Ordenación lineal

Clase 6

IIC 2133 coordinado

Prof. Yadran Eterovic, Mario Droguett, Eduardo Bustos

Objetivos de la clase

- ☐ Conocer algoritmos de ordenación en tiempo lineal
- ☐ Comprender la limitación de dominio para tener tales algoritmos

Sumario

Ordenación lineal

Cierre

Ordenación en tiempo lineal

Consideremos una A secuencia de n naturales entre 0 y k

- Para todo $a_i \in A$, se tiene que $0 \le a_i \le k$
- Notemos que no necesariamente n = k 1
- La secuencia A puede tener elementos repetidos

Propondremos un algoritmo de ordenación que no compara

- Para cada dato, contaremos cuántos datos son menores que él
- Esto nos indica la posición final de cada elemento

Si $k \in \mathcal{O}(n)$, entonces este algoritmo será $\Theta(n)$

El algoritmo CountingSort()

```
input: Arreglo A[0...n-1], natural k
   output: Arreglo B[0...n-1]
   CountingSort (A, k):
        B[0...n-1] \leftarrow \text{arreglo vac\'io de } n \text{ celdas}
       C[0...k] \leftarrow \text{arreglo vac\'io de } k+1 \text{ celdas}
      for i = 0 ... k:
3
            C[i] \leftarrow 0
       for i = 0 ... n - 1:
5
            C[A[j]] \leftarrow C[A[j]] + 1
6
      for p = 1 \dots k:
7
            C[p] \leftarrow C[p] + C[p-1]
8
       for r = n - 1 \dots 0:
9
            B[C[A[r]] - 1] \leftarrow A[r]
10
            C[A[r]] \leftarrow C[A[r]] - 1
11
       return B
12
```

El algoritmo CountingSort()

```
CountingSort (A, k):
       B[0...n-1] \leftarrow \text{arreglo vacío}
       C[0...k] \leftarrow \text{arreglo vacío}
2
     for i = 0 ... k:
3
            C[i] \leftarrow 0

    La complejidad del algoritmo

       for j = 0 ... n - 1:
5
                                                                 es \Theta(n+k)
            C[A[i]] \leftarrow C[A[i]] + 1
       for p = 1 ... k:
7
                                                               ■ Si k \in \mathcal{O}(n), entonces
            C[p] \leftarrow C[p] + C[p-1]
                                                                 CountingSort() es \Theta(n)
       for r = n - 1 ... 0:
 g
            B[C[A[r]] - 1] \leftarrow A[r]
10
            C[A[r]] \leftarrow C[A[r]] - 1
11
       return B
12
```

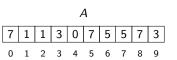
¡Este es un mejor tiempo que $\mathcal{O}(n\log(n))!$

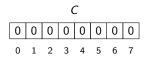
```
CountingSort (A, k):
       B[0...n-1] \leftarrow \text{arreglo vacío}
      C[0...k] \leftarrow \text{arreglo vacío}
2
    for i = 0 ... k:
            C[i] \leftarrow 0
 5
     for j = 0 ... n - 1:
                                                                           Α
            C[A[i]] \leftarrow C[A[i]] + 1
     for p = 1 ... k:
7
            C[p] \leftarrow C[p] + C[p-1]
8
       for r = n - 1 ... 0:
9
            B[C[A[r]] - 1] \leftarrow A[r]
10
            C[A[r]] \leftarrow C[A[r]] - 1
11
       return B
12
```

Hacemos el llamado CountingSort(A,7)

```
CountingSort (A, k):

B[0...n-1] \leftarrow \text{arreglo vac\'io}
C[0...k] \leftarrow \text{arreglo vac\'io}
for i = 0...k :
C[i] \leftarrow 0
```





```
CountingSort (A, k):
       B[0...n-1] \leftarrow \text{arreglo vacío}
1
      C[0...k] \leftarrow \text{arreglo vacío}
2
   for i = 0 ... k:
3
           C[i] \leftarrow 0
4
   for j = 0 ... n - 1:
5
                                                                           C
           C[A[j]] \leftarrow C[A[j]] + 1
6
                                                                           C
```

Hasta aquí, C[x] contiene el número de copias de x en A

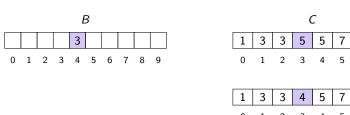
```
CountingSort (A, k):
                                                                            Α
       B[0...n-1] \leftarrow \text{arreglo vac\'io}
   C[0...k] \leftarrow \text{arreglo vacío}
2
      for i = 0 ... k:
3
           C[i] \leftarrow 0
4
    for j = 0 ... n - 1:
5
           C[A[j]] \leftarrow C[A[j]] + 1
6
       for p = 1 ... k:
7
           C[p] \leftarrow C[p] + C[p-1]
8
```

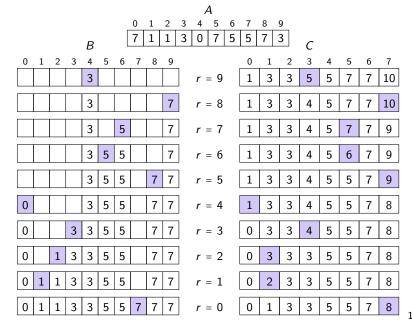
Hasta aquí, C[x] contiene cuántos elementos menores o iguales a x hay en A

9 for
$$r = n - 1 ... 0$$
:
10 $B[C[A[r]] - 1] \leftarrow A[r]$
11 $C[A[r]] \leftarrow C[A[r]] - 1$

Para r = 9







RadixSort

Otro algoritmo de ordenación en tiempo lineal es RadixSort

- Usado por las máquinas que ordenaban tarjetas perforadas
- Cada tarjeta tiene 80 columnas y 12 líneas
- Cada columna puede tener un agujero en una línea

RadixSort

El algoritmo ordena las tarjetas revisando una columna determinada

- Si hay un agujero en la columna, se pone en uno de los 12 compartimientos
- Las tarjetas con la perforación en la primera columna quedan encima
- La misma idea funciona para d columnas

Podemos generalizarla para un número natural de d dígitos

Ordenando por dígito

Consideremos un número natural de d dígitos $n_0 n_1 \cdots n_{d-1}$

- Podemos ordenar según el dígito más significativo n_0
- Según este dígito, separamos los números en compartimientos
- Luego, ordenados recursivamente cada compartimiento por su segundo dígito más significativo n₁
- Finalmente, combinamos los contenidos de cada compartimiento

Problema: posiblemente muchos llamados recursivos

Ordenación estable

Un ingrediente fundamental para el algoritmo que plantearemos es el siguiente

Definición

Dada una secuencia $A[0 \dots n-1]$, sea $B[0 \dots n-1]$ la secuencia resultante de ordenar A usando un algoritmo de ordenación $\mathcal S$. Sean a,a' elementos en A tales que para el algoritmo $\mathcal A$ son equivalentes y a aparece antes que a' en A. Diremos que $\mathcal S$ es estable si los elementos correspondientes b y b' aparecen en el mismo orden relativo en B.

Si ordenamos por el segundo dígito, un orden estable dejaría elementos que comparten segundo dígito en el mismo orden en que nos llegaron

RadixSort

El algoritmo RadixSort ordena por dígito menos significativo

- Ordena por dígito n_{d-1}
- Luego, usando el mismo arreglo, ordena por dígito n_{d-2} , con un algoritmo estable
- Luego de ordenar k dígitos, los datos están ordenados si solo miramos el fragmento $n_{d-k} \cdots n_{d-1}$
- Se requieren solo d pasadas para ordenar la secuencia completa

```
RadixSort(A, d):

for j = 0 \dots d - 1:

StableSort(A, j) \triangleright algoritmo de ordenación estable por j-ésimo dígito menos significativo
```

	Arreglo	Ordenado	Ordenado	Ordenado
	inicial	por unidad	d por decena	por centena
0	0 6 4	0 0 0	0 0 0	0 0 0
1	0 0 8	0 0 1	0 0 1	0 0 1
2	2 1 6	5 1 2	0 0 8	0 0 8
3	5 1 2	3 4 3	5 1 2	0 2 7
4	0 2 7	064	2 1 6	0 6 4
5	7 2 9	1 2 5	1 2 5	1 2 5
6	0 0 0	2 1 6	0 2 7	2 1 6
7	0 0 1	0 2 7	7 2 9	3 4 3
8	3 4 3	0 0 8	3 4 3	5 1 2
9	1 2 5	7 2 9	0 6 4	7 2 9
				-

RadixSort

```
\begin{aligned} & \mathtt{RadixSort}(A,d) \colon \\ & \textbf{for } j = 0 \dots d - 1 \colon \\ & \mathtt{StableSort}(A,j) & \rhd & \mathsf{algoritmo \ de \ ordenaci\'on \ estable \ por} \\ & & j\text{-\'esimo \ d\'igito \ menos \ significativo} \end{aligned}
```

Supongamos que A tiene n datos naturales con d dígitos y se usa algoritmo estable lineal

- Si cada dígito puede tomar k valores distintos...
- Entonces RadixSort toma tiempo $\Theta(d \cdot (n+k))$
- Si d es constante y $k \in \mathcal{O}(n)$, entonces RadixSort es $\Theta(n)$

Dos implementaciones

Estas ideas tiene dos implementaciones

LSD string sort (Least Significant Digit)

- Si todos los strings son del mismo largo (patentes, IP's, teléfonos)
- Funciona bien si el largo es pequeño
- No recursivo

MSD string sort (Most Significant Digit)

- Si los strings tienen largo diferente
- Ordenamos con CountingSort() por primer caracter
- Recursivamente ordenamos subarreglos correspondientes a cada caracter (excluyendo el primero, que es común en cada subarreglo)
- Como Quicksort, puede ordenar de forma independiente
- Pero particiona en tantos grupos como valores del primer caracter

MSD en acción

she	are	are	are	are
sells	by	by	by	by
seashells	she	sells	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	sells	sells
shore	s hore	s e ashells	sells	sells
the	s hells	she	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	the	the	the	the
seashells	the	the	the	the

Cuidados de MSD

En la ejecución de MSD string sort se debe considerar

Si un string s_1 es prefijo de otro s_2 , s_1 es menor que s_2

she ≤ shells

- Pueden usarse diferentes alfabetos
 - binario (2)
 - minúsculas (26)
 - minúsculas + mayúsculas + dígitos (64)
 - ASCII (128)
 - Unicode (65.536)
- Para subarreglos pequeños (e.g. $|A| \le 10$)
 - cambiar a InsertionSort que sepa que los primeros k caracteres son iguales

Sumario

Ordenación lineal

Cierre

Objetivos de la clase

- ☐ Conocer algoritmos de ordenación en tiempo lineal
- ☐ Comprender la limitación de dominio para tener tales algoritmos