Sorts en tiempo O(*n*)

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

Suponemos que cada uno de los *n* datos es un número 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 $\Theta(n)$

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):
    sea count[0..k] un nuevo arreglo
    n = data.length
    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
				3					
				3					7
				3		5			7
				3	5	5			7
				3	5	5		7	7
0				3	5	5		7	7
0			3	3	5	5		7	7
0		1	3	3	5	5		7	7
0	1	1	3	3	5	5	7	7	7
0	1	1	3	3	5	5	7	7	7

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

radixSort : algoritmo usado por las máquinas que ordenaban tarjetas perforadas

Cada tarjeta tiene 80 columnas y 12 líneas; en cada columna se puede perforar un hoyo en una de las 12 líneas

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

Una persona recolecta las tarjetas de cada compartimiento, de modo que las tarjetas con la perforación en la primera línea quedan encima de las tarjetas con la perforación en la segunda línea, 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

Procedimiento “natural”

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 compartimiento 0 quedan arriba de las que vienen del compartimiento 1, éstas quedan arriba de las que vienen del compartimiento 2, etc.

Luego, la totalidad de las tarjetas es ordenada 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

radixsort en acción

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***

```
radixSort(a, d):
    for j = 1 … d:
        usando una ordenación estable,
        ordenar 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 $\boldsymbol{\Theta(n)}$

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 subarreglos 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*

MSD string sort en acción

she	are	are	are	...	are
sells	by	by	by		by
seashells	she	sells	seashells		sea
by	sells	seashells	sea		seashells
the	seashells	sea	seashells		seashells
sea	sea	sells	sells		sells
shore	shore	seashells	sells		sells
the	shells	she	she		she
shells	she	shore	shore		she
she	sells	shells	shells		shells
sells	surely	she	she		shore
are	seashells	surely	surely		surely
surely	the	the	the		the
seashells	the	the	the		the

Precauciones

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

- https://www.youtube.com/watch?v=7zuGmKfUt7s
- https://www.youtube.com/watch?v=nu4gDuFabIM