

Repaso 11

SORTING - ARBOLES - HEAPS - ORDEN
LINEAL



Ordenamiento - Examen 2024-2 P1

Dada la siguiente versión del algoritmo **Selection Sort**:

```
input: Secuencia de datos en Arreglo A[]
output: Nueva secuencia B[] ordenada
selectionSort(A[]):
    B ← vacia
    for i = 0 to n - 2:
        min ← 0
        for j = 1 to n - 1:
            if A[j] < A[min]:
                min ← j
        B[i] ← A[min]
        A[min] ← +infinito
    return B
```

Ordenamiento - Examen 2024-2 P1

- a) Escriba el pseudocódigo para Tournament Sort, el cual es una mejora de Selection Sort, al utilizar una cola priorizada para seleccionar al siguiente elemento en orden.
- b) Determine la complejidad de tiempo de Tournament Sort: ¿Hay un mejor o peor caso? ¿Cuál es su complejidad de memoria?
- c) Argumente que Tournament Sort es correcto
- d) ¿Es Tournament Sort estable? Argumente su respuesta

Ordenamiento - Solución

- a) Escriba el pseudocódigo para Tournament Sort, el cual es una mejora de Selection Sort, al utilizar una cola priorizada para seleccionar al siguiente elemento en orden.

Hay varias opciones de solución, puede ser construyendo un heap de forma incremental o como se muestra a continuación:

```
input: Secuencia de datos en Arreglo A[]
output: Nueva secuencia B[] ordenada
selectionSort(A[]):
    B ← vacia
    A ← BuildHeap(A) // minHeap
    for i = 0 to n - 1:
        min ← extract(A)
        B[i] ← min
    return B
```

Ordenamiento - Solución

b) Determine la complejidad de tiempo de Tournament Sort:
¿Hay un mejor o peor caso? ¿Cuál es su complejidad de memoria?

BuidHeap es $O(n)$. El for ejecuta n veces `extract()` es $O(n \log n)$,
Luego Tournament Sort es $O(n \log n)$ en tiempo.

Al igual que selection sort, TS no aprovecha la composicion inicial del input al seleccionar el menor valor, y tampoco las funciones del heap, luego no hay un mejor o peor caso.

La complejidad de memoria es $O(n)$ ya que requiere la memoria adicional del arreglo B.

```
input: Secuencia de datos en Arreglo A[]
output: Nueva secuencia B[] ordenada
selectionSort(A[]):
    B ← vacia
    A ← BuidHeap(A) // minHeap
    for i = 0 to n - 1:
        min ← extract(A)
        B[i] ← min
    return B
```

Ordenamiento - Solución

c) Argumente que Tournament Sort es correcto.

Para esto debemos probar dos cosas: que **termina** y que cumple lo pedido, es decir, **ordena correctamente**.

El arreglo A es finito, BuildHeap() es correcto (termina), el for es sobre un numero finito de elementos y extract() es correcto (termina) luego, **TS termina**.

De igual forma, sabemos que extract() extrae del minHeap el menor elemento en cada paso, luego en un paso n, los n elementos extraidos son menores que los que permanecen en el heap, dado que sabemos que extract es correcto. Asi en un paso n+1, el valor extraido es mayor igual que los anteriores, por lo que **TS ordena correctamente**.

Ordenamiento - Solución

d) ¿Es Tournament Sort estable? Argumente su respuesta

No. Las operaciones del heap no garantizan mantener el orden de llegada de los elementos del arreglo al realizar el BuildHeap().

Árboles AVL

Usted es un aficionado a la meteorología y descubrió que una estación meteorológica personal con medición de temperatura, humedad, velocidad y dirección del viento y agua caída vale US\$300. Además, existe una red de estaciones meteorológicas mundial conectadas a internet que entregan el acceso a sus datos en tiempo real. Cada estación tiene un identificador compuesto por el sector donde se ubica (puede asociarlo a latitud, longitud o código postal) y nunca se repiten.

Usted está desarrollando un software de predicción del clima y para eso tiene un server recolectando y almacenando en una estructura de datos que contiene el id de la estación, el momento (timestamp) de la medición y los datos meteorológicos. Como su sistema es intensivo en consultas Ud. Ha decidido usar un árbol AVL para las estaciones y en que cada nodo se tiene un arreglo que almacena todos los datos del último mes en orden de llegada (timestamp).

Como los datos pueden ser millones y usan mucho espacio en memoria, usted solo mantendrá en la estructura los datos de las estaciones en línea (online) y si llega a detectar que está offline el sistema saca todos esos datos y los almacena en disco. Cuando la estación comienza a transmitir nuevamente (o sea se recibe un nuevo conjunto de datos) se vuelven a cargar los datos antiguos almacenados en disco, junto al nuevo, en la estructura y la estación se marca online.

Para su modelo meteorológico Ud. Debe ingresar el id de una estación (cualquiera de la red mundial) y el rango de tiempo de una estación y consultar todos los datos de las estaciones cercanas en el mismo rango

Árboles AVL – i1 2024-2 P4

Existe una **red de estaciones conectadas que entregan datos meteorológicos** en tiempo real.

Cada estación tiene un **id único**

El **server** recolecta y **almacena** en una edd **los datos de id de la estación, timestamp medición y los datos como tal**

Sistema es intensivo en consultas: usará **AVL para modelar las estaciones** y cada nodo tiene un arreglo que almacena todos los datos del último mes en orden de llegada (timestamp).

Millones de datos y usan mucho espacio en memoria. Entonces **solo mantendrá en la EDD las estaciones online**

Si estación está **offline**, el sistema almacena los **datos en disco**.

Cuando la estación **comienza a transmitir nuevamente**, se vuelven a cargar los datos antiguos almacenados en disco, junto al nuevo, en la estructura y la estación **se marca online**.

El modelo meteorológico **debe permitir** ingresar el id de una estación (cualquiera de la red mundial), el rango de tiempo de una estación y consultar todos los datos de las estaciones cercanas en el mismo rango

Árboles AVL – i1 2024-2 P4

- a) Haga un esquema de la estructura con nodos online, offline y conjuntos de datos de una estación nueva, llena de datos meteorológicos y con algunos datos

Árboles AVL – i1 2024-2 P4- Solución

- a) Haga un esquema de la estructura con nodos online, offline y conjuntos de datos de una estación nueva, llena de datos meteorológicos y con algunos datos

Nodo AVL:

- id: nombre de la estación
- flag: estado online/offline
- array: arreglo para almacenar (timestamp, dato)

Ejemplo de un nodo:

id: "Estación Norte"

flag: online

array: [(t1,d1), (t2,d2)]

Árboles AVL

b) Escriba en Pseudocódigo las funciones e **indique su complejidad**:

- **Insert(id, tiempo, dato)**: Inserta dato en estación, si existe lo agrega a los datos, si no existe la crea, si estaba offline la trae de vuelta a la edd.
- **Datos(id, inicio, término)**: Obtiene todos los datos de la estación id en un rango de tiempo, si id no existe entrega null
- **Offline(id)**: marca la estación offline, escribe en disco todos los datos meteorológicos y lo borra de la edd. El nodo NO SE BORRA
- **Todo(id, inicio, término)**: Retorna todos los datos en el rango de tiempo de la estación id y todas las estaciones cercanas

Funciones usables que ya existen:

c= Cercanas(id), retorna un arreglo con la lista de las estaciones cercanas geográficamente

void page-out(id), no retorna nada y guarda en disco el arreglo con los datos meteorológicos

A= page-in(id), entrega en el arreglo todos los datos almacenados en disco de la estación id

Árboles AVL - Insert

Insert(Ei, ti, di):

$p \leftarrow \text{InsertAVL}(E_i)$ // Aquí se busca la estación E_i en el árbol AVL, si no existe se crea e inserta

$p.\text{flag} \leftarrow \text{online}$ // Inicialmente se marca online

$\text{InsertDatos}(p.\text{array}, t_i, d_i)$ // Se llama a una función para insertar los datos

Árboles AVL - Insert

InsertDatos(a, t, d):

si hay espacio en array a:

$i \leftarrow$ primer lugar vacío // del arreglo a del nodo E_i

$a[i].t \leftarrow t$; $a[i].d \leftarrow d$

si no (el arreglo está lleno):

for $i = n-1$ to 0 $a[i+1] \leftarrow a[i]$; // Se corren los valores en una posición y se sobrescribe el más antiguo

$a[0].t \leftarrow t$; $a[0].d \leftarrow d$

Árboles AVL - Insert

InsertAVL(nodo, id):

if nodo = null, arbol vacio

return nuevonodo(id)

else if id < nodo.id

nodo.left = InsertAVL(nodo.left, id) // inserta izquierdo

else if id > nodo.id

nodo.right = InsertAVL(nodo.right, id) inserta nodo derecho

Balance

return node

Árboles AVL – Insert + Complejidad

Insert(E_i , t_i , d_i):

$p \leftarrow \text{InsertAVL}(E_i)$ // Aquí se busca la estación E_i en el árbol AVL, si no existe se crea e inserta

$p.\text{flag} \leftarrow \text{online}$ // Inicialmente se marca online

$\text{InsertDatos}(p.\text{array}, t_i, d_i)$ // Se llama a una función para insertar los datos

$$O(\log(n) + n)$$

Árboles AVL – Datos + Complejidad

Datos(id, inicio, término):

$p \leftarrow \text{BuscarNodo}(id)$

si existe:

$i \leftarrow \text{búsqueda binaria en arreglo } p$

mientras $p.a[i].\text{timestamp} \leq \text{término}$:

copiar dato a arreglo de respuesta

return arreglo de datos

$$O(\log(E) + \log(n)) + m$$

Árboles AVL – Offline + Complejidad

Offline(id):

if p = BuscarNodo(id):

 page-out(id)

 p.a ← null

 p.flag ← offline

$$O(\log(E) + 1)$$

Árboles AVL – Todo + Complejidad

Todo(id, inicio, término):

t ← matriz vacía

for i in cercanas(id):

t ← t + Datos(id, inicio, término)

$O(c \log(E))$

Heaps

Como parte de la búsqueda de una Inteligencia Artificial General (AGI) un grupo de aficionados propone utilizar los miles (n) de modelos especializados de AI existentes (narrow IA) en paralelo, enviándoles la misma “pregunta” al mismo tiempo, y elegir como “respuesta” aquella con mayor confiabilidad (la respuesta de cada modelo viene acompañada de un valor entre 0 y 1 que indica la confiabilidad de la respuesta, siendo 1 el 100% de confianza). Actualmente han logrado construir el software necesario para enviar la pregunta a los modelos especializados y almacenar las respuestas en un arreglo $A[0 \dots n - 1]$, en orden de llegada para los primeros 3 segundos.

a) Proponga un algoritmo en pseudocódigo para la función `Answer(A)` que permite obtener de la manera más eficiente la respuesta de mayor confianza desde el arreglo `A` sin eliminarla.

Solución

- Suponemos que los datos se almacenan con atributos respuesta y confiabilidad. Además, luego de recibir los datos iniciales, se aplica BuildHeap(A) para transformar al arreglo A en un heap binario, donde se usa el atributo confiabilidad como prioridad. Con estos supuestos,

input : Arreglo *A* que representa un heap binario

Answer(*A*):

1 **return** *A*[0].*respuesta*

b) Un caso de uso del sistema es que el usuario pueda pedir la "siguiente mejor respuesta" al sistema, eliminándola del arreglo. Proponga el pseudo código para la función nextAnswer(A).

Solución

- Dado el enunciado, puede interpretarse como entregar el elemento más prioritario o entregar el segundo más prioritario. Ambos se consideran correctos para efectos de la corrección. La implementación de ambos

```
input : Arreglo  $A$  que representa un heap binario
nextAnswer( $A$ ):
1   return Extract( $A$ ).respuesta

nextAnswer( $A$ ):
2   if  $A[1].confiabilidad \geq A[2].confiabilidad$  :
3       return Extract( $A$ , 1).respuesta
4   return Extract( $A$ , 2).respuesta
```

En el segundo caso, se asume que Extract(A , i) opera de la misma forma que Extract visto en clase, pero extrayendo la posición indicada y restableciendo la propiedad de heap.

input : Arreglo A que representa un heap binario

nextAnswer(A):

1 **return** $\text{Extract}(A).respuesta$

nextAnswer(A):

2 **if** $A[1].confiabilidad \geq A[2].confiabilidad$:

3 **return** $\text{Extract}(A, 1).respuesta$

4 **return** $\text{Extract}(A, 2).respuesta$

Extract(H):

$i \leftarrow$ última celda no vacía de H

$best \leftarrow H[0]$

$H[0] \leftarrow H[i]$

$H[i] \leftarrow \emptyset$

SiftDown($H, 0$)

return $best$

SiftDown(H, i):

if i tiene hijos :

$j \leftarrow$ hijo de i con mayor prioridad

if $H[j] > H[i]$:

$H[j] \rightleftharpoons H[i]$

SiftDown(H, j)

c) Dado que algunos modelos especializados son más lentos en responder se decide permitir que el arreglo A pueda recibir respuestas "tardías" (después de los 3 segundos iniciales) para ser consideradas en los algoritmos anteriores (una vez que dichas respuestas están disponibles). Proponga el pseudo código para la función `lateAnswer(A, respuesta, confiabilidad)` que permite incorporar de manera eficiente una respuesta nueva al arreglo A una vez que este ya fue "consultado" por los algoritmos anteriores.

Solución

input : Arreglo A que representa un heap binario, respuesta y confiabilidad

lateAnswer(A , $respuesta$, $confiabilidad$):

- 1 $i \leftarrow$ primera celda vacía de A
- 2 $A[i].respuesta \leftarrow respuesta$
- 3 $A[i].confiabilidad \leftarrow confiabilidad$
- 4 **SiftUp**(A, i)

SiftUp(H, i):

if i tiene padre :

$j \leftarrow \lfloor i/2 \rfloor$

if $H[j] < H[i]$:

$H[j] \rightleftharpoons H[i]$

SiftUp(H, j)

Orden Lineal (2015-1-12-P3-B)

b) Otra forma de ordenar strings, especialmente cuando son de diferentes largos, es considerar los caracteres de izquierda a derecha y usar el siguiente método recursivo: Usamos `countingSort()` para ordenar los strings de acuerdo con el primer carácter; luego, recursivamente, ordenamos los strings que tienen un mismo primer carácter (excluyendo este primer carácter). Similarmente a `quickSort()`, este algoritmo particiona el arreglo de strings en subarreglos que pueden ser ordenados independientemente para completar la tarea, sólo que particiona el arreglo en un subarreglo para cada posible valor del primer carácter, en lugar de las dos particiones de `quickSort()`. Escribe este algoritmo.

Orden Lineal (2015-1-12-P3-B)

b) Otra forma de ordenar strings, especialmente cuando son de diferentes largos, es considerar los caracteres de izquierda a derecha y usar el siguiente método recursivo: Usamos countingSort() para ordenar los strings de acuerdo con el primer carácter; luego, recursivamente, ordenamos los strings que tienen un mismo primer carácter (excluyendo este primer carácter). Similarmente a quickSort(), este algoritmo particiona el arreglo de strings en subarreglos que pueden ser ordenados independientemente para completar la tarea, sólo que particiona el arreglo en un subarreglo para cada posible valor del primer carácter, en lugar de las dos particiones de quickSort(). Escribe este algoritmo.

MSD 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

`sortString(a, e, w, k):`

—ordena recursivamente el arreglo **a** de strings, desde **a[e]** hasta **a[w]**, a partir del **k**-ésimo carácter

—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**

—usa arreglos auxiliares **b** y **c**

if **e** < **w**:

for **i** = **e**, ..., **w**:

p = **dgt**(**a**[**i**][**k**])

b[**p**+2] = **b**[**p**+2]+1

for **r** = 0, ..., 128:

b[**r**+1] = **b**[**r**+1] + **b**[**r**]

for **i** = **e**, ..., **w**:

p = **dgt**(**a**[**i**][**k**])

c[**b**[**p**+1]] = **a**[**i**]

b[**p**+1] = **b**[**p**+1]+1

for **i** = **e**, ..., **w**:

a[**i**] = **c**[**i**-**e**]

for **r** = 0, ..., 127:

sortString(**a**, **e**+**b**[**r**], **e**+**b**[**r**+1]-1, **k**+1)

—los cuatro primeros **for** implementan **countingSort()**

—el quinto **for** hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo **k**-ésimo carácter)

`sortString(a, e, w, k):`

—ordena recursivamente el arreglo **a** de strings, desde **a[e]** hasta **a[w]**, a partir del **k**-ésimo carácter

—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**

—usa arreglos auxiliares **b** y **c**

if `e < w`:

for `i = e, ..., w`:

`p = dgt(a[i][k])`

`b[p+2] = b[p+2]+1`

for `r = 0, ..., 128`:

`b[r+1] = b[r+1] + b[r]`

for `i = e, ..., w`:

`p = dgt(a[i][k])`

`c[b[p+1]] = a[i]`

`b[p+1] = b[p+1]+1`

for `i = e, ..., w`:

`a[i] = c[i-e]`

for `r = 0, ..., 127`:

`sortString(a, e+b[r], e+b[r+1]-1, k+1)`

—los cuatro primeros **for** implementan **countingSort()**

—el quinto **for** hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo **k**-ésimo carácter)

Ejemplo:

`sortString(a,0,2,0)`

`a = ["ba", "ab", "aa"]`

`dgt('a') = 0, dgt('b') = 1`

`b = [0, 0, 0, 0]`

`c = [−, −, −]`

sortString(a, e, w, k):
—ordena recursivamente el arreglo **a** de strings, desde **a[e]** hasta **a[w]**, a partir del **k-ésimo** carácter
—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**
—usa arreglos auxiliares **b** y **c**
 if e < w:
 for i = e, ..., w:
 p = dgt(a[i][k])
 b[p+2] = b[p+2]+1
 for r = 0, ..., 128:
 b[r+1] = b[r+1] + b[r]
 for i = e, ..., w:
 p = dgt(a[i][k])
 c[b[p+1]] = a[i]
 b[p+1] = b[p+1]+1
 for i = e, ..., w:
 a[i] = c[i-e]
 for r = 0, ..., 127:
 sortString(a, e+b[r], e+b[r+1]-1, k+1)
—los cuatro primeros **for** implementan **countingSort()**
—el quinto **for** hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo **k-ésimo** carácter)

Ejemplo: a = ["ba", "ab", "aa"]

i	a[i]	p=dgt(a[i][0])	b antes	b después
0	"ba"	1	[0 0 0 0]	[0 0 0 1]
1	"ab"	0	[0 0 0 1]	[0 0 1 1]
2	"aa"	0	[0 0 1 1]	[0 0 2 1]

b = [0, 0, 2, 1]

`sortString(a, e, w, k):`

—ordena recursivamente el arreglo **a** de strings, desde **a[e]** hasta **a[w]**, a partir del **k**-ésimo carácter

—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**

—usa arreglos auxiliares **b** y **c**

if `e < w`:

for `i = e, ..., w`:

`p = dgt(a[i][k])`

`b[p+2] = b[p+2]+1`

[for `r = 0, ..., 128:`
`b[r+1] = b[r+1] + b[r]`]

for `i = e, ..., w`:

`p = dgt(a[i][k])`

`c[b[p+1]] = a[i]`

`b[p+1] = b[p+1]+1`

for `i = e, ..., w`:

`a[i] = c[i-e]`

for `r = 0, ..., 127`:

`sortString(a, e+b[r], e+b[r+1]-1, k+1)`

—los cuatro primeros **for** implementan **countingSort()**

—el quinto **for** hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo **k**-ésimo carácter)

Ejemplo: `a = ["ba", "ab", "aa"]`

`b = [0, 0, 2, 1] → [0, 0, 2, 3]`

sortString(a, e, w, k):

—ordena recursivamente el arreglo **a** de strings, desde **a[e]** hasta **a[w]**, a partir del **k**-ésimo carácter

—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**

—usa arreglos auxiliares **b** y **c**

if e < w:

for i = e, ..., w:

p = dgt(a[i][k])

b[p+2] = b[p+2]+1

for r = 0, ..., 128:

b[r+1] = b[r+1] + b[r]

for i = e, ..., w:

p = dgt(a[i][k])

c[b[p+1]] = a[i]

b[p+1] = b[p+1]+1

for i = e, ..., w:

a[i] = c[i-e]

for r = 0, ..., 127:

sortString(a, e+b[r], e+b[r+1]-1, k+1)

—los cuatro primeros **for** implementan **countingSort()**

—el quinto **for** hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo **k**-ésimo carácter)

Ejemplo: a = ["ba", "ab", "aa"]

i	p	b[p+1] antes	c antes	c después	b[p+1] después
0	1	b[2] = 2	[−, −, −]	[−, −, "ba"]	b[2] = 3
1	0	b[1] = 0	[−, −, "ba"]	"ab" , −, "ba"]	b[1] = 1
2	0	b[1] = 1	["ab", −, "ba"]	["ab", "aa" , "ba"]	b[1] = 2

c = ["ab", "aa", "ba"]

b = [0, 2, 3, 3]

sortString(a, e, w, k):

—ordena recursivamente el arreglo **a** de strings, desde **a[e]** hasta **a[w]**, a partir del **k**-ésimo carácter

—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**

—usa arreglos auxiliares **b** y **c**

if e < w:

for i = e, ..., w:

p = dgt(a[i][k])

b[p+2] = b[p+2]+1

for r = 0, ..., 128:

b[r+1] = b[r+1] + b[r]

for i = e, ..., w:

p = dgt(a[i][k])

c[b[p+1]] = a[i]

b[p+1] = b[p+1]+1

(for i = e, ..., w:

a[i] = c[i-e]

for r = 0, ..., 127:

sortString(a, e+b[r], e+b[r+1]-1, k+1)

—los cuatro primeros **for** implementan **countingSort()**

—el quinto **for** hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo **k**-ésimo carácter)

Ejemplo: a = ["ba", "ab", "aa"]

i	c[i]	a antes	a después
0	"ab"	["ba", "ab", "aa"]	["ab", "ab", "aa"]
1	"aa"	["ab", "ab", "aa"]	["ab", "aa", "aa"]
2	"ba"	["ab", "aa", "aa"]	["ab", "aa", "ba"]

c = ["ab", "aa", "ba"]

b = [0, 2, 3, 3]

a = ["ab", "aa", "ba"]

sortString(a, e, w, k):

—ordena recursivamente el arreglo **a** de strings, desde **a[e]** hasta **a[w]**, a partir del **k**-ésimo carácter

—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**

—usa arreglos auxiliares **b** y **c**

(if e < w:)

for i = e, ..., w:

p = dgt(a[i][k])

b[p+2] = b[p+2]+1

for r = 0, ..., 128:

b[r+1] = b[r+1] + b[r]

for i = e, ..., w:

p = dgt(a[i][k])

c[b[p+1]] = a[i]

b[p+1] = b[p+1]+1

for i = e, ..., w:

a[i] = c[i-e]

(for r = 0, ..., 127:)

sortString(a, e+b[r], e+b[r+1]-1, k+1)

—los cuatro primeros **for** implementan **countingSort()**

—el quinto **for** hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo **k**-ésimo carácter)

r	b[r]	b[r+1]	e+b[r]	e+b[r+1]-1	Llamado
0	0	2	0	1	sortString(a, 0, 1, k+1) ["ab","aa"]
1	2	3	2	2	sortString(a, 2, 2, k+1) ["ba"]
2	3	3	3	2 (vacío)	(e>w, no recursión)
...

c = ["ab", "aa", "ba"]

b = [0, 2, 3, 3]

a = ["ab", "aa", "ba"]

`sortString(a, e, w, k):`

—ordena recursivamente el arreglo **a** de strings, desde **a[e]** hasta **a[w]**, a partir del **k**-ésimo carácter

—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**

—usa arreglos auxiliares **b** y **c**

if **e** < **w**:

for **i** = **e**, ..., **w**:

p = **dgt**(**a**[**i**][**k**])

b[**p**+2] = **b**[**p**+2]+1

for **r** = 0, ..., 128:

b[**r**+1] = **b**[**r**+1] + **b**[**r**]

for **i** = **e**, ..., **w**:

p = **dgt**(**a**[**i**][**k**])

c[**b**[**p**+1]] = **a**[**i**]

b[**p**+1] = **b**[**p**+1]+1

for **i** = **e**, ..., **w**:

a[**i**] = **c**[**i**-**e**]

[**for** **r** = 0, ..., 127:
 sortString(**a**, **e**+**b**[**r**], **e**+**b**[**r**+1]-1, **k**+1) **]**

—los cuatro primeros **for** implementan **countingSort()**

—el quinto **for** hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo **k**-ésimo carácter)

Con el llamado `sortString(a, 0, 1, 1)` terminamos obteniendo el ordenamiento esperado:

`a = ["aa", "ab", "ba"]`

`sortString(a, e, w, k):`

—ordena recursivamente el arreglo **a** de strings, desde **a[e]** hasta **a[w]**, a partir del **k**-ésimo carácter


—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**

—usa arreglos auxiliares **b** y **c**

if `e < w`:

for `i = e, ..., w`:

`p = dgt(a[i][k])`

`b[p+2] = b[p+2]+1`  +2 para no quedar out of range mas adelante


for `r = 0, ..., 128`:

`b[r+1] = b[r+1] + b[r]`

for `i = e, ..., w`:

`p = dgt(a[i][k])`

`c[b[p+1]] = a[i]`

`b[p+1] = b[p+1]+1` 

En el algoritmo de countingSort visto en clases se le restaba uno por leer de derecha a izquierda, en este caso es al revés, por eso se suma 1

for `i = e, ..., w`:

`a[i] = c[i-e]`

for `r = 0, ..., 127`:

`sortString(a, e+b[r], e+b[r+1]-1, k+1)`

—los cuatro primeros **for** implementan **countingSort()**

—el quinto **for** hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo **k**-ésimo carácter)