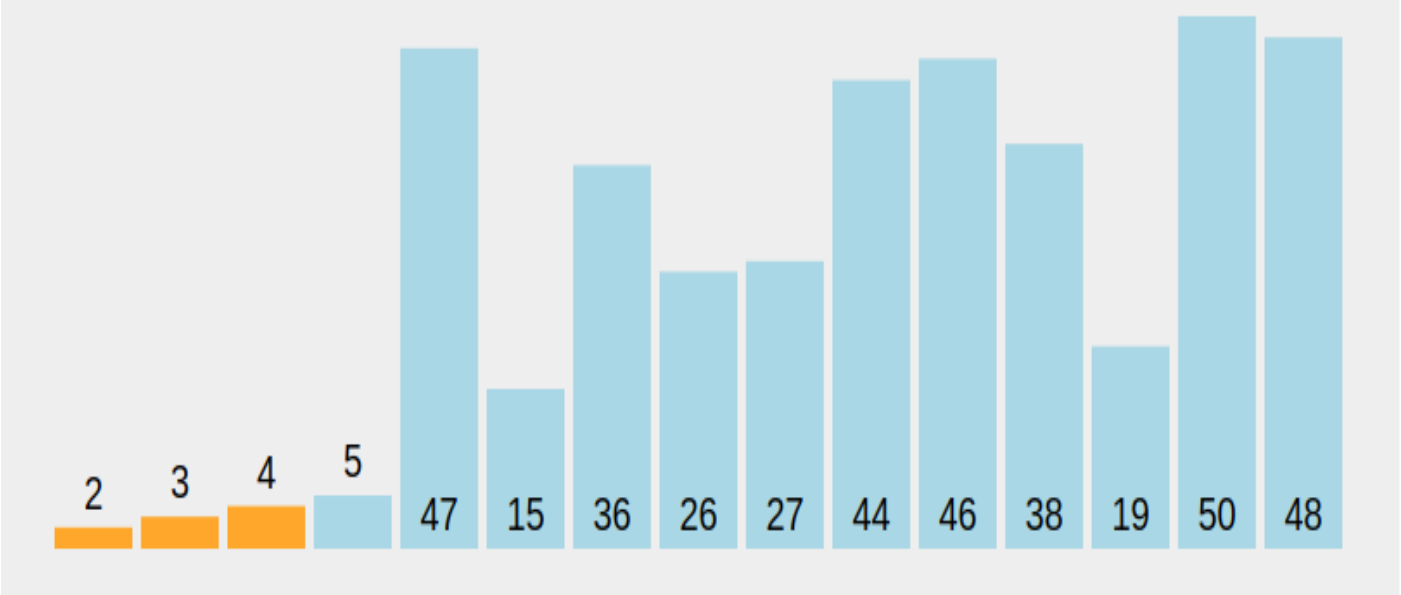


Bubble Sort

Trivia - Adivinen qué algoritmo se está usando

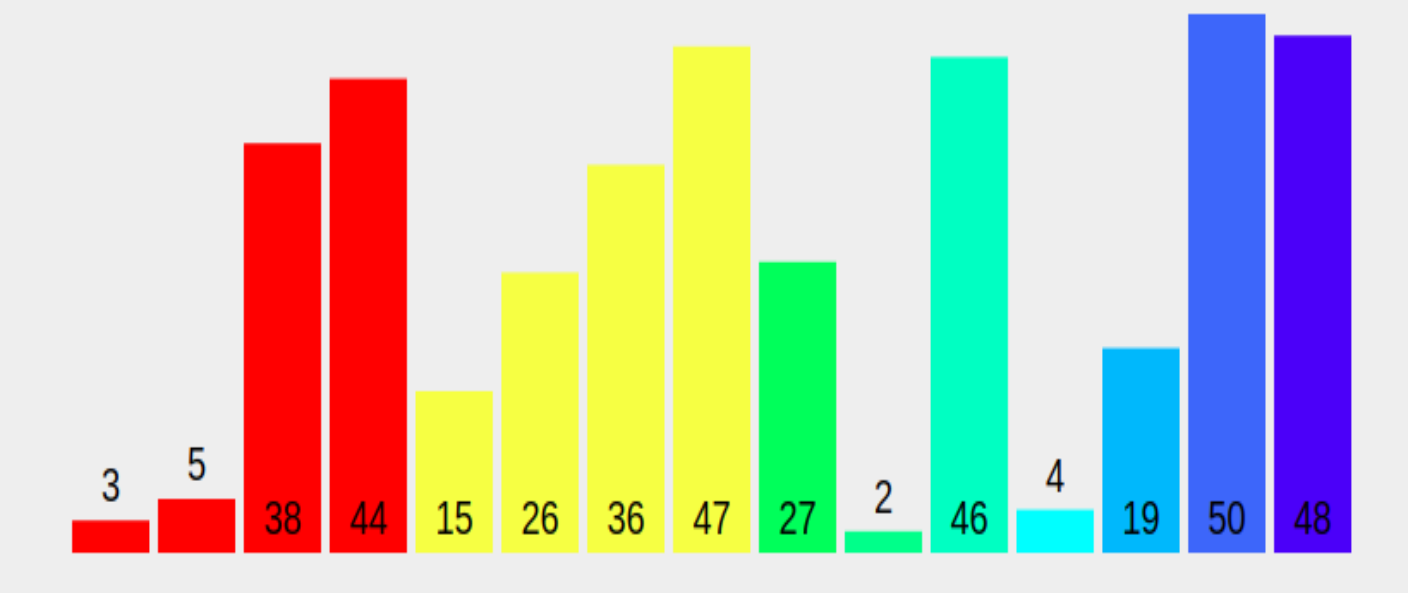


Trivia - Adivinen qué algoritmo se está usando

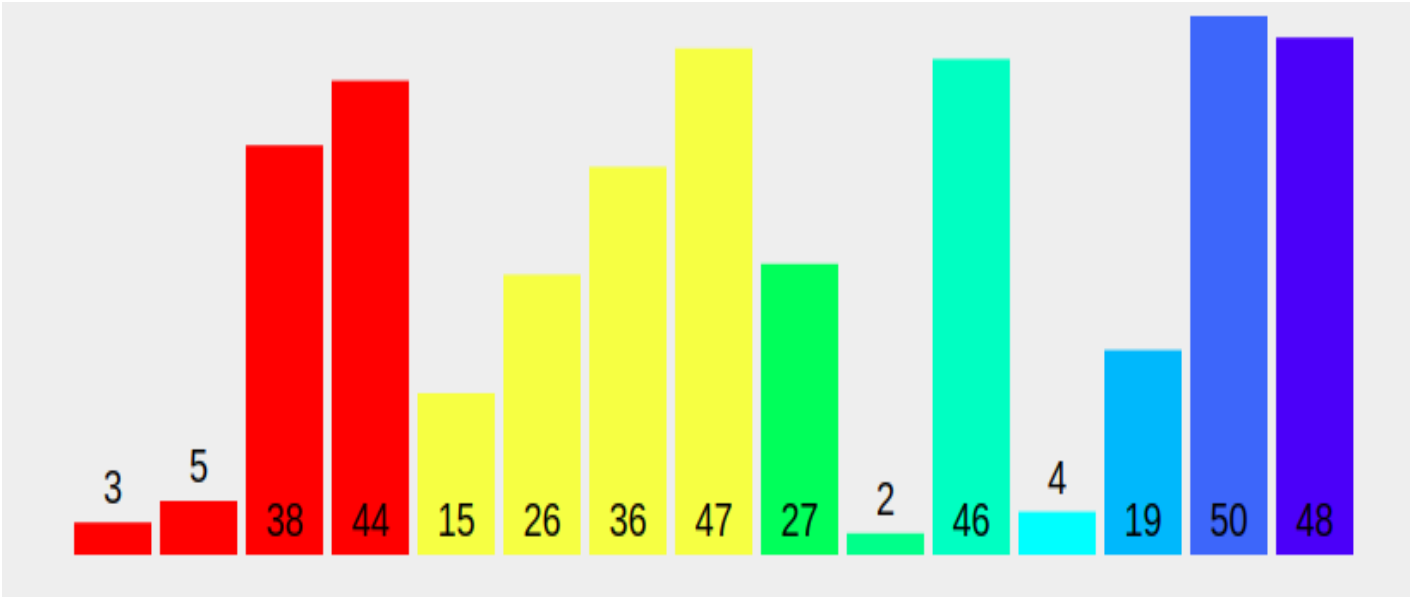


Selection Sort

Trivia - Adivinen qué algoritmo se está usando



Trivia - Adivinen qué algoritmo se está usando



Merge Sort

Reflexión

Todos estos algoritmos que vimos ordenan arreglos comparando los elementos, es decir, son algoritmos de sorting por comparación. Y vimos en la teórica que en el peor caso son $\Omega(n \log n)$. Entonces surge la pregunta ¿habrá algo mejor? Si tenemos información de los elementos, se podrá usar otra técnica para ordenar, más allá de comparar elemento a elemento. Se podrá logar algo.....lineal

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: La distribución loca

Che, ¿No era que lo mejor que podíamos lograr en términos de complejidad cuando queríamos hacer ordenamiento era **O(n log n)**?

Primer ejercicio: La distribución loca

Che, ¿No era que lo mejor que podíamos lograr en términos de complejidad cuando queríamos hacer ordenamiento era **O(n log n)**?

- **Sí y no**
- Sí, hablando de algoritmos de ordenamiento comparativos (la demo la vieron en la teórica).
- No, en el caso en el cual dispongamos de información extra sobre la entrada.
- ¿Qué sabemos de los números a ordenar?

Primer ejercicio: Lo importante

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

Vamos al pizarrón

Counting Sort

Este algoritmo se puede usar con cosas más generales que números. Es necesario tener una función S que reciba cosas del tipo de A y devuelva una posición en un arreglo. K representa el valor máximo de la función S sobre todos los elementos del arreglo.

Algorithm 1 COUNTING-SORT(A, B, k)

1: **for** $i \leftarrow [0..k]$ **do**
2: $\quad C[i] \leftarrow 0$
3: **end for**
4: **for** $j \leftarrow [1..length[A]]$ **do**
5: $\quad C[S(A[j])] \leftarrow C[S(A[j])] + 1$
6: **end for**

Counting Sort

Este algoritmo se puede usar con cosas más generales que números. Es necesario tener una función S que reciba cosas del tipo de A y devuelva una posición en un arreglo. K representa el valor máximo de la función S sobre todos los elementos del arreglo.

Algorithm 2 COUNTING-SORT(A, B, k)

1: **for** $i \leftarrow [0..k]$ **do**
2: $\quad C[i] \leftarrow 0$
3: **end for**
4: **for** $j \leftarrow [1..length[A]]$ **do**
5: $\quad C[S(A[j])] \leftarrow C[S(A[j])] + 1$
6: **end for**
7: **for** $i \leftarrow [1..k]$ **do**
8: $\quad C[i] \leftarrow C[i] + C[i-1]$
9: **end for**

Counting Sort

Este algoritmo se puede usar con cosas más generales que números. Es necesario tener una función S que reciba cosas del tipo de A y devuelva una posición en un arreglo. K representa el valor máximo de la función S sobre todos los elementos del arreglo.

Algorithm 3 COUNTING-SORT(A, B, k)

```
1:  for i ← [0..k] do
2:      C[i] ← 0
3:  end for
4:  for j ← [1..length[A]] do
5:      C[S(A[j])] ← C[S(A[j])] + 1
6:  end for
7:  for i ← [1..k] do
8:      C[i] ← C[i] + C[i − 1]
9:  end for
10: for j ← [length[A]..1] do
11:     B[C[S(A[j])]] ← A[j]
12:     C[S(A[j])] ← C[S(A[j])] − 1
13: end for
```

Figure: Pseudo code of the counting sort algorithm [Cormen, p. 148]

- ¿Complejidad?

Counting Sort

Este algoritmo se puede usar con cosas más generales que números. Es necesario tener una función S que reciba cosas del tipo de A y devuelva una posición en un arreglo. K representa el valor máximo de la función S sobre todos los elementos del arreglo.

Algorithm 4 COUNTING-SORT(A, B, k)

```
1:  for i ← [0..k] do
2:      C[i] ← 0
3:  end for
4:  for j ← [1..length[A]] do
5:      C[S(A[j])] ← C[S(A[j])] + 1
6:  end for
7:  for i ← [1..k] do
8:      C[i] ← C[i] + C[i − 1]
9:  end for
10: for j ← [length[A]..1] do
11:     B[C[S(A[j])]] ← A[j]
12:     C[S(A[j])] ← C[S(A[j])] − 1
13: end for
```

Figure: Pseudo code of the counting sort algorithm [Cormen, p. 148]

- ¿Complejidad? O(Max(n,k)) con $n = length(A)$

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 tupla $\langle nombre$: string, $sexo$: FM, $puntaje$: Nota$\rangle$
- donde FM es $enum\{masc, fem\}$
- Nota es un nat no mayor que 10.

Se necesita ordenar un arreglo(alumno) de forma tal que todas las mujeres aparezcan al inicio de la tabla según un orden creciente de notas y todos los varones aparezcan al final ordenados de la misma manera.

Primer ejercicio: Una planilla de notas

Por ejemplo:

Entrada

Ana	*F*	10
Juan	*M*	6
Rita	*F*	6
Paula	*F*	7
Jose	*M*	7
Pedro	*M*	8

Segundo ejercicio: Una planilla de notas

Por ejemplo:

Entrada

Ana	*F*	10
Juan	*M*	6
Rita	*F*	6
Paula	*F*	7
Jose	*M*	7
Pedro	*M*	8

Salida

Rita	*F*	6
Paula	*F*	7
Ana	*F*	10
Juan	*M*	6
Jose	*M*	7
Pedro	*M*	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. Asumir que el largo de los nombres está acotado por una constante. Justificar.

Segundo ejercicio: Lo importante

alumno es tupla$\langle$*nombre*: string, *sexo*: FM, *puntaje*: Nota$\rangle$
donde FM es *enum*$\{$*masc*, *fem*$\}$ y Nota es un nat no mayor que 10.

Entrada

Ana	*F*	10
Juan	*M*	6
Rita	*F*	6
Paula	*F*	7
Jose	*M*	7
Pedro	*M*	8

Salida

Rita	*F*	6
Paula	*F*	7
Ana	*F*	10
Juan	*M*	6
Jose	*M*	7
Pedro	*M*	8

- Complejidad temporal O(n) en el peor caso, donde n es la cantidad de elementos del arreglo. Strings acotados.
- Las mujeres van primero.
- Hay que ordenar por puntaje.

Segundo ejercicio: Una planilla de notas

b) Supongan ahora que la planilla original ya se encuentra ordenada alfabéticamente por nombre. ¿Pueden asegurar que si existen dos o más mujeres con igual nota, entonces las mismas aparecerán en orden alfabético en la planilla ordenada?

Entrada

Ana	*F*	10
Jose	*M*	7
Juan	*M*	6
Marta	*F*	7
Paula	*F*	7
Pedro	*M*	8
Rita	*F*	6

Salida?

Rita	*F*	6
Marta	*F*	7
Paula	*F*	7
Ana	*F*	10
Juan	*M*	6
Jose	*M*	7
Pedro	*M*	8

Salida?

Rita	*F*	6
Paula	*F*	7
Marta	*F*	7
Ana	*F*	10
Juan	*M*	6
Jose	*M*	7
Pedro	*M*	8

Estabilidad

Sorting se hace en base a una relación de orden. Puede pasar que para esa relación dos cosas seas equivalentes.
Si nuestro algoritmo preserva el orden original entre las cosas equivalentes, decimos que es ESTABLE.

Segundo ejercicio: Una planilla de notas

b) Supongan ahora que la planilla original ya se encuentra ordenada alfabéticamente por nombre. ¿Pueden asegurar que si existen dos o más mujeres con igual nota, entonces las mismas aparecerán en orden alfabético en la planilla ordenada?.

Segundo ejercicio: Una planilla de notas

b) Supongan ahora que la planilla original ya se encuentra ordenada alfabéticamente por nombre. ¿Pueden asegurar que si existen dos o más mujeres con igual nota, entonces las mismas aparecerán en orden alfabético en la planilla ordenada?.

Sí Nuestro algoritmo es estable!

Segundo ejercicio: Una planilla de notas

b) Supongan ahora que la planilla original ya se encuentra ordenada alfabéticamente por nombre. ¿Pueden asegurar que si existen dos o más mujeres con igual nota, entonces las mismas aparecerán en orden alfabético en la planilla ordenada?.

Sí Nuestro algoritmo es estable!

c) Dar un algoritmo que ordene igual que antes, pero ante igual sexo y nota, los ordene por orden alfabético. Dar su complejidad y justificar.

Bucket Sort

Idea:

- Separar los elementos en distintas clases que tienen un orden semejante al buscado.
- Ordenar cada uno de los elementos de la clase.
- Concatenar los resultados.

Algorithm 5 BUCKET-SORT(A)

$n \leftarrow length(A)$
$B \leftarrow crearArregloDeListasDeTam(n)$
for $i \leftarrow [1..n]$ **do**
    insert $A[i]$ into list $B[H(A[i])]$ //*H mandaA*$[i]$ *a su bucket correspondiente*.
end for
for $i \leftarrow [0..n-1]$ **do**
    sort list $B[i]$
end for
concatenate lists $B[0]$,$B[1]$, ..., $B[n-1]$ together

Figure: Pseudocódigo del algoritmo bucket sort [Cormen, p. 153]

- ¿Complejidad?

Bucket Sort

Idea:

- Separar los elementos en distintas clases que tienen un orden semejante al buscado.
- Ordenar cada uno de los elementos de la clase.
- Concatenar los resultados.

Algorithm 6 BUCKET-SORT(A)

$n \leftarrow length(A)$
$B \leftarrow crearArregloDeListasDeTam(n)$
for $i \leftarrow [1..n]$ **do**
  insert $A[i]$ into list $B[H(A[i])]$ //*H mandaA*$[i]$ *a su bucket correspondiente*.
end for
for $i \leftarrow [0..n-1]$ **do**
  sort list $B[i]$
end for
concatenate lists $B[0]$,$B[1]$, ..., $B[n-1]$ together

Figure: Pseudocódigo del algoritmo bucket sort [Cormen, p. 153]

- ¿Complejidad? O(Max(n,M))
  con $n = length(A)$ y $M =$ cantidad de buckets

RADIX Sort

- Está pensado para cadenas de caracteres o tuplas, donde el orden que se quiera dar dependa principalmente de un "dígito", y en el caso de empate se tengan que revisar los otros.
- La idea es usar un algoritmo estable e ir ordenado por dígito.
- Por lo tanto, se ordena del dígito menos significativo al más importante.

Algorithm 7 RADIX-SORT(A, d)

for $i \leftarrow [1..d]$ **do**
    stable sort array A on digit i
end for

Figure: Pseudocódigo del algoritmo RADIX sort [Cormen, p. 148]

Segundo ejercicio: Conclusiones

- La cota **O(n log n)** corresponde a los algoritmos de ordenamiento basados en comparación.

Segundo ejercicio: Conclusiones

- La cota **O(n log n)** corresponde a los algoritmos de ordenamiento basados en comparación.
- El algoritmo que implementamos es **estable**.

Segundo ejercicio: Conclusiones

- La cota **O(n log n)** corresponde a los algoritmos de ordenamiento basados en comparación.
- El algoritmo que implementamos es **estable**.
- ¿Quieren ordenar tuplas, strings, números? **RADIX Sort** puede servir.

Segundo ejercicio: Conclusiones

- La cota **O(n log n)** corresponde a los algoritmos de ordenamiento basados en comparación.
- El algoritmo que implementamos es **estable**.
- ¿Quieren ordenar tuplas, strings, números? **RADIX Sort** puede servir. PERO es muy importante poder tratar a la entrada como una lista o tupla.

Segundo ejercicio: Conclusiones

- La cota **O(n log n)** corresponde a los algoritmos de ordenamiento basados en comparación.
- El algoritmo que implementamos es **estable**.
- ¿Quieren ordenar tuplas, strings, números? **RADIX Sort** puede servir. PERO es muy importante poder tratar a la entrada como una lista o tupla. "Sirve" cuando la cantidad de elementos a ordenar es muy grande en comparación al tamaño de cada elemento en particular.

Segundo ejercicio: Conclusiones

- La cota **O(n log n)** corresponde a los algoritmos de ordenamiento basados en comparación.
- El algoritmo que implementamos es **estable**.
- ¿Quieren ordenar tuplas, strings, números? **RADIX Sort** puede servir. PERO es muy importante poder tratar a la entrada como una lista o tupla. "Sirve" cuando la cantidad de elementos a ordenar es muy grande en comparación al tamaño de cada elemento en particular.
- Los algoritmos estables se pueden usar juntos sin muchos problemas.

Segundo ejercicio: Conclusiones

- La cota **O(n log n)** corresponde a los algoritmos de ordenamiento basados en comparación.
- El algoritmo que implementamos es **estable**.
- ¿Quieren ordenar tuplas, strings, números? **RADIX Sort** puede servir. PERO es muy importante poder tratar a la entrada como una lista o tupla. "Sirve" cuando la cantidad de elementos a ordenar es muy grande en comparación al tamaño de cada elemento en particular.
- Los algoritmos estables se pueden usar juntos sin muchos problemas.
- Si tenemos una jerarquía para ordenar tenemos que ordenar primero por el criterio menos importante e ir subiendo.

Estabilidad, la revancha

- Bucket Sort es estable, si los algoritmos que se usan para cada balde son estables.

Estabilidad, la revancha

- Bucket Sort es estable, si los algoritmos que se usan para cada balde son estables.
- RADIX Sort es estable, aunque depende del algoritmo subyacente.

Estabilidad, la revancha

- Bucket Sort es estable, si los algoritmos que se usan para cada balde son estables.
- RADIX Sort es estable, aunque depende del algoritmo subyacente.
- Counting Sort es estable, bien programado.

Tercer ejercicio: Ejercicio 18

- Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n en tiempo $O(n)$.

Tercer ejercicio: Ejercicio 18

- Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n en tiempo $O(n)$.
- CountingSort(A, n, n) //Complejidad: O(n+n) = O(n)

Tercer ejercicio: Ejercicio 18

- 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.

Tercer ejercicio: Ejercicio 18

- 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.
- RadixSort(A, n, n) //A: arreglo de entrada, n: cantidad de elementos, m: radio o base de los elementos

Tercer ejercicio: Ejercicio 18

- RadixSort(A, n, m) //A: arreglo de entrada, n: cantidad de elementos, m: radio o base de los elementos

Algorithm 8 RadixSort(A, n, m)

MaxDigitos $\leftarrow \log_m Max(A)$
B $\leftarrow$ *crearArregloDeListasDeTam*(n)
for $i \leftarrow [0..n-1]$ **do**
  $B[i] \leftarrow$ *descomposición en base m de* $A[i]$ // Los números que tienen menos dígitos se padean a izquierda con 0.
end for
for $j \leftarrow [0..MaxDigitos]$ **do**
  CountinSort(B, m-1, j)
end for
return volverNumerosASuBase(B)

Tercer ejercicio: Ejercicio 18

- Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n^k para una constante arbitraria pero fija k.
- ¿Qué complejidad se obtiene si se generaliza la idea para arreglos de enteros no acotados?

Tercer ejercicio: Ejercicio 18

- Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n^k para una constante arbitraria pero fija k.
- ¿Qué complejidad se obtiene si se generaliza la idea para arreglos de enteros no acotados?

Usamos el mismo algoritmo

- ¿Complejidad?

Tercer ejercicio: Ejercicio 18

- Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n^k para una constante arbitraria pero fija k.
- ¿Qué complejidad se obtiene si se generaliza la idea para arreglos de enteros no acotados?

Usamos el mismo algoritmo

- ¿Complejidad? O((n+m) * [log_m(Máx(A)) + 1])

Cuarto ejercicio: Chan chan

Se tienen k arreglos de naturales $A_0, ..., A_{k-1}$. Cada uno de ellos está ordenado de forma creciente. Se sabe que ningún natural está dos veces en el mismo arreglo y no aparece el mismo natural en dos arreglos distintos. Además, se sabe que para todo i el arreglo A_i tiene exactamente 2^i elementos.

Dar un algoritmo que devuelva un arreglo B de $n = \sum_{i=0}^{k-1} 2^i = 2^k - 1$ elementos, ordenado crecientemente, de manera que un natural está en B si y sólo si está en algún A_i (o sea, B es la unión de los A_i y está ordenado).

Cuarto ejercicio: Chan chan

a) Escribir el pseudocódigo de un algoritmo que resuelva el problema planteado. El algoritmo debe ser de tiempo lineal en la cantidad de elementos en total de la entrada, es decir $O(n)$.

b) Calcular y justificar la complejidad del algoritmo propuesto.

Conclusiones

- Es importante justificar de manera correcta la complejidad, ya que es condición necesaria para la aprobación del ejercicio (¡Y en la vida está bueno saber que lo que uno usa se comporta como se cree!).
- Consulten.
- Hagan las prácticas.
- Estudien.

Preguntas?

Figure: “¿La policía sabía que asuntos internos le tendía una trampa?”

Algunas cosas para leer, ver y/o escuchar

- T. H. Cormen, C. E. Leiserson, R.L Rivest, and C. Stein. **"Introduction to Algorithms"**. MIT Press, 2nd edition edition, August 2001.
- Un paper donde tratan las complejidades de los algoritmos vistos en clase. Además, los explican de manera clara. http://arxiv.org/pdf/1206.3511.pdf
- Canal de YouTube donde se mezclan danzas clásicas y los algoritmos de Sorting más comunes (IMPERDIBLE) http://www.youtube.com/user/AlgoRythmics
- Otros "algoritmos de Sorting" (o no tan sorting) http://xkcd.com/1185/
- GRAN banda under argentina http://www.monstruito.com.ar/