Merge Sort

Clase 03

IIC 2133 - Sección 1

Prof. Sebastián Bugedo

Sumario

Obertura

Merge

Merge Sort

Epílogo

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	?	?	?	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	?	?	?	?
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	?	?	?	?
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

SelectionSort in place

input : Secuencia A, largo $n \geq 2$

output: $\varnothing$

```
SelectionSort (A, n):
    for i = 1 ... n − 1 :
        min = i
        for j = i + 1 ... n :
            if A[j] < A[min] :
                min = j
        Intercambiar A[i] con A[min]
```

SelectionSort e InsertionSort in place

```
SelectionSort (A, n):
    for i = 1 ... n − 1 :
        min = i
        for j = i + 1 ... n :
            if A[j] < A[min] :
                min = j
        Intrc (A[i], A[min])
```

```
InsertionSort (A, n):
    for i = 1 ... n − 1 :
        j = i
        while (j > 0) ∧ (A[j] < A[j − 1]) :
            Intrc (A[j], A[j − 1])
            j = j − 1
```

Ordenación hasta ahora

`SelectionSort`

- No tiene un mejor caso que sea *mejor* que su peor caso: $\mathcal{O}(n^2)$
- Siempre revisa la secuencia completa para determinar el mínimo

`InsertionSort`

- Cuando la secuencia está ordenada toma $\mathcal{O}(n)$
- En el caso promedio es $\mathcal{O}(n^2)$, tomando el promedio sobre todas las permutaciones igualmente probables
- Argumentamos esto mediante conteo de inversiones en cada permutación

¿Podemos tener un algoritmo de ordenación
con mejor complejidad que $\mathcal{O}(n^2)$ en el peor caso?

Hay esperanza: corregir varias inversiones a la vez

Ejemplo

Recordemos que el siguiente arreglo tiene 9 inversiones

0	1	2	3	4	5
34	8	64	51	32	21

Si intercambiamos 34 y 8:

0	1	2	3	4	5
8	34	64	51	32	21

- Corregimos $(0, 1)$

Si intercambiamos 34 y 21:

0	1	2	3	4	5
21	8	64	51	32	34

- Corregimos $(0, 4)$, $(0, 5)$, $(4, 5)$

Un escenario relacionado

Consideremos una secuencia parcialmente ordenada

Para ser más precisos, una secuencia que está formada por dos sub-secuencias ordenadas

8	20	29	40	50	60	70	82	15	32	41	65
0	1	2	3	4	5	6	7	8	9	10	11

Además, sabemos exactamente dónde comienza la segunda sub-secuencia ordenada

¿Cómo aprovechamos este hecho para ordenar la secuencia completa?

Primer intento: `InsertionSort`

Estado inicial

$$\overbrace{\begin{array}{|c|c|c|c|c|c|c|c|}\hline 8 & 20 & 29 & 40 & 50 & 60 & 70 & 82 \\ \hline \end{array}}^{A_1}\overbrace{\begin{array}{|c|c|c|c|}\hline 15 & 32 & 41 & 65 \\ \hline \end{array}}^{A_2}$$

8	20	29	40	50	60	70	82	15	32	41	65
0	1	2	3	4	5	6	7	8	9	10	11

`InsertionSort` no intercambia nada del tramo A_1 y los índices i, j llegan al tramo A_2

$j = 8$

$\downarrow$

8	20	29	40	50	60	70	82	15	32	41	65
0	1	2	3	4	5	6	7	8	9	10	11

$\uparrow$

$i = 8$

Hasta este punto la ejecución es $\mathcal{O}(n)$

Primer intento: `InsertionSort`

El valor 15 se va intercambiando hasta llegar a su posición final $j = 2$

$j = 1$

8	15	20	29	40	50	60	70	82	32	41	65
0	1	2	3	4	5	6	7	8	9	10	11

$i = 8$

En la siguiente iteración,

$j = 9$

8	15	20	29	40	50	60	70	82	32	41	65
0	1	2	3	4	5	6	7	8	9	10	11

$i = 9$

Conclusión: en este tramo el algoritmo vuelve a ser $\mathcal{O}(n^2)$

Hoy veremos una mejor estrategia para aprovechar el orden

Objetivos de la clase

- ☐ Comprender el algoritmo `Merge` para combinar secuencias ordenadas
- ☐ Determinar complejidad de `Merge` y el *trade off* de ejecutarlo *in place*
- ☐ Demostrar correctitud de `Merge`
- ☐ Comprender el uso de `Merge` como algoritmo de ordenación general en `MergeSort`
- ☐ Determinar la complejidad de `MergeSort`

Sumario

Obertura

Merge

Merge Sort

Epílogo

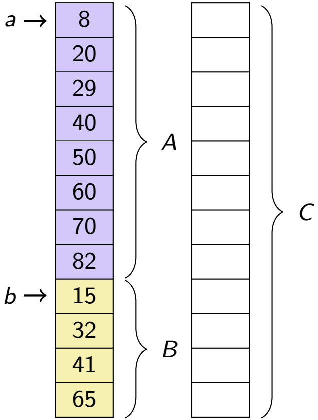
Mezcla (*merge*) de secuencias ordenadas

Proponemos el siguiente algoritmo para combinar dos secuencias ordenadas para formar una nueva **ordenada**

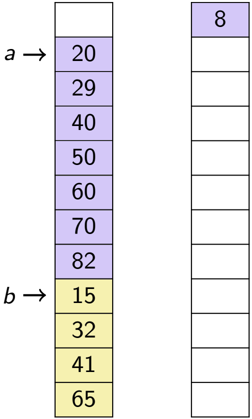
input : Secuencias ordenadas A y B
output: Nueva secuencia ordenada C

```
Merge(A, B):
1   Iniciamos C vacía
2   Sean a y b los primeros elementos de A y B
3   Extraer de su secuencia respectiva el menor entre a y b
4   Insertar el elemento extraído al final de C
5   Si quedan elementos en A y B, volver a línea 2
6   Concatenar C con la secuencia que aún tenga elementos
7   return C
```

`Merge`: Ejemplo de ejecución

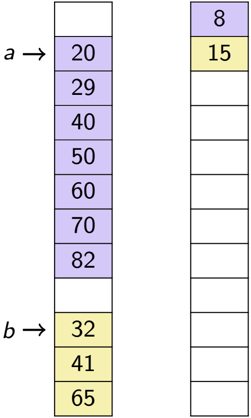


Estado inicial

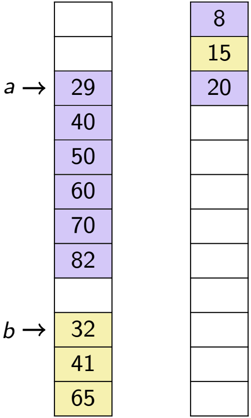


Estado luego de la primera iteración

Merge: Ejemplo de ejecución

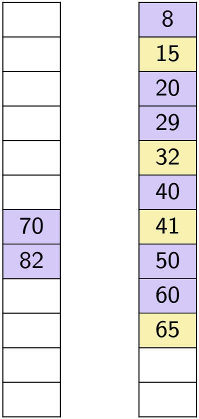


Estado luego de la
segunda iteración

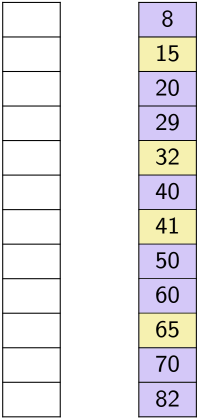


Estado luego de la
tercera iteración

Merge: Ejemplo de ejecución



Estado luego de insertar en C
el último elemento de B



Estado luego de
concatenar el resto de A

Correctitud de `Merge`

Demostración (finitud)

En cada iteración del algoritmo antes de ejecutar la línea 6, se extrae siempre un elemento de A o B, y se inserta en C.

Luego, cuando una de las secuencias se vacía, se insertan todos sus elementos en C.

En total se realizan $n = |A| + |B|$ inserciones y un número menor a n de comparaciones entre elementos. Luego, el algoritmo termina en una cantidad finita de pasos. $\square$

Correctitud de `Merge`

Demostración (propósito)

Para A, B inicialmente ordenadas, consideremos la propiedad

$$\begin{array}{lll}\mathbf{P(n)} & := & \text{Luego de insertar el } n\text{-ésimo elemento en } C, \\ & & A, B, C \text{ se encuentran ordenadas}\end{array}$$

1. **Caso base.** $\mathbf{P(1)}$ corresponde al estado de las secuencia luego de insertar el primer elemento en C.
   - Dado que se extrajo el menor elemento de alguna de las otras secuencias, estas se mantienen ordenadas. Esto aplica trivialmente si dicha secuencia queda vacía.
   - Dado que C solo tiene un elemento, está ordenada.

Correctitud de Merge

Demostración (propósito)

$$\mathbf{P}(\mathbf{n}) \quad := \quad \text{Luego de insertar el } n\text{ -ésimo elemento en } C, \; A, B, C \text{ se encuentran ordenadas}$$

2. **H.I.** Suponemos que luego de agregar el n-ésimo elemento, A, B, C están ordenadas.

   P.D. Luego de agregar el $(n+1)$-ésimo elemento, A, B, C siguen ordenadas.

   Tenemos dos casos

   - Si quedan elementos en A y en B, sea c_{n+1} el menor entre las cabezas de A y B.
   - Sin pérdida de generalidad, si solo quedan elementos en A, sea c_{n+1} la cabeza de A.

   Se elimina c_{n+1} de su secuencia respectiva y se inserta al final de C.

Correctitud de `Merge`

Demostración (propósito)

Por **H.I.** tenemos que la secuencia de origen de c_{n+1} se encontraba ordenada antes de sacarlo. Como es el mínimo de la secuencia por ser el primer elemento y ser una secuencia ordenada, se preserva el orden. Si la secuencia se vacía, también está ordenada.

Por **H.I.** tenemos que los primeros n elementos de C cumplen

$$c_1 \leq \cdots \leq c_n$$

Si c_{n+1} fuera estrictamente menor a alguno de estos elementos, implicaría una de las siguientes contradicciones

- A o B no están ordenadas (ya probamos que lo están)
- c_{n+1} es extraído en una iteración anterior por el criterio de selección

Luego, concluimos el resultado buscado

$$c_1 \leq \cdots \leq c_n \leq c_{n+1}$$

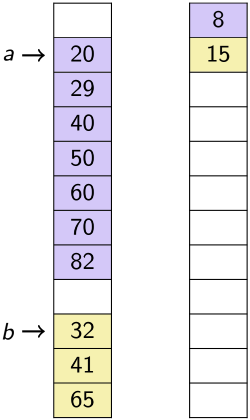
$\square$

Complejidad de `Merge`

Ejercicio

Determine la complejidad de memoria y tiempo de `Merge`

Complejidad de memoria de `Merge`



La ejecución de ejemplo que mostramos considera una nueva secuencia C donde se insertan los valores

- Para $|A| + |B| = n$ necesitamos memoria adicional $\mathcal{O}(n)$
- No necesita mover elementos dentro de ninguna secuencia

También se puede realizar *in place*

- Usar el mismo espacio reservado a A y B: memoria adicional $\mathcal{O}(1)$
- Mover todos los datos mayores al insertado
- Impacta en la complejidad de tiempo...

Complejidad de tiempo de `Merge`

Consideramos la implementación sugerida mediante una secuencia **adicional**

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
   - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
   - Se inserta el dato en C $\mathcal{O}(1)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$
2. Reubicación de la secuencia no vacía restante
   - Se saca un elemento de la restante $\mathcal{O}(1)$
   - Se inserta el dato en C $\mathcal{O}(1)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n)$

Usando $\mathcal{O}(n)$ memoria adicional, `Merge` es $\mathcal{O}(n)$

Complejidad de tiempo de `Merge`

Si consideramos usar el espacio reservado para A y B

El algoritmo tiene dos fases

1. Extracción desde ambas secuencias A y B
   - Se decide quién extraer comparando los menores $\mathcal{O}(1)$
   - Se inserta el dato en C $\mathcal{O}(n)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n^2)$
2. Reubicación de la secuencia no vacía restante
   - Se saca un elemento de la restante $\mathcal{O}(1)$
   - Se inserta el dato en C $\mathcal{O}(n)$
   - Esto se repite $\mathcal{O}(n)$ veces total $\mathcal{O}(n^2)$

Usando $\mathcal{O}(1)$ memoria adicional, `Merge` es $\mathcal{O}(n^2)$

Complejidad de tiempo de `Merge`

- Tenemos un algoritmo **lineal** para obtener una secuencia ordenada
- Pero el requisito de las sub-secuencias ordenadas es demasiado exigente

¿Podemos usar `Merge` para ordenar una secuencia arbitraria?

- Dada una secuencia arbitraria
- Estamos listos si logramos crear dos sub-secuencias ordenadas a partir de ella
- Luego las combinamos con `Merge`

Sumario

Obertura

Merge

Merge Sort

Epílogo

Dividir para conquistar

El plan para usar `Merge` en un algoritmo de ordenación sigue la estrategia **dividir para conquistar**

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o más) **sub-problemas** del mismo tipo
2. Resolver **recursivamente** cada sub-problema
3. Encontrar solución al problema original **combinando** las soluciones a los sub-problemas

Los sub-problemas son instancias más pequeñas del problema a resolver

Dividir para conquistar y `Merge`

Podemos usar la estrategia **dividir para conquistar** en el problema de ordenación, usando `Merge`

¿En qué parte del dividir para conquistar usaremos `Merge`?

La idea general para ordenar usando `Merge`define un nuevo algoritmo que llamaremos `MergeSort`

1. Dividir la secuencia original en dos sub-secuencias
2. Llamamos recursivamente a `MergeSort` sobre las dos sub-secuencias
3. Combinamos las secuencias ordenadas resultantes mediante `Merge`

El algoritmo `MergeSort`

A continuación tenemos el pseudocódigo del algoritmo recursivo `MergeSort`

input : Secuencia A
output: Secuencia ordenada B

`MergeSort` *(A)*:

  if $|A| = 1$ **: return** A
  Dividir A en mitades A_1 y A_2
  $B_1 \leftarrow \texttt{MergeSort}(A_1)$
  $B_2 \leftarrow \texttt{MergeSort}(A_2)$
  $B \leftarrow \texttt{Merge}(B_1, B_2)$
  return B

MergeSort: Ejemplo de ejecución

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

Correctitud de `MergeSort`

Ejercicio (propuesto)

Demuestre que `MergeSort` es correcto

input : Secuencia A
output: Secuencia ordenada B

`MergeSort` *(A)*:

    if $|A| = 1$ **: return** A
    Dividir A en mitades A_1 y A_2
    $B_1 \leftarrow \texttt{MergeSort}(A_1)$
    $B_2 \leftarrow \texttt{MergeSort}(A_2)$
    $B \leftarrow \texttt{Merge}(B_1, B_2)$
    return B

Carácter recursivo de MergeSort

input : Secuencia A
output: Secuencia ordenada B

```
MergeSort (A):
1    if |A| = 1 : return A
2    Dividir A en mitades A_1 y A_2
3    B_1 ← MergeSort(A_1)
4    B_2 ← MergeSort(A_2)
5    B ← Merge(B_1, B_2)
6    return B
```

Todo algoritmo recursivo debe chequear primero el **caso base**

- Es el caso cuya solución no requiere recursión
- En `MergeSort`: línea 1

Los **llamados recursivos** se hacen sobre casos distintos al original

- Se acercan un poco más al caso base
- En `MergeSort`: líneas 3 y 4

Complejidad de `MergeSort`

Para el análisis de complejidad de tiempo, definimos

$$T(n) := \# \text{ pasos para ordenar } n \text{ elementos}$$

Con esto, consideramos los dos casos posibles al llamar a `MergeSort`

```
MergeSort (A):
    if |A| = 1 : return A
    Dividir A en A1 y A2
    B1 ← MergeSort(A1)
    B2 ← MergeSort(A2)
    B ← Merge(B1, B2)
    return B
```

- Si $n = 1$, aplica el caso base y solo involucra un paso

$$T(1) = 1$$

- Si $n > 1$, aplican los llamados
  - Dos llamados de tamaño $n/2$
  - Llamado a `Merge`

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

Este análisis aplica **para toda** secuencia de input:
Nos entregará el resultado de peor, mejor y caso promedio

Complejidad de `MergeSort`

La siguiente relación es una **relación de recurrencia**

$$T(1) = 1, \quad T(n) = 2T\left(\frac{n}{2}\right) + n$$

Podemos resolverla notando que la parte recursiva puede ser reescrita como

$$T(n) = 2T(n/2) + n \quad \Leftrightarrow \quad \frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

La gracia de esta expresión es que numeradores y denominadores incluyen la misma fracción de n

Sin pérdida de generalidad, suponemos que n es potencia de 2

Complejidad de `MergeSort`

Construimos un sistema de ecuaciones reemplazando el argumento del lado izquierdo por $n, n/2, n/4, \ldots, 2$ de forma que el último término contiene $T(1)$ (nuestro caso base)

$$\begin{array}{lccc}
\text{ecuación } 1 & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} + 1 \\
\text{ecuación } 2 & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} + 1 \\
 & & \cdots & \\
\text{ecuación } k & \dfrac{T(2)}{2} & = & \dfrac{T(1)}{1} + 1
\end{array}$$

Como el lado derecho de la i-ésima ecuación considera la potencia 2^i, de la k-ésima ecuación deducimos

$$1 = \frac{n}{2^k} \quad \Rightarrow \quad 2^k = n \quad \Rightarrow \quad k = \log(n)$$

Complejidad de MergeSort

Sumamos las $\log(n)$ ecuaciones y simplificamos los términos que aparecen a ambos lados

$$
\begin{array}{lcccc}
\text{ecuación } 1 & \dfrac{T(n)}{n} & = & \dfrac{T(n/2)}{n/2} & +1 \\
\text{ecuación } 2 & \dfrac{T(n/2)}{n/2} & = & \dfrac{T(n/4)}{n/4} & +1 \\
 & & \cdots & & \\
\text{ecuación } k & \dfrac{T(2)}{2} & = & \dfrac{T(1)}{1} & +1 \\
\text{suma} & \dfrac{T(n)}{n} & = & \dfrac{T(1)}{1} & +\log(n)
\end{array}
$$

Despejando, obtenemos $T(n) = n\log(n) + n$

La complejidad de tiempo de MergeSort es $\mathcal{O}(n\log(n))$

Complejidad de `MergeSort`

En términos de memoria adicional

```
MergeSort (A):
1   if |A| = 1 : return A
2   Dividir A en A1 y A2
3   B1 ← MergeSort(A1)
4   B2 ← MergeSort(A2)
5   B ← Merge(B1, B2)
6   return B
```

- El paso recursivo no ocupa memoria adicional
- Para $|A| = n$, la línea 5 ocupa $\mathcal{O}(n)$
- Ojo! Los llamados recursivos no van acumulando memoria reservada, por lo que no sumamos $\mathcal{O}(n)$ por llamado

La complejidad de memoria de `MergeSort` es $\mathcal{O}(n)$

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

Notemos la mejora en tiempo con `MergeSort`
a cambio de memoria adicional

Sumario

Obertura

Merge

Merge Sort

Epílogo

Ideas al cierre

- Existen algoritmos en que podemos mejorar el tiempo si ocupamos más memoria adicional (*trade off*)
- `Merge` permite ordenar secuencias con características muy específicas
- `Merge` es $\mathcal{O}(n)$ cuando se ocupa $\mathcal{O}(n)$ memoria
- La estrategia dividir para conquistar requiere dividir el problema en subproblemas cuyos resultados se puedan combinar
- `MergeSort` es un ejemplo de la estrategia dividir para conquistar
- `MergeSort` es la extensión recursiva que permite usar `Merge` en un algoritmo de ordenación $\mathcal{O}(n \log(n))$

Epílogo

Ve a

www.menti.com

Introduce el código

4381 9803

O usa el código QR