¿Es posible ordenar más rápidamente que O(*n* log *n*)?

¿Es posible ordenar más rápidamente que O(*n* log *n*)?

No

¿Es posible ordenar más rápidamente que O(*n* log *n*)?

No,

… si la única información usada por el algoritmo es el resultado de comparar, repetidamente, dos datos y determinar cuál es mayor:

- cada comparación puede producir dos resultados
- es un algoritmo de *ordenación por comparación* —todos los algoritmos de ordenación que ya estudiamos son por comparación

¿Cómo podemos demostrar la afirmación anterior?

El conjunto de comparaciones es un árbol binario completo —un **árbol de decisión**:

- cada nodo interno representa una comparación entre dos datos
- cada hoja representa una permutación de los datos que están siendo ordenados
- ver ejemplo en la pizarra para la secuencia $<a_0, a_1, a_2>$

Un algoritmo de ordenación por comparación sigue una ruta en el árbol

… desde la raíz hasta una hoja:

el desempeño del algoritmo en el peor caso corresponde a seguir la ruta más larga

El árbol para ordenar n datos tiene $n!$ hojas

Como es un árbol binario, la longitud h de la ruta más larga cumple con $2^h \geq n!$

Luego, $h \geq \log n! = \Omega(n \log n)$

`countingSort`: Un algoritmo de ordenación que no compara los datos que está ordenando

Suponemos que cada uno de los *n* datos es un entero en el rango 0 a *k*, con k entero

… esta es información con la que no contábamos antes:

si *k* es O(*n*), entonces *countingSort* corre en tiempo Θ(*n*)

Determinamos, para cada dato *x*, el número de datos menores que *x*:

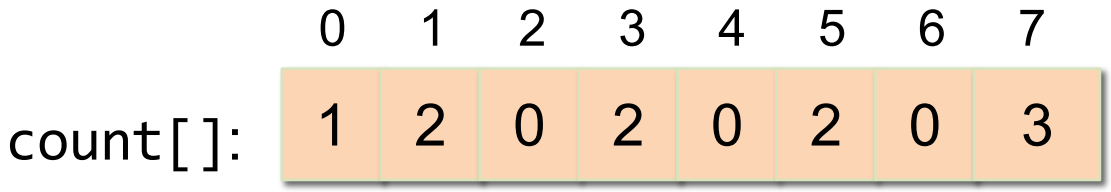
- esto permite ubicar a *x* directamente en su posición final en el arreglo de salida
- hay que manejar el caso en que varios datos tengan el mismo valor

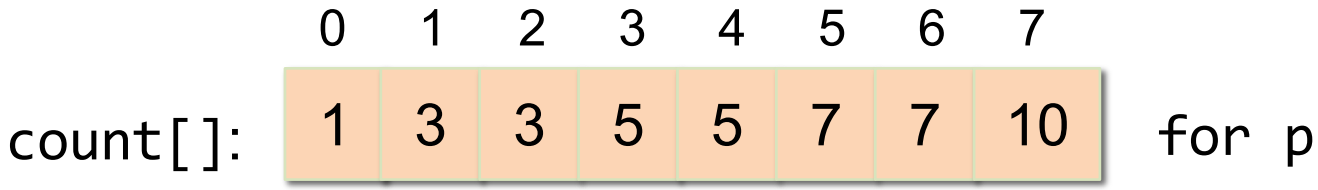
```
countingSort(data, tmp, k, n):
    sea count[0..k] un nuevo arreglo
    for i = 0 … k:
        count[i] = 0
    for j = 1 … n:
        count[data[j]] = count[data[j]]+1
    for p = 1 … k:
        count[p] = count[p]+count[p-1]
    for r = n … 1:
        tmp[count[data[r]]] = data[r]
        count[data[r]] = count[data[r]]-1
```

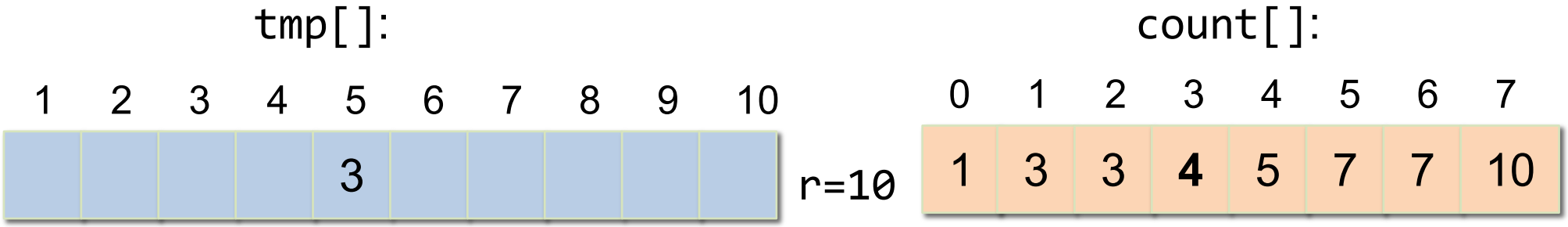
Este algoritmo es (claramente) $\Theta(k+n)$

Si k es $O(n)$, entonces `countingSort` es $\Theta(n)$


1 2 3 4 5 6 7 8 9 10
data[]:
7 1 1 3 0 7 5 5 7 3


0 1 2 3 4 5 6 7
count[]:
1 2 0 2 0 2 0 3
for j


0 1 2 3 4 5 6 7
count[]:
1 3 3 5 5 7 7 10
for p


tmp[]:
1 2 3 4 5 6 7 8 9 10
3
r=10
count[]:
0 1 2 3 4 5 6 7
1 3 3 4 5 7 7 10

1	2	3	4	5	6	7	8	9	10		0	1	2	3	4	5	6	7
				3						r=10	1	3	3	**4**	5	7	7	10
				3					7	r=9	1	3	3	4	5	7	7	**9**
				3		5			7	r=8	1	3	3	4	5	**6**	7	9
				3	5	5			7	r=7	1	3	3	4	5	**5**	7	9
				3	5	5		7	7	r=6	1	3	3	4	5	5	7	**8**
0				3	5	5		7	7	r=5	**0**	3	3	4	5	5	7	8
0			3	3	5	5		7	7	r=4	0	3	3	**3**	5	5	7	8
0		1	3	3	5	5		7	7	r=3	0	**2**	3	3	5	5	7	8
0	1	1	3	3	5	5	7	7	7	r=2	0	**1**	3	3	5	5	7	8
0	1	1	3	3	5	5	7	7	7	r=1	0	1	3	3	5	5	7	**7**

radixSort era el algoritmo usado por las máquinas que ordenaban tarjetas perforadas

Cada tarjeta tiene 80 columnas

… en cada columna se puede perforar un hoyo en uno de 12 lugares

La máquina se programa para examinar una determinada columna de cada tarjeta y distribuir la tarjeta en uno de 12 compartimientos, dependiendo de la perforación

radixSort era el algoritmo usado por las máquinas que ordenaban tarjetas perforadas

Una persona recolecta las tarjetas de cada compartimiento

… de modo que las tarjetas con la perforación en el primer lugar quedan encima de las tarjetas con la perforación en el segundo lugar, etc.

Un número de *d*-dígitos ocupa *d* columnas

Como la máquina mira sólo una columna a la vez

… ordenar *n* tarjetas según un número de *d*-dígitos requiere un algorirtmo de ordenación

Podríamos ordenar los números según su dígito más significativo,

… luego ordenar recursivamente cada compartimiento,

… y finalmente combinar los contenidos de cada compartimiento:

> para ordenar recursivamente cada compartimiento, hay que poner a un lado los contenidos de los otros nueve

radixSort ordena *según el dígito menos significativo primero*

Luego, las tarjetas son combinadas de modo que las que vienen del compatimiento 0 quedan arriba de las que vienen del 1, éstas quedan arriba de las que vienen del 2, etc.

Luego, todas las tarjetas son ordenadas nuevamente, ahora según el segundo dígito menos significativo, y recombinadas similarmente

El proceso sigue hasta que las tarjetas han sido ordenadas según los *d* dígitos

En este punto, **las tarjetas están totalmente ordenadas según el número de *d* dígitos**:

se necesita sólo *d* pasadas por todas las tarjetas

Arreglo inicial	Ordenado por dígito 1s	Ordenado por dígito 10s	Ordenado por dígito 100s
0 6 4	0 0 **0**	0 **0** 0	**0** 0 0
0 0 8	0 0 **1**	0 **0** 1	**0** 0 1
2 1 6	5 1 **2**	0 **0** 8	**0** 0 8
5 1 2	3 4 **3**	5 **1** 2	**0** 2 7
0 2 7	0 6 **4**	2 **1** 6	**0** 6 4
7 2 9	1 2 **5**	1 **2** 5	**1** 2 5
0 0 0	2 1 **6**	0 **2** 7	**2** 1 6
0 0 1	0 2 **7**	7 **2** 9	**3** 4 3
3 4 3	0 0 **8**	3 **4** 3	**5** 1 2
1 2 5	7 2 **9**	0 **6** 4	**7** 2 9

La ordenación por dígito *debe ser estable*

```
void radixSort(a, d)
    for j = 1 … d:
        usando una ordenación estable,
        ordene el arreglo a según el dígito j
```

Si **a** contiene *n* números de *d* dígitos,

… en que cada dígito puede tomar hasta *k* valores posibles,

… entonces *radixSort* toma tiempo $\Theta(d(n+k))$ en ordenar los *n* números:

si *d* es constante y $k = O(n)$, entonces *radixSort* es $\Theta(\boldsymbol{n})$

El algoritmo es útil para ordenar strings, cuando todos son del mismo largo: *LSD string sort*

P.ej.,

- patentes de automóviles
- números telefónicos
- direcciones IP

Además, el largo de los strings debe ser más bien pequeño

MSD string sort : Strings de largos diferentes

Usamos `countingSort` para ordenar los strings según el primer carácter

… luego, recursivamente, ordenamos los subarreglos correspondientes a cada carácter (excluyendo el primer carácter, que es el mismo para cada string en el subarreglo)

Así como *quicksort*, *MSD string sort* particiona el arreglo en sub-arreglos que pueden ser ordenados independientemente,

… pero lo particiona en **un subarreglo para cada posible valor del primer carácter**, en lugar de las dos particiones de *quicksort*

she	**a**re	are	are	…	are
sells	**b**y	by	by		by
seashells	**s**he	s**e**lls	se**a**shells		sea
by	**s**ells	s**e**ashells	se**a**		seashells
the	**s**eashells	s**e**a	se**a**shells		seashells
sea	**s**ea	s**e**lls	s**e**lls		sells
shore	**s**hore	s**e**ashells	s**e**lls		sells
the	**s**hells	s**h**e	she		she
shells	**s**he	s**h**ore	shore		she
she	**s**ells	s**h**ells	shells		shells
sells	**s**urely	s**h**e	she		shore
are	**s**eashells	s**u**rely	surely		surely
surely	**t**he	the	the		the
seashells	**t**he	the	the		the

Cuidados

Fin del string:

- “she” es menor que “shells”

Alfabeto:

- binario (2), minúsculas (26), minúsculas + mayúsculas + dígitos (64), ASCII (128), Unicode (65,536)

Subarreglos pequeños:

- p.ej., tamaño ≤ 10
- cambiar a un *insertionSort* que sepa que los *p* primeros caracteres de los strings que está ordenando son iguales