Tres algoritmos de ordenación no tan simples

heapSort:

- tiempo O(*n*log*n*) en el peor caso —**ya estudiado**

mergeSort:

- tiempo O(*n*log*n*) en el peor caso
- necesita O(*n*) memoria extra

quickSort:

- tiempo O(n^2) en el peor caso
- tiempo O(*n*log*n*) en el caso promedio
- en la práctica, se desempeña mejor que *heapSort*

`mergeSort` aplica la estrategia dividir-para-conquistar

Dividir:

dividir la secuencia de *n* datos en dos subsecuencias de *n*/2 datos cada una

Conquistar:

ordenar las dos subsecuencias recursivamente usando *mergeSort*

Combinar:

mezclar las dos subsecuencias ya ordenadas, para producir la permutación ordenada de la secuencia original


dividir
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
29 5 3 59 19 43 17 13 47 53 31 2 11 37 23 7
mezclar
5 29 3 59 19 43 13 17 47 53 2 31 11 37 7 23
3 5 29 59 13 17 19 43 2 31 47 53 7 11 23 37
3 5 13 17 19 29 43 59 2 7 11 23 31 47 37 53
2 3 5 7 11 13 17 19 23 29 31 37 43 47 53 59

Ordenamos el arreglo r llamando a mergeSort(r)

```
mergeSort(a):
    tmp = a
    mergeSort(a, tmp, 0, a.length-1)

mergeSort(a, tmp, e, w):
    if e < w:
        int m = (e+w)/2
        mergeSort(a, tmp, e, m)
        mergeSort(a, tmp, m+1, w)
        merge(a, tmp, e, m+1, w)
```

merge mezcla los subarreglos ordenados a[e..m-1] y a[m..w] en el subarreglo ordenado a[e..w]

```
merge(a, tmp, e, m, w):
    p = e, k = e, q = m
    while p <= m-1 && q <= w:
        if a[p].key < a[q].key:
            tmp[k] = a[p]; k = k+1; p = p+1
        else:
            tmp[k] = a[q]; k = k+1; q = q+1
    while p <= m-1:
        tmp[k] = a[p]; k = k+1; p = p+1
    while q <= w:
        tmp[k] = a[q]; k = k+1; q = q+1
    for k = e; k <= w; k=k+1:
        a[k] = tmp[k]
```

El tiempo de ejecución de *mergeSort* es O(*n*log*n*), en el peor caso y también en el caso promedio

Sabemos que $T(1) = 0$ y $T(n) = 2T(n/2) + n$

Luego,

$$T(n)/n = T(n/2)/(n/2) + 1$$

$$T(n/2)/(n/2) = T(n/4)/(n/4) + 1$$

$$T(n/4)/(n/4) = T(n/8)/(n/8) + 1$$

$$\ldots \quad \ldots$$

$$T(2)/2 = T(1)/1 + 1$$

Si sumamos ambos lados del signo = y cancelamos los términos que aparecen a ambos lados, obtenemos

$$T(n)/n = T(1)/1 + \log n$$

Podemos reducir significativamente el tiempo de ejecución de `mergeSort`

Usar `insertionSort()` para subarreglos pequeños, p.ej., 15 o menos elementos

Probar si el arreglo ya está ordenado, después de las llamadas recursivas a `mergeSort()` y antes de la llamada a `merge()`

Eliminar la copia al arreglo auxiliar `tmp`, intercambiando los roles de los arreglos `a` y `tmp` en cada nivel de la recursión

Podemos implementar `mergeSort()` *botom-up*

quickSort es probablemente el algoritmo de ordenación más usado

Fue inventado por el matemático inglés C.A.R. Hoare en 1960 y publicado en 1962

… y analizado extensamente por R. Sedgewick entre 1975 y 1978

Es popular por varias razones:

- no es difícil de implementar
- funciona bien para varios tipos de datos
- consume menos recursos que cualquier otro algoritmo en muchas situaciones

Ventajas

No usa memoria extra (es decir, ordena *in situ*), excepto por un stack auxiliar pequeño

Toma tiempo proporcional a *n*log*n* en promedio para ordenar *n* datos,

… y los factores constantes de proporcionalidad son pequeños

Desventajas

No es estable

Toma tiempo proporcional a n^2 en el peor caso

Un error simple en su implementación puede causar problemas de desempeño

quickSort es otra aplicación de la estrategia dividir-para-conquistar

Dividir: *Particionar* el arreglo `a[e..w]` en dos subarreglos, `a[e..m-1]` y `a[m+1..w]`, de modo que cada dato en `a[e..m-1]` es menor o igual que `a[m]`, que, a su vez, es menor o igual que cada dato en `a[m+1..w]`

… y calcular el índice `m` durante la partición

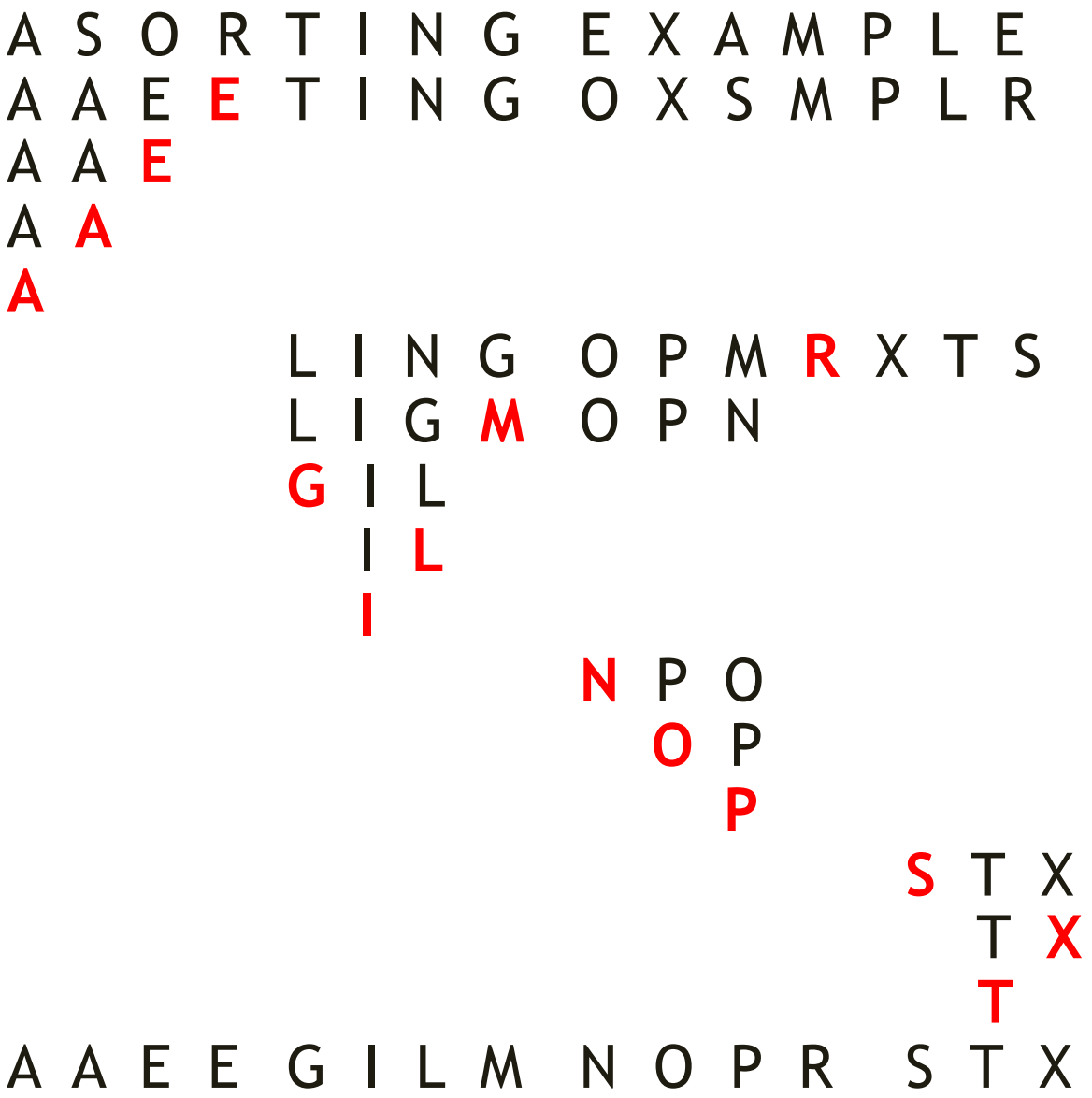
Conquistar: Ordenar los dos subarreglos `a[e..m-1]` y `a[m+1..w]` mediante sendas llamadas recursivas a *quickSort*

Combinar: No es necesario hacer nada; el arreglo `a[e..w]` está ahora ordenado

Ordenamos el arreglo **r** llamando a **quickSort(r)**

```
quickSort(a):
    quickSort(a, 0, a.length-1)

quickSort(a, e, w):
    if e < w:
        m = partition(a, e, w)
        quickSort(a, e, m-1)
        quickSort(a, m+1, w)
```


A S O R T I N G E X A M P L E
A A E E T I N G O X S M P L R
A A E
A A
A
L I N G O P M R X T S
L I G M O P N
G I L
I L
I
N P O
O P
P
S T X
T X
T
A A E E G I L M N O P R S T X

¿Cómo particionamos el arreglo `a[e..w]`? Esta es una manera

Elegimos arbitrariamente `a[w]` como **pivote**

Revisamos a desde `a[e]`, hasta encontrar un `a[j] > a[w]`, y revisamos a desde `a[w-1]`, hasta encontrar un `a[k] < a[w]`

`a[j]` y `a[k]` están *fuera de lugar* en el arreglo particionado final ⇒ los intercambiamos

Continuando de esta manera, nos aseguramos de que ningún dato de a (que esté) a la izquierda de `j` sea mayor que `a[w]`, y de que ningún dato de a (que esté) a la derecha de k sea menor que `a[w]`

Cuando `j` y k se cruzan —es decir, revisamos todo a— intercambiamos `a[w]` con `a[j]`, devolvemos `j` y terminamos la partición

```
int partition(a, e, w):
    j = e-1, k = w
    v = a[w]
    while true:
        j = j+1
        while a[j] < v:
            j = j+1
        k = k-1
        while a[k] > v:
            k = k-1
            if k == e:
                break
        if j >= k:
            break
        exchange(a[j], a[k])
    exchange(a[j], a[w])
    return j
```

Aspectos que hay que considerar

La partición es *in situ*

Al hacer la revisión, de izquierda a derecha y de derecha a izquierda, hay que cuidar no salirse de los límites del (sub)arreglo —por eso,

```
if k == e: break
```

Hay que cuidar terminar el *loop* de la partición, cuando los punteros `j` y k se cruzan —por eso,

```
if j >= k: break
```

Hay que cuidar manejar correctamente los elementos iguales al pivote

Hay que cuidar terminar la recursión —por eso,

```
if e < w: …
```

Desempeño de `quickSort`

La partición siempre deja al menos un dato —el que queda en la posición `j`— en su posición correcta:

se puede demostrar por inducción que `quickSort` efectivamente ordena

El *loop* interno en la partición es muy simple:

- incrementa un índice

  … y compara un elemento del arreglo contra un valor fijo

- en cambio, `shellSort` y `mergeSort` también hacen asignaciones en su loop interno

En último término, el número de operaciones ejecutadas depende de si la partición es balanceada o no,

… lo que a su vez depende de qué datos se usan como pivotes:

- si la partición es balanceada, `quickSort` es asintóticamente tan rápido como `mergeSort`
- si la partición no es balanceada, `quickSort` puede ser tan lento como `insertionSort`

`quickSort` es muy ineficiente en ciertos casos

Un arreglo que ya está ordenado producirá sólo particiones de tamaños n-1 y 0,

… y en total realizará unas $n^2/2$ comparaciones

En estos casos, además, el espacio necesario para manejar la recursión será proporcional a n

Pero hay formas de reducir drásticamente la probabilidad de que este peor caso ocurra

El mejor caso es cuando cada partición divide al arreglo exactamente en mitades: el número de comparaciones satisface la recurrencia

$$C(n) = 2C(n/2) + n$$

… con solución $C(n) = n \log n$

¿Y si todas las particiones producen una división de proporcionalidad 9 a 1?

La recurrencia para el número de comparaciones es

$$C(n) = C(9n/10) + C(n/10) + kn$$

… cuya solución también es

$$C(n) = \mathrm{O}(n \log n)$$

Cualquier división de proporcionalidad constante (p.ej., 99 a 1) produce un número de comparaciones $\mathrm{O}(n \log n)$

¿Cómo analizamos intuitivamente el caso promedio?

El comportamiento de *quickSort* depende de la ordenación relativa de los datos en el arreglo

¿Qué tan frecuentemente ocurren los distintos tipos de secuencias de entrada?

Supondremos que todas las permutaciones de los datos son igualmente probables

Algunas particiones serán razonablemente bien balanceadas, y otras, bastante desbalanceadas

En el caso promedio, *partition* produce una mezcla de particiones buenas y malas,

… distribuidas aleatoriamente a lo largo de la ejecución de *quickSort*

Haremos dos suposiciones acerca de las particiones …

1) Las divisiones buenas y malas se dan alternadamente a lo largo de las llamadas recursivas

2) Las divisiones buenas son del mejor tipo posible, y las malas, del peor

Intuitivamente, la combinación de una división mala seguida por una buena produce tres subarreglos:

- sus tamaños son 0, $(n–2)/2$ y $(n–2)/2$
- el costo de partición total es $n + n – 1 = O(n)$

Este resultado no es peor que el de una única división que produce dos subarreglos:

- ambos de tamaño $(n–1)/2$ a un costo de n
- **… y esta situación es balanceada**

Para un análisis riguroso, veamos las próximas diapositivas

Sea $C(n)$ el número total de comparaciones que hace *quickSort* para un arreglo de n datos

Primero *partition* realiza $n + 1$ comparaciones (incluyendo el cruce de los punteros), y luego se producen dos llamadas recursivas:

- supongamos que al terminar *partition*, el pivote quedó en la *k-ésima* posición del arreglo
- una llamada recursiva es sobre los k-1 datos a la izquierda del pivote, y la otra es sobre los n-k datos a la derecha del pivote

Entonces

$$C(n) = n + 1 + R(n)$$

… en que $R(n)$ es el número de comparaciones debido a las dos llamadas recursivas

Como el pivote puede quedar en cualquiera de las *n* posiciones,

… *R*(*n*) debe calcularse como el promedio de todas las posibilidades:

$$R(n) = \frac{1}{n} \sum_{1 \le k \le n} (C(k-1) + C(n-k))$$

Así,

$$C(n) = n + 1 + \frac{1}{n} \sum_{1 \le k \le n} (C(k-1) + C(n-k))$$

con $C(1) = C(0) = 0$

¿Cómo resolvemos esta ecuación de recurrencia?

Notamos que $C(0) + \cdots + C(n-1) = C(n-1) + \cdots + C(0)$:

$$C(n) = n + 1 + \frac{2}{n} \sum_{1 \le k \le n} C(k-1)$$

Luego, multiplicamos ambos lados por n

y restamos la misma fórmula para $n-1$:

$$nC(n) = (n+1)C(n-1) + 2n$$

Finalmente, dividimos ambos lados por $n(n-1)$:

$$\frac{C(n)}{n-1} = \frac{C(n-1)}{n} + \frac{2}{n+1}$$

$$= \frac{C(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \cdots$$

$$= \frac{C(2)}{3} + \sum_{3 \le k \le n} \frac{2}{k+1}$$

$$\frac{C(n)}{n+1} \approx 2 \sum_{1 \le k < n} \frac{1}{k} \approx 2 \int_{1}^{n} \frac{1}{x} dx = 2 \ln n$$

Versión aleatorizada de quickSort: Desempeño independiente de cómo vienen los datos

```
randomPartition(a, e, w)
    j = random(e, w)
    exchange(a[j], a[w])
    return partition(a, e, w)

randomQuickSort(a, e, w)
    if e < w:
        m = randomPartition(a, e, w)
        randomQuickSort(a, e, m-1)
        randomQuickSort(a, m+1, w)
```

Posibles mejoras

Cambiar a `insertionSort()` para subarreglos pequeños ($n \leq 20$)

Usar la mediana de tres elementos como pivote

Si hay cantidades grandes de claves duplicadas en el arreglo de entrada —p.ej., un archivo de fechas— es muy probable que `quickSort()` particione, innecesariamente, subarreglos en que todas las claves son iguales:

- podemos particionar el arreglo en tres partes: ítemes con claves menores que el pivote, iguales al pivote, y mayores que el pivote

El problema de selección

Input: Un conjunto a de n números distintos, y un número j, $1 \leq j \leq n$

Output: El elemento x de a que es más grande que exactamente $j - 1$ otros elementos de a —la ***j*-ésima estadística de orden**

Por supuesto, este problema puede resolverse en tiempo $O(n \log n)$

¿Puede resolverse en menos tiempo?

Estadísticas de orden extremas

¿Cuántas comparaciones tenemos que hacer para determinar el mínimo de un conjunto de *n* elementos?

¿Y para determinar el máximo?

¿Y para determinar el mínimo y el máximo?

Selección en tiempo O(*n*) en promedio

```
int randomSelect(a, p, r, j):
    if p == r:
        return a[p]
    q = randomPartition(a, p, r)
    k = q-p+1
    if j == k:
        return a[q]
    else
        if j < k:
            return randomSelect(a, p, q-1, j)
        else:
            return randomSelect(a, q+1, r, j-k)
```

¿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*

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

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 $\Theta(n)$

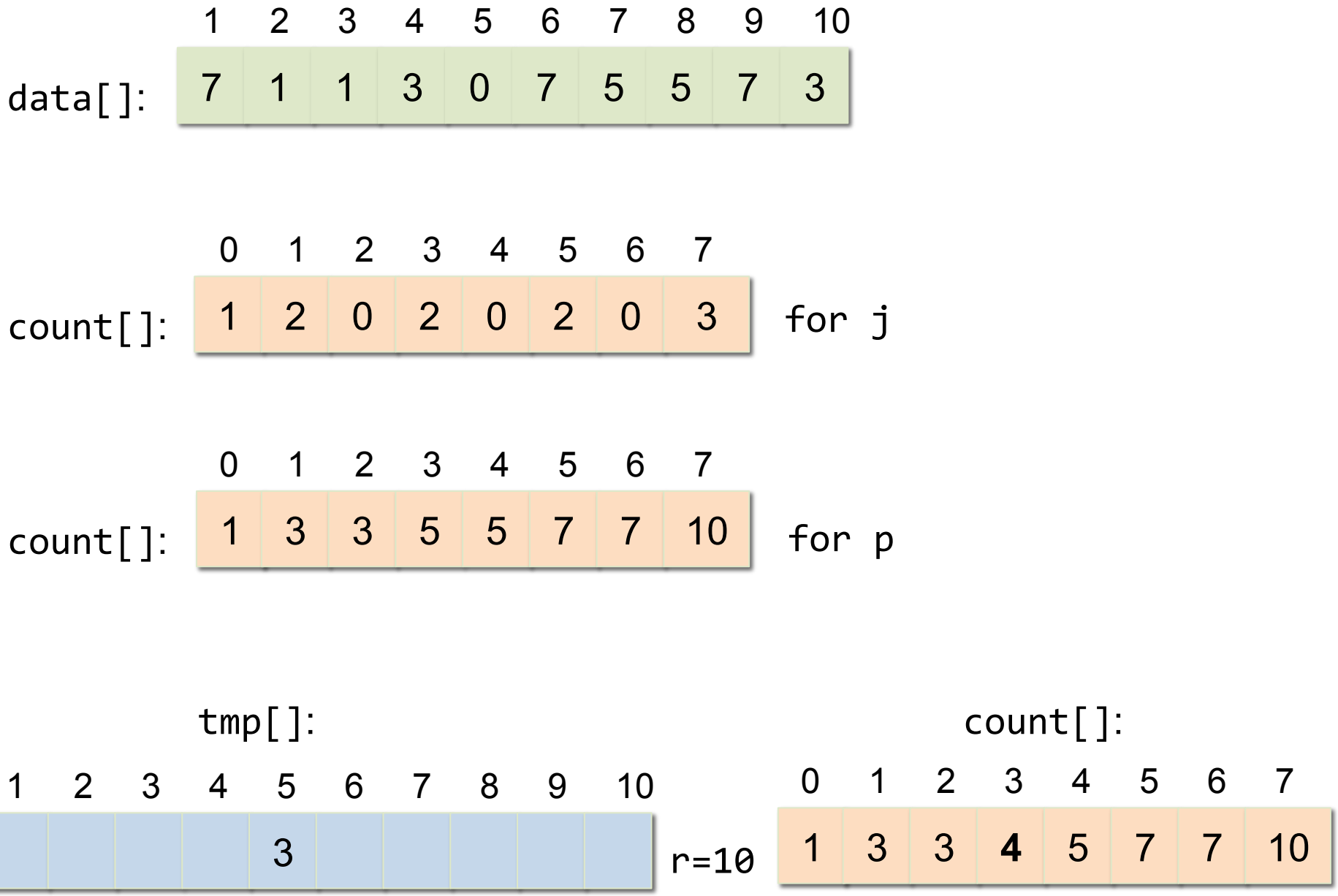
Determinamos, para cada dato *x*, el número de datos menores que *x*:

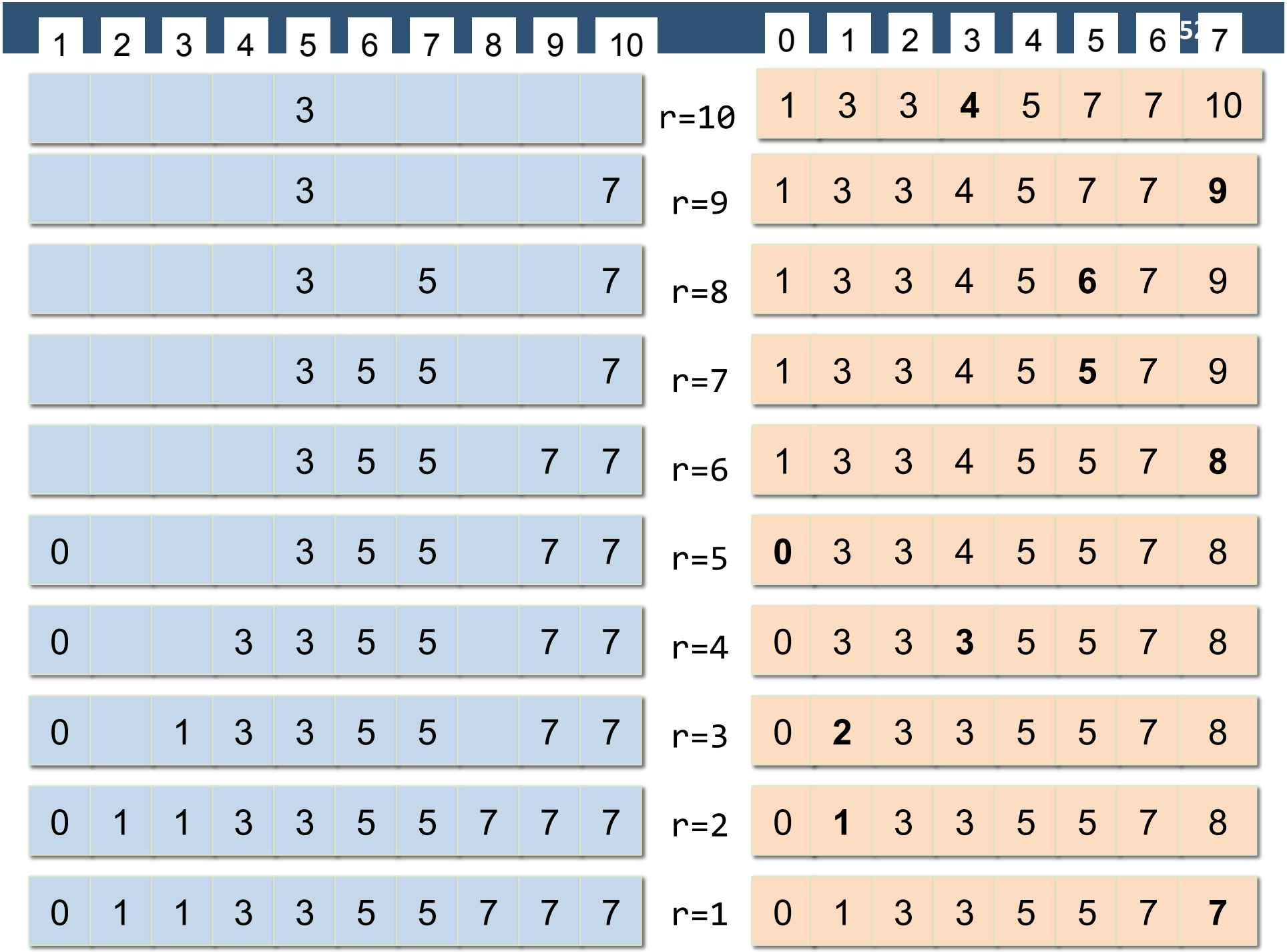
- esto permite ubicar a *x* directamente en su posición final en el arreglo de salida
- hay que manejar el caso en que varios datos tengan el mismo valor

```
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
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, 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

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 $\boldsymbol{\Theta(n)}$

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*

she	**a**re	are	are	…	are
seels	**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	se**l**ls		sells
shore	**s**hore	s**e**ashells	se**l**ls		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