Ordenamiento - Sorting - Sortin'

Christian Russo

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

9 de Octubre de 2015

Algunas aclaraciones

- Este tipo de ejercicios requieren práctica y paciencia.

Algunas aclaraciones

- Este tipo de ejercicios requieren práctica y paciencia.
- También requieren que se sepa un poco de estructuras de datos.

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).

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 deberá estar acompañada de la correspondiente complejidad temporal.

Algunas estrategias

- Utilizar algoritmos ya conocidos (Merge Sort, Quick Sort, etc.)

Algunas estrategias

- Utilizar algoritmos ya conocidos (Merge Sort, Quick Sort, etc.)
- Intuir algo de la complejidad pedida.

Algunas estrategias

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

Un pequeño repaso

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

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. Son n elementos e insertar ordenado cuesta $O(n)$
- Selection Sort, $O(n^2)$

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. 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 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.
- Bubble Sort, $O(n^2)$

  Semejante a Selection Sort, solo que mientras busca el menor va moviendo su candidato a menor para adelante.
- MergeSort, $O(n \log n)$

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. 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 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.
- Bubble Sort, $O(n^2)$

  Semejante a Selection Sort, solo que mientras busca el menor va moviendo su candidato a menor para adelante.
- MergeSort, $O(n \log n)$

  Algoritmo recursivo que ordena sus dos mitades y luego las fusiona.
- QuickSort, $O(n^2)$

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. 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 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.
- Bubble Sort, $O(n^2)$

  Semejante a Selection Sort, solo que mientras busca el menor va moviendo su candidato a menor para adelante.
- MergeSort, $O(n \log n)$

  Algoritmo recursivo que ordena sus dos mitades y luego las fusiona.
- QuickSort, $O(n^2)$

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

Un pequeño repaso

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

  Se va insertando en el lugar correspondiente cada elemento en un arreglo ordenado. 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 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.
- Bubble Sort, $O(n^2)$

  Semejante a Selection Sort, solo que mientras busca el menor va moviendo su candidato a menor para adelante.
- MergeSort, $O(n \log n)$

  Algoritmo recursivo que ordena sus dos mitades y luego las fusiona.
- QuickSort, $O(n^2)$

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

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

Primer 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*$\}$ y 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

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

Salida

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

Primer 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.

Primer ejercicio: Una planilla de notas

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

Primer ejercicio: Una planilla de notas

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.

Primer ejercicio: Una planilla de notas

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.

> 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**.

- Vamos al pizarrón...

Primer 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.

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 1 BUCKET-SORT(A)

$n \leftarrow length[A]$
for $i \leftarrow [1..n]$ **do**
    insert $A[i]$ into list $B[n \cdot A[i]/M]$
end for
for $i \leftarrow [0..n-1]$ **do**
    sort list $B[i]$ with insertion sort
end for
concatenate lists $B[0], B[1], \ldots, B[n-1]$ together

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

Primer 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?

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.

Primer 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?.

Primer 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!

Primer 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.

Pista: Por lo visto en b), podríamos usar el algoritmo de a).

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 2 RADIX-SORT(A, d)

```
for i ← [1..d] do
    (stable) sort array A on digit i
end for
```

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

Primer ejercicio: Una planilla de notas

d) Si en los buckets de la parte a) hubieramos ordenado las cadenas de la misma manera, ¿La complejidad hubiese cambiado?

Primer ejercicio: Una planilla de notas

d) Si en los buckets de la parte a) hubieramos ordenado las cadenas de la misma manera, ¿La complejidad hubiese cambiado?

No! La complejidad de RADIX Sort es lineal sobre la cantidad de cadena, si el largo de las cadenas y sus digitos están acotados.

Primer ejercicio: Conclusiones

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

Primer ejercicio: Conclusiones

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

Primer 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.

Primer 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.

Primer 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.

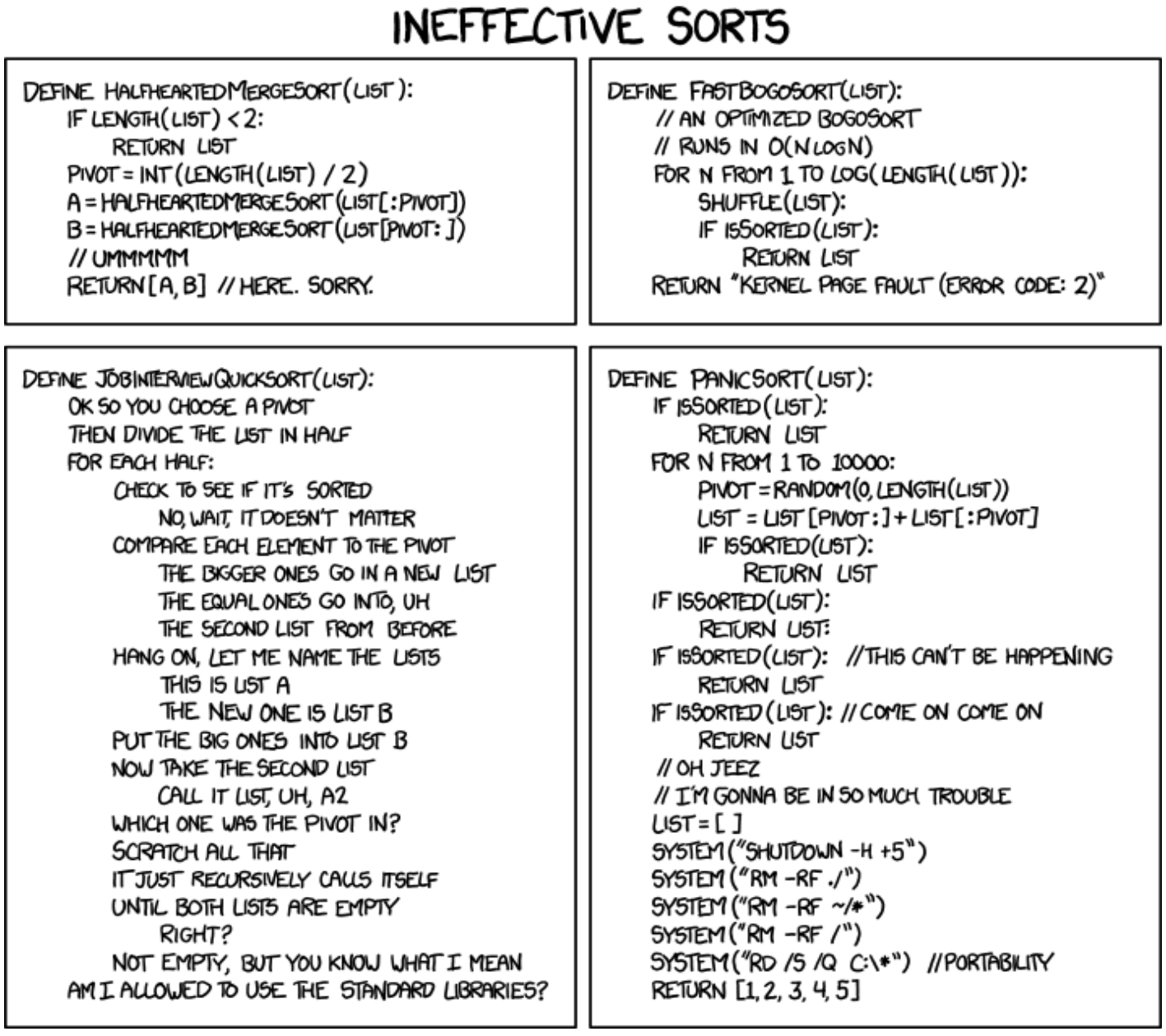
Primer ejercicio: Conclusiones

- 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.

Primer 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.

Recreo!


INEFFECTIVE SORTS
DEFINE HALFHEARTEDMERGESORT(LIST):
IF LENGTH(LIST) < 2:
RETURN LIST
PIVOT = INT(LENGTH(LIST) / 2)
A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
// UMMMMM
RETURN [A, B] // HERE. SORRY.
DEFINE FASTBOGOSORT(LIST):
// AN OPTIMIZED BOGOSORT
// RUNS IN O(N LOG N)
FOR N FROM 1 TO LOG(LENGTH(LIST)):
SHUFFLE(LIST):
IF ISSORTED(LIST):
RETURN LIST
RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
DEFINE JOBINTERVIEWQUICKSORT(LIST):
OK SO YOU CHOOSE A PIVOT
THEN DIVIDE THE LIST IN HALF
FOR EACH HALF:
CHECK TO SEE IF IT'S SORTED
NO, WAIT, IT DOESN'T MATTER
COMPARE EACH ELEMENT TO THE PIVOT
THE BIGGER ONES GO IN A NEW LIST
THE EQUAL ONES GO INTO, UH
THE SECOND LIST FROM BEFORE
HANG ON, LET ME NAME THE LISTS
THIS IS LIST A
THE NEW ONE IS LIST B
PUT THE BIG ONES INTO LIST B
NOW TAKE THE SECOND LIST
CALL IT LIST, UH, A2
WHICH ONE WAS THE PIVOT IN?
SCRATCH ALL THAT
IT JUST RECURSIVELY CALLS ITSELF
UNTIL BOTH LISTS ARE EMPTY
RIGHT?
NOT EMPTY, BUT YOU KNOW WHAT I MEAN
AM I ALLOWED TO USE THE STANDARD LIBRARIES?
DEFINE PANICSORT(LIST):
IF ISSORTED(LIST):
RETURN LIST
FOR N FROM 1 TO 10000:
PIVOT = RANDOM(0, LENGTH(LIST))
LIST = LIST[PIVOT:] + LIST[:PIVOT]
IF ISSORTED(LIST):
RETURN LIST
IF ISSORTED(LIST):
RETURN LIST:
IF ISSORTED(LIST): //THIS CAN'T BE HAPPENING
RETURN LIST
IF ISSORTED(LIST): // COME ON COME ON
RETURN LIST
// OH JEEZ
// I'M GONNA BE IN SO MUCH TROUBLE
LIST = [ ]
SYSTEM("SHUTDOWN -H +5")
SYSTEM("RM -RF ./")
SYSTEM("RM -RF ~/*")
SYSTEM("RM -RF /")
SYSTEM("RD /S /Q C:*") //PORTABILITY
RETURN [1, 2, 3, 4, 5]

Segundo 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.

Segundo 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.

Vamos al pizarrón...

Segundo ejercicio: Lo importante

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

Segundo 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.

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.

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
```

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

Tercer 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).

Tercer 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.

Tercer ejercicio: Lo importante

- k arreglos de naturales $A_0, ..., A_{k-1}$. Cada uno de ellos está ordenado de forma creciente.
- Dado un arreglo A_i, este tiene 2^i elementos.
- Si un número está en un arreglo de la entrada, no está los demás.
- Se nos pide $O(n)$, donde $n = \sum_{i=0}^{k-1} 2^i = 2^k - 1$ (**importante**).

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?

Figura: “¿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/