Ordenamiento - Sorting

Algoritmos y Estructuras de Datos 2

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

4 de junio de 2018

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.
- 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.
- 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.

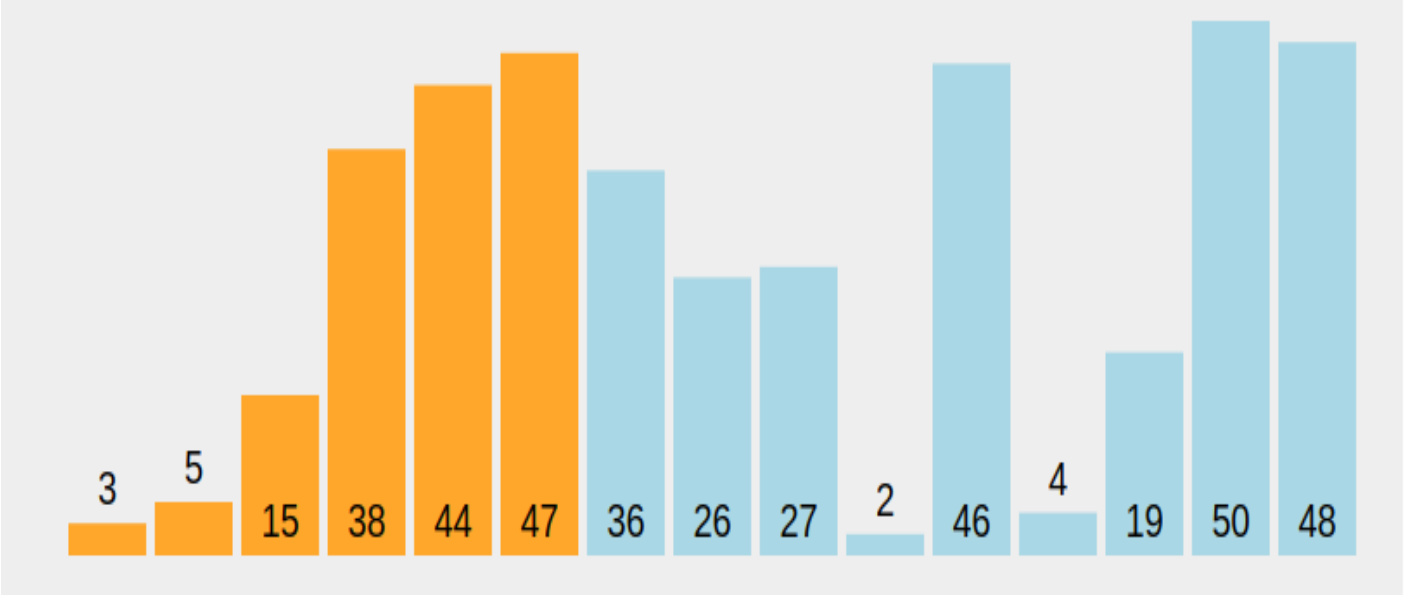
  Va *burbujeando* los elementos de a pares, llevando el más grande hacia adelante, cuando encuentra uno más grande burbujea ese.
- MergeSort, $\Theta(n \log n)$

  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 de lo mejor que hay en la práctica.
- HeapSort, $\Theta(n + n \log n)) = O(n \log n)$

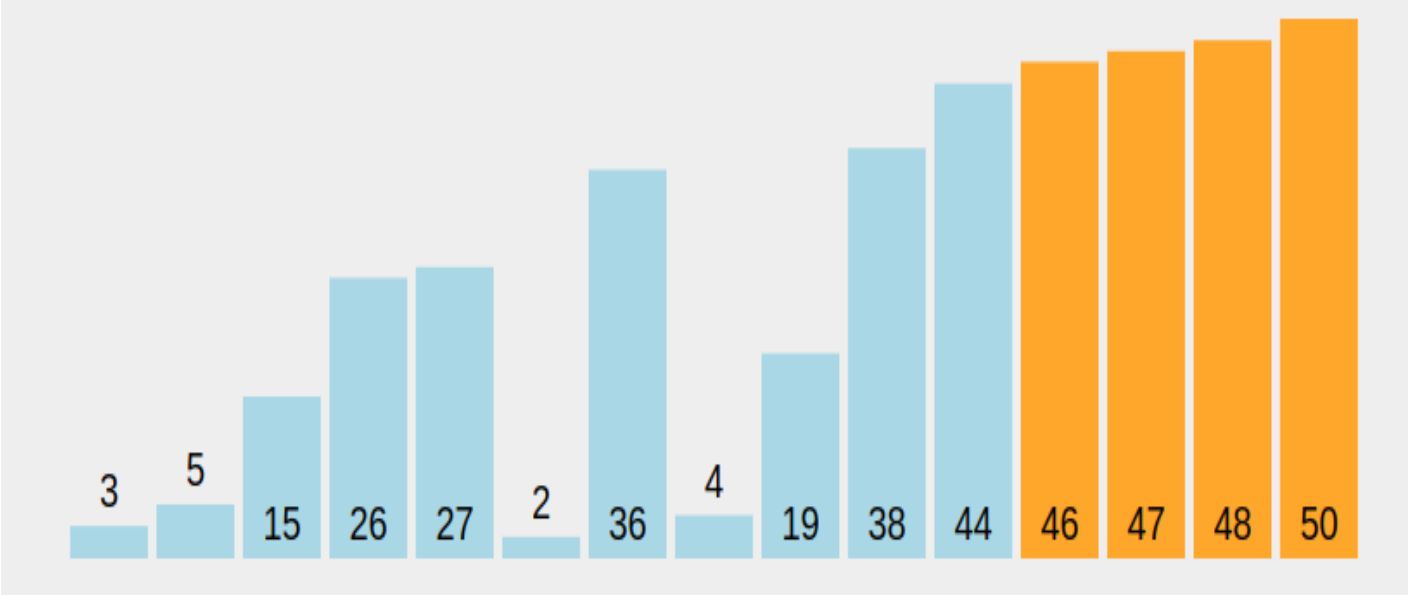
  Arma el Heap en $O(n)$ y va sacando los elementos ordenados pagando $O(\log n)$.

Trivia - Adivinen qué algoritmo se está usando



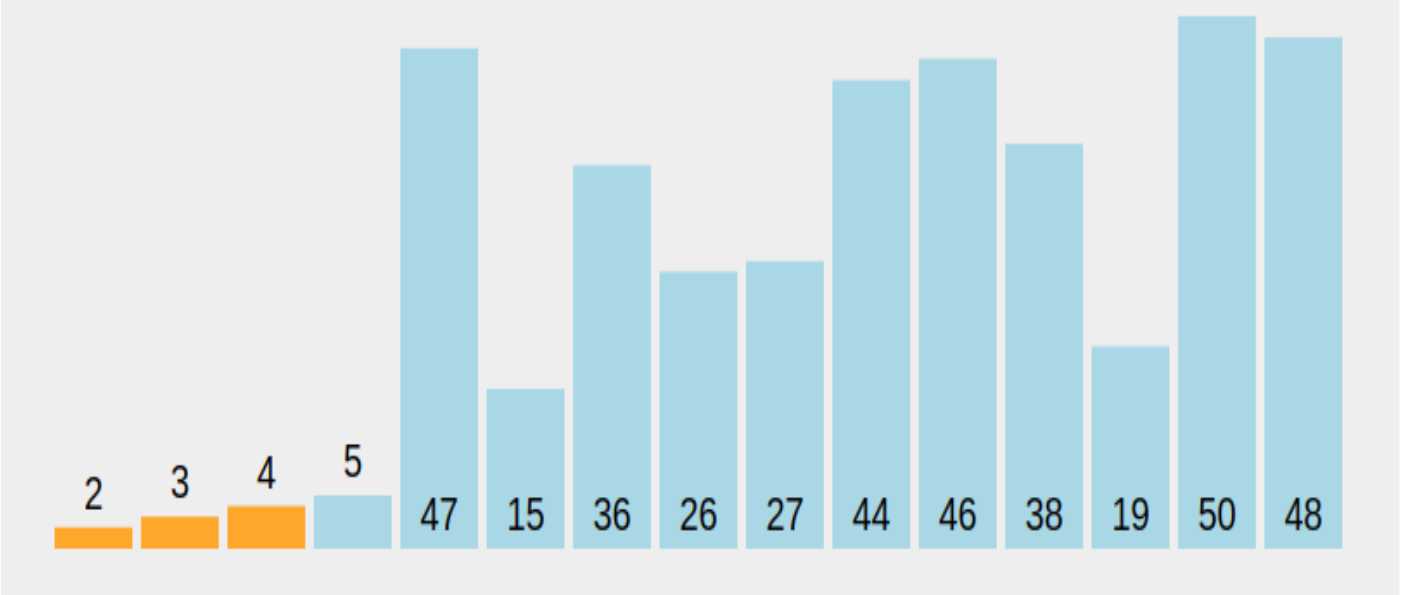
Insertion Sort

Trivia - Adivinen qué algoritmo se está usando



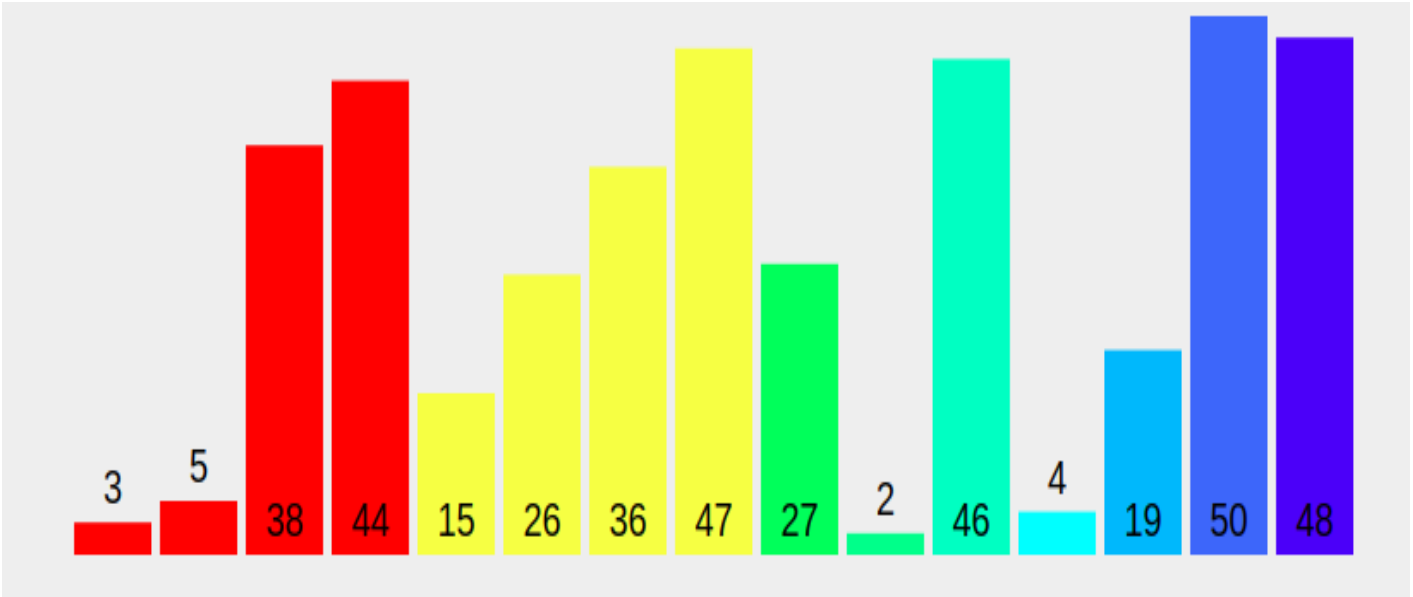
Bubble Sort

Trivia - Adivinen qué algoritmo se está usando



Selection Sort

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.

Counting Sort

- **Counting Sort** asume que los elementos de un arreglo A de naturales están acotados por 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, podría tomarse k como el máximo valor de A.

Counting Sort

k representa el valor máximo del arreglo.

Algorithm 1 COUNTING-SORT(A: arreglo(nat), k: nat)

```
1:  B ← arreglo de tamaño k + 1
2:  for i ← [0..k] 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] do
12:     for j ← [1..B[i]] do
13:         A[indexA] ← i
14:         indexA + +
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)

Bucket Sort

- **Bucket Sort** ordena arreglos de cualquier tipo.
- Supone que los elementos pueden separarse (según algún criterio) en M categorías ordenadas.
- Es decir, si $i < j$, 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.
  3. Reconstruir el arreglo A, concatenando las listas en orden.
- **Observación:** Cuando la distribución en categorías es uniforme, se puede esperar que el paso 2 sea rápido.
- **Observación:** Si se omite el paso 2 es una suerte de Counting Sort "generico" (para arreglos de cualquier tipo).

Bucket Sort

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

Algorithm 2 BUCKET-SORT(A)

$n \leftarrow length(A)$
$B \leftarrow$ 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... $O(n+M)$.

Radix Sort (versión para ordenar naturales)

- **Radix Sort**, en su versión más básica, ordena una lista de 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).
- 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 intuir algo de eso.
- Algo de ingenio para resolver problemas...

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 $O(n)$

Discutamosló...

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) para que todas las mujeres aparezcan al inicio según un orden creciente de notas y todos los varones al final con el mismo orden. 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. 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.
- Las mujeres van primero.
- Hay que ordenar por puntaje.

Segundo ejercicio: Una planilla de notas

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

Salida?		
Rita	*F*	6
Marta	*F*	7
Paula	*F*	7
Ana	*F*	10
Juan	*M*	6
Jose	*M*	7

Salida?		
Rita	*F*	6
Paula	*F*	7
Marta	*F*	7
Ana	*F*	10
Juan	*M*	6
Jose	*M*	7

b) Si la planilla original estaba ordenada alfabéticamente por nombre. ¿Podemos 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) ¿Qué habría que modificar de nuestro algoritmo para que ordene igual que antes pero ante igual sexo 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" 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, n, m)

$d \leftarrow \log_m Max(A)$
$B \leftarrow$ arreglo de n arreglos de tamaño d
for $i \leftarrow [0..n-1]$ **do**
  $B[i] \leftarrow$ descomposición en base m 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)

- ¿Complejidad? $O(n \cdot d) + d \cdot O(n + m) = O(d \cdot (n + m))$, con $d = \log_m(\max(A))$
- **Observación:** ¡mientras mayor sea la base, menor será la cantidad de dígitos necesaria!

Tercer ejercicio: Ejercicio 18

1. Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n en tiempo $O(n)$.
   - CountingSort...
   - Complejidad: $O(n + n) = O(n)$
2. Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n^2 en tiempo $O(n)$. Pista: Usar varias llamadas al ítem anterior.
   - ¡RadixSort en base n!
   - Complejidad: $O(\log_n(\max(A)) \cdot (n + n)) = O(4n) = O(n)$

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?
  $O(\log_n(\max(A)) \cdot (n + n)) = O(k \cdot 2n) = O(k \cdot n)$

Cuarto ejercicio: Chan chan

Se tienen k arreglos de naturales $A_0, ..., A_{k-1}$, donde A_i tiene exactamente 2^i elementos.

Dar un algoritmo que combine todos los elementos de los arreglos en un arreglo B de tamaño $n = \sum_{i=0}^{k-1} 2^i = 2^k - 1$, ordenado crecientemente.

1. 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 $O(n)$.
2. 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/