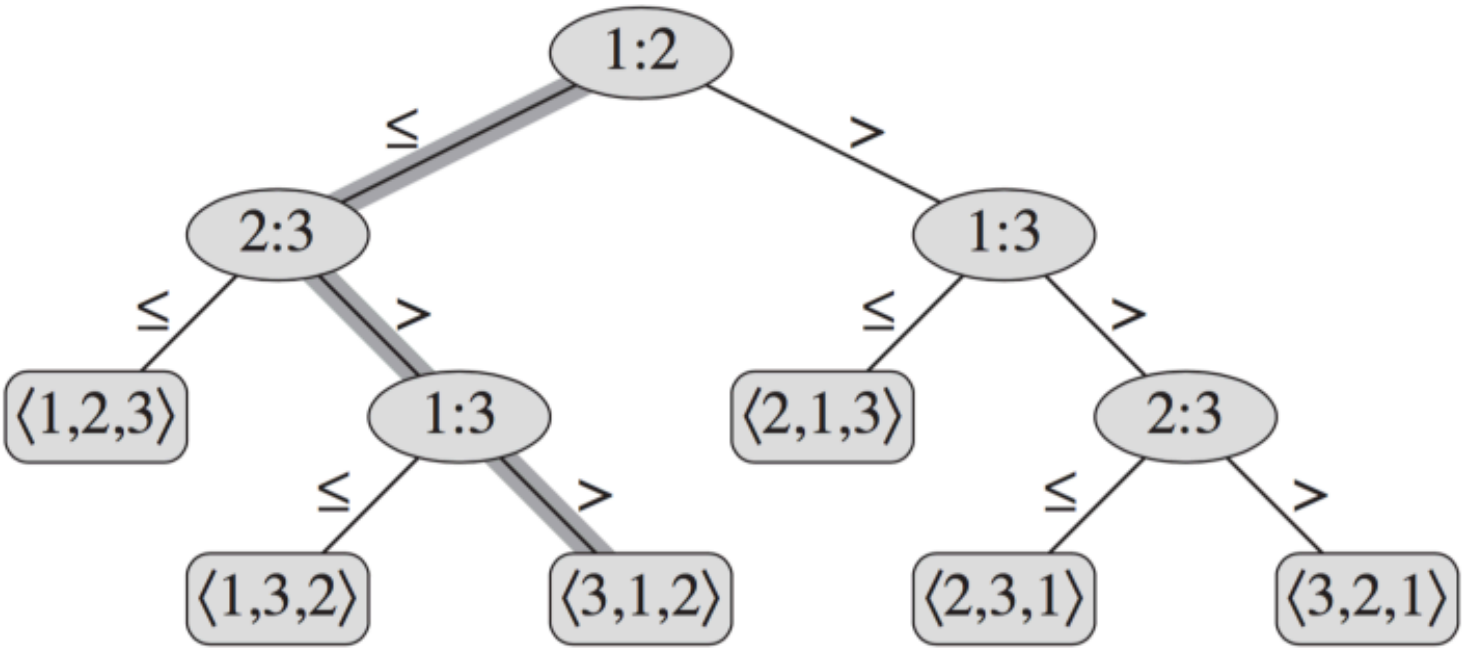
Notas de algoritmos de ordenamiento, última parte

Cota inferior para ordenamientos por comparación

Para ordenar tres valores, ¿cuántas comparaciones tenemos que hacer? Contando el caso "else" como una comparación, veamos el siguiente árbol de comparaciones posibles, a fin de tener ordenados tres valores.



Siguiendo el camino señalado: `1:2` significa compara `A[1]` con `A[2]`, si es *menor o igual*, entonces revisamos la comparación `2:3`, si resulta que `A[2]` es mayor que `A[3]`, entonces comparamos `1:3`, si resulta que `A[1] > A[3]`, entonces la permutación que dá como resultado la lista ordenada de valores en el arreglo será `<3, 1, 2>`. Esto es, el arreglo ordenado será

```
A[3], A[1], A[2]
```

Como se observa, un algoritmo de ordenamiento por comparación debe seguir por lo menos, un camino desde la raíz hasta una hoja. Las hojas contienen las permutaciones posibles para los índices del arreglo de tres elementos a ordenar.

En general, sabemos que el número de permutaciones de `n` elementos es `n!`. También sabemos que un árbol binario de `h` niveles contiene no más de `2^h` hojas. Entonces un árbol de decisión como el anterior con `L` hojas alcanzables, cumplirá:

`n! <= L <= 2^h`

Para saber el mínimo número de pasos que un algoritmo de comparacion debe seguir, basta saber la altura de su árbol de comparaciones.

Tomando logaritmos, esto implica que:

```
h >= lg(n!)   ; de la 2a desigualdad,
h = Omega(n lg n)   ; de la fórmula de Stirling
```

Dado que *Heapsort*, *Merge* y otros tardan `O(n lg n)`, estos se les llama *algoritmos óptimos de comparación*.

> A continuación veremos un par de algoritmos que no utilizan la metodología de comparar los elementos a ordenar.

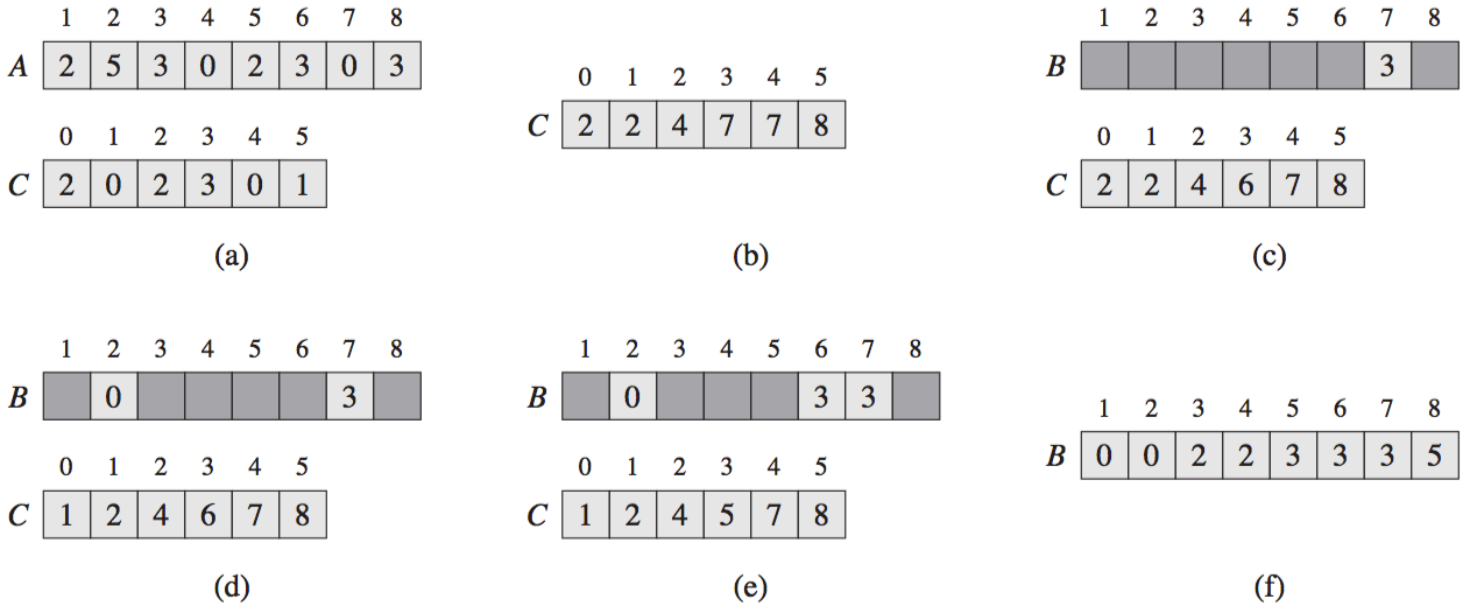
Ordenamiento por conteo

Por supuesto, los datos no pueden ser tan generales como con los anteriores algoritmos (por cierto, hemos usado estos algoritmos para ordenar enteros, pero los elementos del arreglo pueden ser cualquier cosa comparable).

Para el caso de ordenamiento por conteo asumimos que los `n` elementos a comparar son enteros en el rango de `0` a `k`.

Esencialmente el algoritmo determina la posición de un elemento al contar cuantos elementos a ordenar son mas pequeños que él.

```
Ordenamiento-por-conteo(A, B, K):
 1 Sea C[0...k] un arreglo auxiliar
 2 Para i = 0 hasta k
 3     C[i] = 0
 4 Para j = 0 hasta len(A) - 1
 5     C[A[j]] = C[A[j]] + 1
 6     ; Ahora C[i] contiene el número de elementos iguales a i
 7 Para i = 1 hasta k
 8     C[i] = C[i] + C[i-1]
 9     ; Ahora C[i] contiene el número de elementos menores o iguales a i
10 Para j = len(A) - 1 hasta 0
11     B[C[A[j]]] = A[j]
12     C[A[j]] = C[A[j]] - 1
```

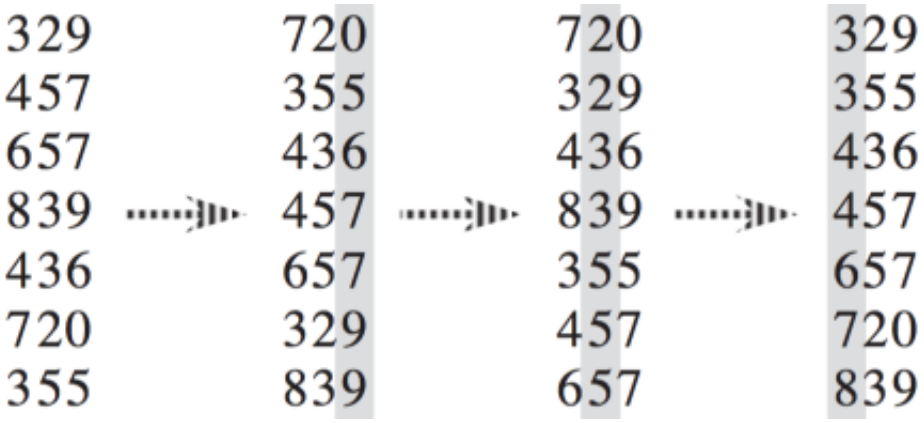


Este algoritmo en el ciclo de las líneas 2-3 toma tiempo `Theta(k)`, el ciclo de las líneas 4-5 toma tiempo `Theta(n)`. El ciclo de las líneas 7-8 toma tiempo `Theta(k)`, y por último el ciclo de las líneas 10-12 toma tiempo `Theta(n)`. Así el algoritmo de conteo ocupa un tiempo total de `Theta(n + k)`. Cuando `k=O(n)`, este algoritmo corre en tiempo `Theta(n)`.

Radix sort

Sirve para ordenar conjuntos de valores numéricos con la misma cantidad de dígitos. Es análogo a como se ordenan fechas, por ejemplo, ordenando primero los días, y si hay empates se ordenan los meses, y si hay empates se ordenan los años. O como se ordenaría un conjunto de datos con una hoja de cálculo: ordena primero por la columna A, luego por la columna B, luego por la C, etc...

```
Ordenamiento-Radix(A, d):
1 Para i = 1 hasta d
2     Use un algoritmo estable para ordenar el arreglo A con el dígito i
```



Dado que en cada columna tenemos que ordenar respecto a un dígito, podemor usar ordenamiento por conteo para ordenar el arreglo. Así, si el arreglo tiene `n` elementos a ordenar, para cada dígito tardaremos `Theta(n + k)`. Así para los `d` dígitos tenemos que el ordenamiento Radix se tarda `Theta(d(n + k))`. Si `d` es constante y `k=O(n)`, tenemos que esta algoritmo se tarda también `Theta(n)`.

- ¿Cómo se comparan los tiempos de estos algoritmos con los algoritmos basados en comparaciones?
- ¿Cuánto espacio ocupan?
- ¿Cuáles serían las implementaciones más eficientes para estos algoritmos?