

Complejidad algorítmica

Giovanni Ramírez García, PhD

Escuela de Ciencias Físicas y Matemáticas
Universidad de San Carlos de Guatemala

Guatemala, 29 de enero de 2021



Introducción

Complejidad computacional

Clases de Complejidad

Algoritmos de ordenamiento

Algoritmo de burbuja

Merge Sort

Introducción

Complejidad computacional

Clases de Complejidad

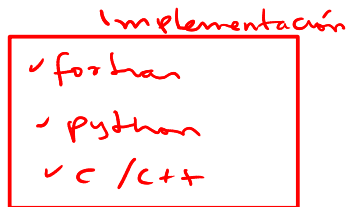
Algoritmos de ordenamiento

Algoritmo de burbuja

Merge Sort

¿Cómo elijo un algoritmo?

- ▶ Lo más importante, en el momento del diseño e implementación, es que se debe resolver el problema.
- ▶ Algunas veces el rendimiento es un aspecto importante de la exactitud.
- ▶ Escribir un programa eficiente no es fácil [Guttag, 2013]



¿Cómo elijo un algoritmo?

- ▶ Lo más importante, en el momento del diseño e implementación, es que se debe resolver el problema.
- ▶ Algunas veces el rendimiento es un aspecto importante de la exactitud.
- ▶ Escribir un programa eficiente no es fácil [Guttag, 2013]
- ▶ La solución más sencilla no siempre es la más eficiente.
- ▶ Los algoritmos computacionalmente eficientes usualmente usan trucos que los hacen difíciles de entender.
- ▶ A esto se le conoce como incremento de la complejidad conceptual para reducir la complejidad computacional.

Introducción

Complejidad computacional

Clases de Complejidad

Algoritmos de ordenamiento


Algoritmo de burbuja

Merge Sort

Complejidad computacional

- ¿en cuánto tiempo se ejecuta este código?

```
DO K=1,N
  DO J=1,N
    DO I=1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    END DO
  END DO
END DO
```



Complejidad computacional

- ▶ ¿en cuánto tiempo se ejecuta este código?
- ▶ ¿depende de la computadora que se usa?

```
DO K=1,N
  DO J=1,N
    DO I=1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    END DO
  END DO
END DO
```


Complejidad computacional

swap

- ▶ ¿en cuánto tiempo se ejecuta este código?
- ▶ ¿depende de la computadora que se usa?
- ▶ ¿depende del sistema operativo?

```
DO K=1,N
  DO J=1,N
    DO I=1,N
      C(I,J)=C(I,J)+A(I,K)*B(K,J)
    END DO
  END DO
END DO
```

Complejidad computacional

“indicador”
Vamos a usar tiempo como “testigo” de la Complejidad Computacional

► ¿en cuánto tiempo se ejecuta este código?

► ¿depende de la computadora que se usa?

► ¿depende del sistema operativo?

► ¿depende del tamaño de las matrices?

DO K=1,N

DO J=1,N

DO I=1,N

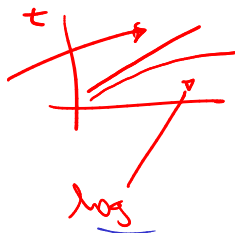
C(I,J)=C(I,J)+A(I,K)*B(K,J)

END DO

END DO

END DO

lineal



- ▶ *Random Access Machine* modelo de cómputo donde los pasos se ejecutan secuencialmente.

Random Access Machine

- ▶ *Random Access Machine* modelo de cómputo donde los pasos se ejecutan secuencialmente.
- ▶ Un paso es una operación que toma una cantidad fija de tiempo, por ejemplo: una comparación, una operación aritmética, acceso a memoria.

Random Access Machine

$$a = b$$
$$a + b$$

- ▶ *Random Access Machine* modelo de cómputo donde los pasos se ejecutan secuencialmente.
- ▶ Un paso es una operación que toma una cantidad fija de tiempo, por ejemplo: una comparación, una operación aritmética, acceso a memoria.
- ▶ Con este modelo, nuestra medida sólo depende del tamaño de la entrada y la medida del tiempo que se tarda es una buena aproximación

- ▶ *Random Access Machine* modelo de cómputo donde los pasos se ejecutan secuencialmente.
- ▶ Un paso es una operación que toma una cantidad fija de tiempo, por ejemplo: una comparación, una operación aritmética, acceso a memoria.
- ▶ Con este modelo, nuestra medida sólo depende del tamaño de la entrada y la medida del tiempo que se tarda es una buena aproximación.
- ▶ mejor caso: cuando las entradas son tan favorables como sea posible.

Random Access Machine

- ▶ *Random Access Machine* modelo de cómputo donde los pasos se ejecutan secuencialmente.
- ▶ Un paso es una operación que toma una cantidad fija de tiempo, por ejemplo: una comparación, una operación aritmética, acceso a memoria.
- ▶ Con este modelo, nuestra medida sólo depende del tamaño de la entrada y la medida del tiempo que se tarda es una buena aproximación.

- ▶ **mejor caso:** cuando las entradas son tan favorables como sea posible.
- ▶ **peor caso:** cuando las entradas no son favorables y obtenemos el peor tiempo.

4, 3, 2
3, 4, 2 →

3, 2, 4
2, 3, 4
5 pasos

Random Access Machine

- ▶ *Random Access Machine* modelo de cómputo donde los pasos se ejecutan secuencialmente.
- ▶ Un paso es una operación que toma una cantidad fija de tiempo, por ejemplo: una comparación, una operación aritmética, acceso a memoria.
- ▶ Con este modelo, nuestra medida sólo depende del tamaño de la entrada y la medida del tiempo que se tarda es una buena aproximación.
- ▶ **mejor caso:** cuando las entradas son tan favorables como sea posible.
- ▶ **peor caso:** cuando las entradas no son favorables y obtenemos el peor tiempo.
- ▶ **caso promedio:** es un tiempo promedio sobre todas las posibles entradas de un mismo tamaño.

$$\frac{2+5}{2} = \frac{7}{2}$$

3.5 pasos

$$f(x) = P_n(x) + \underline{O}(x^{n+1})$$

→ Función factorial

- Entrada: número natural, n
- Salida: factorial de la entrada,
 $n! = n(n-1)(n-2) \cdots (2)(1)$

1. $factorial = 1$
2. MIENTRAS QUE $n > 1$ HACER
 - 2.1 $factorial = factorial * n$
 - 2.2 $n = n - 1$
3. FIN

$n=3, factorial = 6$

1. $factorial = 1$

2. $factorial = (1) * (3) = 3$

2.2 $n = 2$

2.1 $factorial = (3) * (2) = 6$

2.2 $n = 1$

3.

Notación asintótica

Función factorial

Entrada: número natural, n

Salida: factorial de la entrada,

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

1. $factorial = 1$ ✓
2. MIENTRAS QUE $n > 1$ HACER
 - 2.1 $factorial = factorial * n$
 - 2.2 $n = n - 1$
3. FIN

Conteo de pasos

- 1 paso para asignar el primer valor

Notación asintótica

Función factorial

Entrada: número natural, n

Salida: factorial de la entrada,

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

1. $factorial = 1$
2. MIENTRAS QUE $n > 1$ HACER
 - 2.1 $factorial = factorial * n$
 - 2.2 $n = n - 1$
3. FIN

Conteo de pasos

- ▶ 1 paso para asignar el primer valor
- ▶ n pasos para la comparación en el ciclo

Notación asintótica

Función factorial

Entrada: número natural, n

Salida: factorial de la entrada,

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

1. $factorial = 1$
2. MIENTRAS QUE $n > 1$ HACER
 - 2.1 $factorial = factorial * n$ ✓
 - 2.2 $n = n - 1$ ✓
3. FIN

Conteo de pasos

- ▶ 1 paso para asignar el primer valor
- ▶ n pasos para la comparación en el ciclo
- ▶ $2n$ pasos por el acceso a memoria y la operación aritmética

Notación asintótica

Función factorial

Entrada: número natural, n

Salida: factorial de la entrada,

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

1. $factorial = 1$
2. MIENTRAS QUE $n > 1$ HACER
 - 2.1 $factorial = factorial * n$
 - 2.2 $n = n - 1$
3. FIN

Conteo de pasos

- ▶ 1 paso para asignar el primer valor
- ▶ n pasos para la comparación en el ciclo
- ▶ $2n$ pasos por el acceso a memoria y la operación aritmética
- ▶ $2n$ pasos por el acceso a memoria y la operación aritmética

Notación asintótica

Función factorial

Entrada: número natural, n

Salida: factorial de la entrada,

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

1. $factorial = 1$
2. MIENTRAS QUE $n > 1$ HACER
 - 2.1 $factorial = factorial * n$
 - 2.2 $n = n - 1$
3. FIN

Conteo de pasos

- ▶ ~~1~~ paso para asignar el primer valor
- ▶ ~~n~~ pasos para la comparación en el ciclo
- ▶ ~~2~~ n pasos por el acceso a memoria y la operación aritmética
- ▶ ~~2~~ n pasos por el acceso a memoria y la operación aritmética
- ▶ TOTAL: $5n + 1$ ✓

Notación asintótica

Función factorial

Entrada: número natural, n

Salida: factorial de la entrada,

$$n! = n(n-1)(n-2) \cdots (2)(1)$$

1. $factorial = 1$
2. MIENTRAS QUE $n > 1$ HACER
 - 2.1 $factorial = factorial * n$
 - 2.2 $n = n - 1$
3. FIN

Conteo de pasos

- ▶ 1 paso para asignar el primer valor
- ▶ n pasos para la comparación en el ciclo
- ▶ $2n$ pasos por el acceso a memoria y la operación aritmética
- ▶ $2n$ pasos por el acceso a memoria y la operación aritmética
- ▶ TOTAL: $5n + 1$
- ▶ es decir $O(n)$ ←
Para $n \gg 1$

Introducción

Complejidad computacional

Clases de Complejidad

Algoritmos de ordenamiento

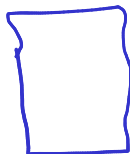
Algoritmo de burbuja

Merge Sort

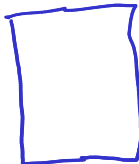
Clases de complejidad (I)

- ▶ $O(1)$: tiempo constante
- ▶ $O(\log(n))$: tiempo logarítmico
- ▶ $O(n)$: tiempo lineal
- ▶ $O(n \log(n))$: tiempo log-lineal
- ▶ $O(n^k)$: tiempo polinómico
- ▶ $O(c^n)$: tiempo exponencial

c. alg



c. comp.



ordenar números
enteros

merge-sort $\rightarrow n \log(n)$

burbuja $\rightarrow n^2$

Clases de complejidad (I)

- ▶ $O(1)$: tiempo constante
- ▶ $O(\log(n))$: tiempo logarítmico
- ▶ $O(n)$: tiempo lineal
- ▶ $O(n \log(n))$: tiempo log-lineal
- ▶ $O(n^k)$: tiempo polinómico
- ▶ $O(c^n)$: tiempo exponencial
- ▶ Complejidad independiente de las entradas. ✓
- ▶ Códigos con ciclos independientes de las entradas. ✓
- ▶ Códigos con llamadas recursivas independientes de las entradas. ✓

Clases de complejidad (II)

- ▶ $O(1)$: tiempo constante
- ▶ $O(\log(n))$: tiempo logarítmico
- ▶ $O(n)$: tiempo lineal
- ▶ $O(n \log(n))$: tiempo log-lineal
- ▶ $O(n^k)$: tiempo polinómico
- ▶ $O(c^n)$: tiempo exponencial

Clases de complejidad (II)


- ▶ $O(1)$: tiempo constante
 - ▶ $O(\log(n))$: tiempo logarítmico
 - ▶ $O(n)$: tiempo lineal
 - ▶ $O(n \log(n))$: tiempo log-lineal
 - ▶ $O(n^k)$: tiempo polinómico
 - ▶ $O(c^n)$: tiempo exponencial
- ▶ Aplicaciones como la búsqueda binaria, el método de bisección. ✓
 - ▶ no importa la base del logaritmo

Clases de complejidad (III)

- ▶ $O(1)$: tiempo constante
- ▶ $O(\log(n))$: tiempo logarítmico
- ▶ $O(n)$: tiempo lineal
- ▶ $O(n \log(n))$: tiempo log-lineal
- ▶ $O(n^k)$: tiempo polinómico
- ▶ $O(c^n)$: tiempo exponencial

Clases de complejidad (III)

2, 1, 2, 3, 4, 2



- ▶ $O(1)$: tiempo constante
 - ▶ $O(\log(n))$: tiempo logarítmico
 - ▶ $O(n)$: tiempo lineal
 - ▶ $O(n \log(n))$: tiempo log-lineal
 - ▶ $O(n^k)$: tiempo polinómico
 - ▶ $O(c^n)$: tiempo exponencial
- ▶ Aparece varios problemas que implican listas, por ejemplo una búsqueda en una lista ordenada
 - ▶ Puede haber ciclos, pero que no son dependientes del tamaño de la entrada

Clases de complejidad (IV)

- ▶ $O(1)$: tiempo constante
- ▶ $O(\log(n))$: tiempo logarítmico
- ▶ $O(n)$: tiempo lineal
- ▶ $O(n \log(n))$: tiempo log-lineal
- ▶ $O(n^k)$: tiempo polinómico
- ▶ $O(c^n)$: tiempo exponencial

Clases de complejidad (IV)

- ▶ $O(1)$: tiempo constante
 - ▶ $O(\log(n))$: tiempo logarítmico
 - ▶ $O(n)$: tiempo lineal
 - ▶ $O(n \log(n))$: tiempo log-lineal
 - ▶ $O(n^k)$: tiempo polinómico
 - ▶ $O(c^n)$: tiempo exponencial
- ▶ Muchos de algoritmos prácticos están en esta clase.
 - ▶ Por ejemplo el algoritmo de ordenamiento merge-sort

Clases de complejidad (V)

- ▶ $O(1)$: tiempo constante
- ▶ $O(\log(n))$: tiempo logarítmico
- ▶ $O(n)$: tiempo lineal
- ▶ $O(n \log(n))$: tiempo log-lineal
- ▶ $O(n^k)$: tiempo polinómico
- ▶ $O(c^n)$: tiempo exponencial

Clases de complejidad (V)

- ▶ $O(1)$: tiempo constante
 - ▶ $O(\log(n))$: tiempo logarítmico
 - ▶ $O(n)$: tiempo lineal
 - ▶ $O(n \log(n))$: tiempo log-lineal
 - ▶ $O(n^k)$: tiempo polinómico, $k: \text{cte}$
 - ▶ $O(c^n)$: tiempo exponencial
- ▶ Otros algoritmos prácticos están en esta clase, en especial de orden cuadrático. $k=2$, $k=4$
 - ▶ Por ejemplo: el algoritmo para verificar si un conjunto es subconjunto de otro, bubble sort

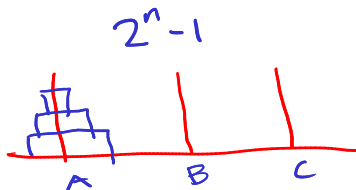
Clases de complejidad (VI)

- ▶ $O(1)$: tiempo constante
- ▶ $O(\log(n))$: tiempo logarítmico
- ▶ $O(n)$: tiempo lineal
- ▶ $O(n \log(n))$: tiempo log-lineal
- ▶ $O(n^k)$: tiempo polinómico
- ▶ $O(c^n)$: tiempo exponencial

Clases de complejidad (VI)

- ▶ $O(1)$: tiempo constante
- ▶ $O(\log(n))$: tiempo logarítmico
- ▶ $O(n)$: tiempo lineal
- ▶ $O(n \log(n))$: tiempo log-lineal
- ▶ $O(n^k)$: tiempo polinómico
- ▶ $O(c^n)$: tiempo exponencial

- ▶ muchos problemas importantes son exponenciales, por ejemplo generar el conjunto potencia, las Torres de Hanoi



Puntos a tomar en cuenta

Man

- ▶ Tener en cuenta que esta aproximación de complejidad usando el tiempo es distinta a la aproximación del espacio necesario para un algoritmo.
- ▶ Por ejemplo, en una función recursiva el espacio aumenta pues se reserva una nueva *pila* por cada llamada recursiva.

$n = 2 \times 2$
 $n = 3 \times 3$
⋮

Which time
/usr/bin/time

- ▶ En GNU/Linux se puede medir la memoria RAM usada y el porcentaje de procesador.
- ▶ Ejemplo: uso de time

```
▶ time sleep 3  
▶ /usr/bin/time -f "%e%M%P" sleep 3  
▶ /usr/bin/time -f "%e%M%P" -o log sleep  
3
```

"%e%M%P" ->
append

%e : tiempo (s)
%M : memoria (kb)
%P : porcentaje de uso

Introducción

Complejidad computacional

Clases de Complejidad

Algoritmos de ordenamiento

Algoritmo de burbuja

Merge Sort

Algoritmos de ordenamiento

Son algoritmos que se usan para ordenar números enteros en una secuencia (ordenamiento numérico) o para ordenar letras (ordenamiento lexicográfico)

ordenamiento numérico

→ {5, 3, 2, 6}

- {2, 3, 5, 6} ✓

ordenamiento lexicográfico

→ {a, z, m, p}

{a, m, p, z} ✓

Existen varios tipos: *partitioning*,
merging, *selection*, *insertion*,
exchanging

Algoritmos de ordenamiento

Son algoritmos que se usan para ordenar números enteros en una secuencia (ordenamiento numérico) o para ordenar letras (ordenamiento lexicográfico)

ordenamiento numérico

$\{5, 3, 2, 6\}$

$\{2, 3, 5, 6\}$

ordenamiento lexicográfico

$\{a, z, m, p\}$

$\{a, m, p, z\}$

Existen varios tipos: *partitioning*,
merging, *selection*, *insertion*,
exchanging

Algunos algoritmos (de una lista de 24 en Wikipedia):

- ▶ Insertion sort
- ▶ Selection sort
- ▶ Merge sort
- ▶ Heapsort
- ▶ Quicksort
- ▶ Bubble sort
- ▶ Comb sort
- ▶ Counting sort
- ▶ Bucket sort

Algoritmos de ordenamiento

Son algoritmos que se usan para ordenar números enteros en una secuencia (ordenamiento numérico) o para ordenar letras (ordenamiento lexicográfico)

ordenamiento numérico

{5, 3, 2, 6}

{2, 3, 5, 6}

ordenamiento lexicográfico

{a, z, m, p}

{a, m, p, z}

Existen varios tipos: *partitioning*,
merging, *selection*, *insertion*,
exchanging

Algunos algoritmos (de una lista de 24 en Wikipedia):

► Insertion sort

► Selection sort

Simples

► Merge sort

► Heapsort

► Quicksort

► Bubble sort

► Comb sort

► Counting sort

► Bucket sort

Algoritmos de ordenamiento

Son algoritmos que se usan para ordenar números enteros en una secuencia (ordenamiento numérico) o para ordenar letras (ordenamiento lexicográfico)

ordenamiento numérico

$\{5, 3, 2, 6\}$

$\{2, 3, 5, 6\}$

ordenamiento lexicográfico

$\{a, z, m, p\}$

$\{a, m, p, z\}$

Existen varios tipos: *partitioning*,
merging, *selection*, *insertion*,
exchanging

Algunos algoritmos (de una lista de 24 en Wikipedia):

► Insertion sort

► Selection sort

► Merge sort ✓

► Heapsort

► Quicksort

► Bubble sort

► Comb sort

► Counting sort

► Bucket sort

Eficientes

Algoritmos de ordenamiento

Son algoritmos que se usan para ordenar números enteros en una secuencia (ordenamiento numérico) o para ordenar letras (ordenamiento lexicográfico)

ordenamiento numérico

{5, 3, 2, 6}

{2, 3, 5, 6}

ordenamiento lexicográfico

{a, z, m, p}

{a, m, p, z}

Existen varios tipos: *partitioning*,
merging, *selection*, *insertion*,
exchanging

Algunos algoritmos (de una lista de 24 en Wikipedia):

► Insertion sort

► Selection sort

► Merge sort

► Heapsort

► Quicksort

► Bubble sort ✓

► Comb sort ✓

► Counting sort

► Bucket sort

Burbuja

Algoritmos de ordenamiento

Son algoritmos que se usan para ordenar números enteros en una secuencia (ordenamiento numérico) o para ordenar letras (ordenamiento lexicográfico)

ordenamiento numérico

$\{5, 3, 2, 6\}$

$\{2, 3, 5, 6\}$

ordenamiento lexicográfico

$\{a, z, m, p\}$

$\{a, m, p, z\}$

Existen varios tipos: *partitioning*,
merging, *selection*, *insertion*,
exchanging

Algunos algoritmos (de una lista de 24 en Wikipedia):

- ▶ Insertion sort
- ▶ Selection sort
- ▶ Merge sort
- ▶ Heapsort
- ▶ Quicksort
- ▶ Bubble sort
- ▶ Comb sort
- ▶ Counting sort
- ▶ Bucket sort

Distribución

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

Algoritmo de Burbuja o *Bubble Sort*

$\{4, 2, 6, 1\}$

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

$\{4, 2, 6, 1\}$

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

$\{4, 2, 6, 1\}$

$\{2, 4, 6, 1\}$

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

$\{4, 2, 6, 1\}$

$\{2, 4, 6, 1\}$

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

$\{4, 2, 6, 1\}$

$\{2, 4, 6, 1\}$

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

{4, 2, 6, 1}

{2, 4, 6, 1}

{2, 4, 1, 6}

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

{4, 2, 6, 1}

{2, 4, 6, 1}

{2, 4, 1, 6}

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

{4, 2, 6, 1}

{2, 4, 6, 1}

{2, 4, 1, 6}

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

$\{4, 2, 6, 1\}$
 $\{2, 4, 6, 1\}$
 $\{2, 4, 1, 6\}$
 $\{2, 1, 4, 6\}$

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

$\{4, 2, 6, 1\}$
 $\{2, 4, 6, 1\}$
 $\{2, 4, 1, 6\}$
 $\{2, 1, 4, 6\}$

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

$\{4, 2, 6, 1\}$
 $\{2, 4, 6, 1\}$
 $\{2, 4, 1, 6\}$
 $\{2, 1, 4, 6\}$

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

$\{4, 2, 6, 1\}$
 $\{2, 4, 6, 1\}$
 $\{2, 4, 1, 6\}$
 $\{2, 1, 4, 6\}$
 $\{1, 2, 4, 6\}$

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

$\{4, 2, 6, 1\}$
 $\{2, 4, 6, 1\}$
 $\{2, 4, 1, 6\}$
 $\{2, 1, 4, 6\}$
 $\{1, 2, 4, 6\}$

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

{4, 2, 6, 1}
{2, 4, 6, 1}
{2, 4, 1, 6}
{2, 1, 4, 6}
{1, 2, 4, 6}

Algoritmo de Burbuja o *Bubble Sort*

- ▶ Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.
- ▶ El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

{4, 2, 6, 1}

{2, 4, 6, 1}

{2, 4, 1, 6}

{2, 1, 4, 6}

{1, 2, 4, 6}

la cadena está ordenada

Algoritmo de Burbuja o *Bubble Sort*

- Consiste en ir examinando la lista, tomando pares de números y comparándolos para ordenarlos haciendo un intercambio o *swap* si es necesario.

- El proceso se repite hasta que los números están ordenados.

Es un algoritmo del tipo de ordenamiento por comparación o *comparison sort*.

{4, 2, 6, 1} $n = 4$

{2, 4, 6, 1}

{2, 4, 1, 6}

{2, 1, 4, 6}

{1, 2, 4, 6}

la cadena está ordenada

- Mejor caso: n

- Caso promedio: n^2

- Peor caso: n^2

} $O(n)$
 $O(n^2)$
 $n \gg 1$

Complejidad del algoritmo bubblesort

Algoritmo

Entrada: Lista de números enteros de longitud n .

Salida: Lista de números ordenados

1. $swap = \text{TRUE}$. { *boolean variable* }
2. MIENTRAS QUE $swap = \text{TRUE}$. HACER
 - 2.1 $swap = \text{FALSE}$.
 - 2.2 PARA $1 < i < n - 1$ HACER
 - 2.2.1 SI $\text{Lista}(i) > \text{Lista}(i+1)$ HACER
 - (a) intercambiar los números
 - (b) $swap = \text{TRUE}$.
3. FIN

► Mejor caso: n

► Caso promedio: n^2

► Peor caso: n^2

Optimización: después de m revisiones de la lista, hay m números ordenados.

Optimización: sólo revisar hasta la posición donde se hizo el último swap.

Algoritmo Merge Sort

n :

- ▶ Se divide la lista original en n sublistas ordenadas.
- ▶ Luego se unen o concatenan en $n/2$ sublistas ordenadas.
- ▶ Luego en $n/4$ sublistas ordenadas.
- ▶ Se repite hasta que queda una sola lista.

Es un algoritmo del tipo *divide y vencerás* y su invención se atribuye a John von Neumann en 1945.

Algoritmo Merge Sort

$\{4, 2, 6, 1\}$

- ▶ Se divide la lista original en n sublistas ordenadas.
- ▶ Luego se unen o concatenan en $n/2$ sublistas ordenadas.
- ▶ Luego en $n/4$ sublistas ordenadas.
- ▶ Se repite hasta que queda una sola lista.

Es un algoritmo del tipo *divide y vencerás* y su invención se atribuye a John von Neumann en 1945.

Algoritmo Merge Sort

- ▶ Se divide la lista original en n sublistas ordenadas.
- ▶ Luego se unen o concatenan en $n/2$ sublistas ordenadas.
- ▶ Luego en $n/4$ sublistas ordenadas.
- ▶ Se repite hasta que queda una sola lista.

$\{4, 2, 6, 1\}$
 $\{4, 2\}; \{6, 1\}$

Es un algoritmo del tipo *divide y vencerás* y su invención se atribuye a John von Neumann en 1945.

Algoritmo Merge Sort

- ▶ Se divide la lista original en n sublistas ordenadas. ✓
- ▶ Luego se unen o concatenan en $n/2$ sublistas ordenadas.
- ▶ Luego en $n/4$ sublistas ordenadas.
- ▶ Se repite hasta que queda una sola lista.

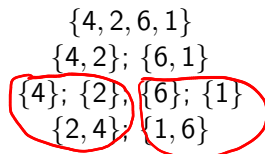
Es un algoritmo del tipo *divide y vencerás* y su invención se atribuye a John von Neumann en 1945.

$\{4, 2, 6, 1\}$
 $\{4, 2\}; \{6, 1\}$
 $\{4\}; \{2\}; \{6\}; \{1\}$

Algoritmo Merge Sort

- ▶ Se divide la lista original en n sublistas ordenadas.
- ▶ Luego se unen o concatenan en $n/2$ sublistas ordenadas.
- ▶ Luego en $n/4$ sublistas ordenadas.
- ▶ Se repite hasta que queda una sola lista.

Es un algoritmo del tipo *divide y vencerás* y su invención se atribuye a John von Neumann en 1945.



Algoritmo Merge Sort

- ▶ Se divide la lista original en n sublistas ordenadas.
- ▶ Luego se unen o concatenan en $n/2$ sublistas ordenadas.
- ▶ Luego en $n/4$ sublistas ordenadas.
- ▶ Se repite hasta que queda una sola lista.

Es un algoritmo del tipo *divide y vencerás* y su invención se atribuye a John von Neumann en 1945.

$\{4, 2, 6, 1\}$
 $\{4, 2\}; \{6, 1\}$
 $\{4\}; \{2\}; \{6\}; \{1\}$
 $\{2, 4\}; \{1, 6\}$
 ~~$\{1, 2, 4, 6\}$~~

Algoritmo Merge Sort

- ▶ Se divide la lista original en n sublistas ordenadas.
- ▶ Luego se unen o concatenan en $n/2$ sublistas ordenadas.
- ▶ Luego en $n/4$ sublistas ordenadas.
- ▶ Se repite hasta que queda una sola lista.

Es un algoritmo del tipo *divide y vencerás* y su invención se atribuye a John von Neumann en 1945.

$\{4, 2, 6, 1\}$
 $\{4, 2\}; \{6, 1\}$
 $\{4\}; \{2\}; \{6\}; \{1\}$
 $\{2, 4\}; \{1, 6\}$
 $\{1, 2, 4, 6\}$

- ▶ Mejor caso: $n \log(n)$ ✓
- ▶ Caso promedio: $n \log(n)$ ✓
- ▶ Peor caso: $n \log(n)$ ✓

Complejidad del algoritmo mergesort

Algoritmo recursivo MergeSort

Entrada: Lista de números enteros de longitud n .

Salida: Lista de números ordenados

1. SI $n < 1$ ENTONCES Regresar
2. SI $n = 2$ ENTONCES
 - 2.1 Ordenar los dos números
 - 2.2 Regresar
3. Llamada recursiva a MergeSort(mitad izquierda de Lista)
4. Llamada recursiva a MergeSort(mitad derecha de Lista)
5. Llamada a Merge(mitad izquierda, mitad derecha)
6. FIN

Complejidad del algoritmo mergesort

Algoritmo recursivo MergeSort

Entrada: Lista de números enteros de longitud n

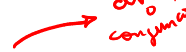
Salida: Lista de números ordenados

1. SI $n < 1$ ENTONCES Regresar
2. SI $n = 2$ ENTONCES
 - 2.1 Ordenar los dos números
 - 2.2 Regresar
3. Llamada recursiva a MergeSort(mitad izquierda de Lista)
4. Llamada recursiva a MergeSort(mitad derecha de Lista)
5. Llamada a Merge(mitad izquierda, mitad derecha)
6. FIN

Algoritmo Merge

Entrada: Listas izquierda L y derecha R , con longitudes n_L y n_R

Salida: Lista de números ordenados W de longitud $n = n_L + n_R$

1. $i = 1, j = 1, k = 1$ 
2. MIENTRAS $(i < n_L) \wedge (j < n_R)$ HACER
 - 2.1 SI $L(i) < R(j)$ ENTONCES
 $W(k) = \underline{L(i)}$, $i = i + 1$;
SINO $W(k) = \underline{R(j)}$, $j = j + 1$
 - 2.2 $k = k + 1$
3. SI $i < n_L$ ENTONCES
 $W(k : n) = L(i : n_L)$
4. FIN

¡Muchas gracias!

Contacto:

Giovanni Ramírez García, PhD

ramirez@ecfm.usac.edu.gt

<http://ecfm.usac.edu.gt/ramirez>