Una introducción a los algoritmos

1 Complejidad de los algoritmos

Usamos el término *algoritmo* para referirnos a un conjunto de reglas bien-definidas o instrucciones para obtener un resultado específico a partir de un conjunto de entrada específico en un número finito de pasos. Por ejemplo, la serie de pasos que seguimos en la división (larga) de números enteros es un algoritmo. El uso e interés en los algoritmos en matemáticas se ha incrementado rápidamente los años pasados, incremento que ha sido estimulado principalmente por el creciente uso de las computadoras. Como veremos, los algoritmos juegan un papel mayor en el estudio de la teoría de las gráficas. Antes de investigar algunos algoritmos fundamentales, empezaremos con un tema central para los algoritmos: la complejidad.

La *complejidad* de un algoritmo mide la cantidad de esfuerzo computacional que se gasta cuando la computadora resuelve un problema usando un algoritmo. Esta medida puede referirse al número de pasos computacionales, el tiempo o el espacio utilizados. No existe una diferencia real, sin embargo, entre los dos primeros; y el número de pasos computacionales será la interpretación de la complejidad que usaremos. La complejidad de un algoritmo es comúnmente una función del tamaño y la presentación de los datos de entrada.

Frecuentemente hay más de un algoritmo para resolver un problema dado. Por ejemplo, un problema computacional común es encontrar la inversa de una matriz (no-singular). Supongamos que el algoritmo A encuentra la inversa de una matriz de $n \times n$ en $.5n^4$ unidades de tiempo. Otro algoritmo B encuentra la inversa de una matriz de $n \times n$ en $2.5n^3$ unidades de tiempo. Así, el algoritmo A encontrará la inversa de una matriz de 4×4 en 128 unidades de tiempo mientras que al algoritmo B le tomará veinticinco por ciento más tiempo. Por otro lado, cuando $n > 5$, el algoritmo B es más rápido que el algoritmo A. De hecho, si usamos el método de la eliminación gaussiana, la inversa de una matriz de $n \times n$ puede obtenerse en a lo más cn^3 unidades de tiempo, donde c es una constante.

Cuando decimos que al algoritmo A *le toma* $.5n^4$ unidades de tiempo encontrar la inversa de una matriz de $n \times n$, queremos decir que *a lo más* $.5n^4$ unidades tiempo. De hecho, relativamente pocas matrices de $n \times n$ necesitarán de $.5n^4$ unidades de tiempo. Esto es, sólo en el peor de los casos es que necesitamos de tanto tiempo. Este tipo de medida de la complejidad se llama *complejidad del peor-caso*.

Sin duda, hay instancias del problema de la inversión de matrices que requieren de muchísimo menos tiempo que la complejidad del peor-caso, como para las matrices diagonales. Y aunque la complejidad del peor-caso es la medida más común de complejidad, existen otras. Por ejemplo, la complejidad del caso-promedio describe el tiempo promedio que correo un algoritmo sobre todas las matrices de $n \times n$.

Un algoritmo es *eficiente* o «rápido» si su complejidad es un polinomio del tamaño de la entrada, digamos n, de los datos o está acotado por un polinomio en n. Por ejemplo, los algoritmos cuyas complejidades son $\sqrt{n}$, $.5n^4$ o $n \log n$ son eficientes, mientras que aquéllos cuyas complejidades son 2^n, $n!$ o n^n no son eficientes[1].

Un problema computacional es *tratable* si existe un algoritmo eficiente para resolver el problema. Un problema computacional es *intratable* si se puede establecer que **no** existe ningún algoritmo eficiente para resolver el problema. Veremos ejemplos de problemas tratables a lo largo del texto. Para muchos problemas, no estamos seguros si son o no tratables.

Un ejemplo de un problema intratable es el llamado problema de la teselación. Dado un polígono, ¿todo el plano puede ser teselado con mosaicos con la forma del polígono?

Berger probó que el problema de la teselación es intratable, es decir, no existe un algoritmo eficiente para determinar si el plano puede ser teselado con un polígono dado. Por supuesto, existe algoritmos eficientes para resolver instancias de este problema. Ciertamente, el problema de la teselación se resuelve fácilmente para rectángulos y hexágonos regulares. También tiene una solución fácil para pentágonos regulares: el plano *no puede ser* teselado con pentágonos regulares.

Para comparar de forma más efectiva las complejidades de los algoritmos, describiremos el «orden» de una función. Sean f y g dos funciones reales definidas en el mismo conjunto de números reales no-negativos. Decimos que f *es de orden a lo más* g si existe una constante real positiva B y un número

[1]Esta noción de eficiencia fue propuesta por Jack Edmonds en `https://cms.math.ca/openaccess/cjm/v17/cjm1965v17.0449-0467.pdf`, un artículo sólo sobre matemáticas (salvo una digresión).

real no-negativo b tales que

$$|f(x)| \leq B|g(x)| \text{ para todo } x > b.$$

Lo denotamos como $f(x) = \mathcal{O}(g(x))$ o decimos que $f(x)$ *es* $\mathcal{O}(g(x))$ (y lo leemos como $f(x)$ es «o grande» de $g(x)$). Esto significa que f **no crece más rápido** que g. Las funciones f y g son del *mismo orden* si $f(x) = \mathcal{O}(g(x))$ y $g(x) = \mathcal{O}(f(x))$.

Supongamos que $f(n) = 3n + 5$ y $g(n) = n^2$ para todo número natural n. Así, $f(n) = \mathcal{O}(g(n))$. Para la constante $C = 1$,

$$3n + 5 \leq 1n^2 \qquad \text{para todo } n > 4 = n_0.$$

Esta desigualdad puede probarse por medio de la inducción matemática, pero usaremos otra aproximación. Notemos que ${}^3\!/_n \leq {}^3\!/_5$ cuando $n \geq 5$ y ${}^5\!/_{n^2} \leq {}^1\!/_5$ cuando $n \geq 5$. Así,

$$\frac{3}{n} + \frac{5}{n^2} \leq \frac{3}{5} + \frac{1}{5} < 1 \qquad \text{para todo } n > 4.$$

Al multiplicar $({}^3\!/_n + {}^5\!/_{n^2}) \leq 1$ por n^2 obtenemos el resultado deseado. También pudimos haber verificado que $f(n) = \mathcal{O}(g(n))$ seleccionando otros valores de C y n_0. Por ejemplo, supongamos que $C = 3$. Así, tenemos que $3n + 5 \leq 3n^2$ para $n > 2 = n_0$. Para mostrar esto, basta con observar que ${}^3\!/_n \leq 1$ cuando $n \geq 3$ y ${}^5\!/_{n^2} < 2$ cuando $n \geq 3$. Por lo tanto,

$$\frac{3}{n} + \frac{5}{n^2} \leq 1 + 2 = 3 \qquad \text{para } n > 2.$$

Al multiplicar por n^2 obtenemos la desigualdad $3n + 5 \leq 3n^2$. Otras elecciones de C y n_0 también funcionan, por ejemplo, $C = 11$ y $n_0 = 0$.

Supongamos que $f(n) = .5n^4$ y $g(n) = 2.5n^3$. Así, $g(n) = \mathcal{O}(f(n))$; de hecho, $g(n)$ es $\mathcal{O}(n^3)$. También, $f(n)$ es $\mathcal{O}(n^4)$; sin embargo, $f(n) \neq \mathcal{O}(g(n))$. Si $h(n)$ es cualquier polinomio en n de grado k entonces $h(n) = \mathcal{O}(n^k)$. Así, un algorimto para resolver un problema con datos de entrada de tamaño n y función de complejidad $f(n)$ es eficiente si $f(n)$ es $\mathcal{O}(n^k)$ para algún entero positivo fijo k. Sui los algoritmos A y B tienen funciones de complejidad $f(n)$ y $g(n)$, respectivamente, decimos que el algoritmo A es *más eficiente* que el algoritmo B si $f(n) = \mathcal{O}(g(n))$ pero $g(n) \neq \mathcal{O}(f(n))$.

Para obtener una idea de la jerarquía de los órdenes crecientes de las funciones, hemos listado varios órdenes comunes. Escribimos $\log n$ para $\log_2 n$, aunque la base del logarítmo es irrelevante ya que mientras a, $b > 1$, $\log_a n$ es un múltiplo constante de $\log_b n$.

Jerarquía de los órdenes crecientes:

$$\mathcal{O}(1),\ \mathcal{O}(\log n),\ \mathcal{O}(n),\ \mathcal{O}(n\log n),\ \mathcal{O}(n^2),\ \mathcal{O}(n^3),\ \mathcal{O}(2^n),\ \mathcal{O}(n!)$$

Ya que $\log n < n$ para todo $n \geq 1$, $\log n$ es $\mathcal{O}(n)$. También, 2^n es $\mathcal{O}(n!)$ ya que $2^n < n!$ para $n \geq 4$. Sin embargo, 2^n no es de orden polinomial ya que para todo entero positivo k y toda constante positiva real C, la desigualdad $2^n > Cn^k$ se satisface para n suficientemente grandes.

Supongamos que dos algoritmos A y B se usan para resolver el mismo problema y A tiene complejidad de tiempo $\mathcal{O}(f(n))$ mientras B tiene complejidad de tiempo $\mathcal{O}(g(n))$, donde $g(n) = \mathcal{O}(f(n))$. Así, para valores grandes de n, el algoritmo B requiere, salvo una constante, no más tiempo para resolver el problema que el algoritmo A.

En la siguiente tabla, el tiempo de ejecución de varias funciones de complejidad comunes se muestran para varios valores de n, donde la unidad de tiempo es un segundo. Si el tiempo de ejecución en la tabla no está dado en segundos, el tiempo es aproximado.

Valor de n

Complejidad	**2**	**8**	**32**	**64**
$\log n$	1 s	3 s	5 s	6 s
n	2 s	8 s	32 s	1.07 min
$n\log n$	2 s	24 s	2.67 min	6.4 min
n^2	4 s	1.07 min	17.07 min	1.14 horas
n^3	8 s	8.53 min	9.1 horas	3.03 días
2^n	4 s	4.27 min	1.36 siglos	5.86×10^9 siglos
$n!$	2 s	4.2 h	8.34×10^{25} siglos	4.02×10^{79} siglos

Concluiremos esta parte con un algoritmo bastante simple (para encontrar la primera letra, alfabéticamente, en una lista de n palabras) y una discusión sobre su complejidad. En un algoritmo, la notación $A \leftarrow B$ significa que el valor (actual) de A se reemplaza por el valor de B. Si A y B son palabras, $A < B$ indicará que A precede a B alfabéticamente.

Evaluaremos la complejidad del algoritmo 1 determinando el número de comparaciones que se requieren. Ya que este número es $n-1$, el algoritmo 1 tiene complejidad $n-1$ o $\mathcal{O}(n)$.

Algoritmo 1 Algoritmo para determinar la primera letra, alfabéticamente, en una lista de palabras

Entrada: Una lista de palabras $W(1)$, $W(2)$, ..., $W(n)$.
Salida: La primera palabra (alfabéticamente) y su posición en la lista.

```
1: function PRIMERA_PALABRA(lista)
2:     PRIMERA ← W(1)   ▷ PRIMERA representa la primera palabra hasta el momento
3:     p ← 1            ▷ p es la posición de la palabra deseada
4:     for k ← 2 to n
5:         if W(k) < PRIMERA then
6:             PRIMERA ← W(k)
7:             p ← k
8:     next k
9:     return PRIMERA, p
```

2 Algoritmos de búsqueda

En esta sección describiremos un problema fundamental así como dos algoritmos que resuelven el problema. También compararemos la eficiencia de estos algoritmos.

Dada una lista alfabética de palabras, deseamos determinar si y en dónde una palabra específica aparece en la lista. Por ejemplo, nos gustaría saber cuándo el nombre de una persona aparece en una lista alfabética de pasajeros para un tomar un cierto vuelo. ¿Cómo podríamos (o, preferentemente, una computadora) buscar en la lista para determinar si contiene el nombre en cuestión? (Más adelante veremos cómo ordenar alfabéticamente una lista de palabras). Una forma simple consiste en empezar desde el principio de la lista y buscar, secuencialmente, hasta que hallemos el nombre o hasta que terminemos con la lista.

A este algoritmo frecuentemente se le llama **algoritmo de búsqueda secuencial**. Denotemos por $W(k)$ la palabra en la k-ésima posición en la lista. Si hay n palabras en la lista, entonces las palabras, en orden alfabetico, serán $W(1)$, $W(2)$, ..., $W(n)$. Si la palabra deseada es $CLAVE$, la pregunta en cuestión es si $W(k) = CLAVE$ para algún k con $k \in \{1, 2, \ldots, n\}$. Véase el algoritmo 2.

Ciertamente, al algoritmo le tomará a la más n comparaciones (para comparar cada palabra con $CLAVE$) antes que el algoritmo termine. Por lo tanto, si complejidad en términos del número de comparaciones (en vez de en unidades de tiempo) entonces la complejidad (del peor-caso) del algoritmo

Algoritmo 2 Algoritmo de búsqueda secuencial

Entrada: Una palabra dada $CLAVE$ y una lista de palabras $W(1)$, $W(2)$, $\ldots$, $W(n)$.

Salida: Determina si $CLAVE$ aparece en la lista y, en tal caso, regresa su posición en la lista.

```
1: function BÚSQUEDA_SECUENCIAL(CLAVE, (W(1), W(2), ..., W(n)))
2:     for k ← 1 to n
3:         if W(k) = CLAVE then
4:             return k y el algoritmo se detiene
5:     next k
6:     return «No aparece en la lista.»
```

es n o $\mathcal{O}(n)$.

Observemos que el algoritmo de búsqueda secuencial no utiliza el hecho de que la lista dada está ordenada alfabéticamente. Ahora describiremos una técnica de *divide y vencerás* que busca una palabra específica en una lista ordenada alfabéticamente. El término **divide-y-vencerás** significa que el dividiendo el problema en subproblemas más pequeños puede ser resuelto más fácilmente. El *algoritmo de búsqueda binaria* logra la misma tarea que el algoritmo 2 pero es más eficiente; es decir, en el peor caso requiere de significamente menos comparaciones para n grande.

El algoritmo de búsqueda secuencial empieza con la primera palabra de la lista. En contraste, el algoritmo de búsqueda binaria empieza a la mitad de la lista. El algoritmo de búsqueda binaria compara la palabra «de en medio» para determina si es $CLAVE$. Si la palabra no es $CLAVE$ y $CLAVE$ precede a esta palabra de en medio alfabéticamente, entonces $CLAVE$ se compara con la palabra que está entre la primera palabra y la palabra de enmedio de la lista; en otro caso, comparamos $CLAVE$ con la palabra que está a la mitad entre la palabra de en medio y la última palabra de la lista. Si la lista tiene n palabras, entonces por la «palabra de en medio» entenderemos la palabra que aparece en la posición $\lfloor {}^{n+1}/_{2} \rfloor$.

Supongamos que $n = 25$. Así, $W(13)$ es la palabra de en medio de la lista y compararemos $W(13)$ y $CLAVE$. Si $W(13) = CLAVE$ entonces $CLAVE$ sí está en la lista y es la décimo tercera palabra en la lista. Si $W(13) \neq CLAVE$ entonces checamos si $CLAVE$ está antes de $W(13)$ alfabéticamente. Esto lo denotaremos por $CLAVE < W(13)$. Si, de hecho, $K > W(13)$ entonces compararemos $CLAVE$ y $W(19)$.

Por ejemplo, supongamos que deseamos determinar si la palabra FLE-

CHA aparece en la siguiente lista:

$$\begin{aligned} W(1) &= BALÓN \\ W(2) &= CARRO \\ W(3) &= ESCALERA \\ W(4) &= FLECHA \\ W(5) &= MANO \\ W(6) &= PIE \\ W(7) &= PUERTA \\ W(8) &= RED \\ W(9) &= SARTÉN \\ W(10) &= TIENDA \end{aligned}$$

Usando el algoritmo de búsqueda binaria, primero checamos si $W(5) = MANO$ es la palabra dada $FLECHA$. Si no, ya que $FLECHA < MANO$, ahora consideramos la palabra de en medio en la sublista $W(1)$, $W(2)$, $W(3)$ y $W(4)$, que es $W(2) = CARRO$. Ya que $CARRO \neq FLECHA$, pero $FLECHA > CARRO$, ahora comparamos $FLECHA$ con la palabra de en-medio de la sublista $W(3)$ y $W(4)$, a saber, $W(3) = ESCALERA$. Ya que $ESCALERA \neq FLECHA$, pero $FLECHA > ESCALERA$, sólo resta una palabra, a saber, $W(4)$ y hemos hallado la palabra.

El algoritmo 3 muestra el algoritmo de búsqueda binaria.

Ahora determinaremos la complejidad (del peor-caso) del algoritmo de búsqueda binaria. Observemos que los pasos 2, 3 y 11 se ejecutan sólo una vez por lo que involucran un número constante de operaciones. Así, su complejidad es $\mathcal{O}(1)$. Sea $B(n)$ el número máximo de veces que ejecutamos el paso 4 (junto con sus subpasos 5–10). Ciertamente, el número máximo se dará cuando la palabra clave no aparezca en la lista.

Despues de ejecutar el paso 4 por vez primera, tendremos una lista de a lo más $\lfloor \frac{n}{2} \rfloor$ palabras. Así,

$$B(n) \leq 1 + B\left(\left\lfloor \frac{n}{2} \right\rfloor\right). \tag{1}$$

Usamos inducción para probar que

$$B(n) \leq 1 + \log_2 n = 1 + \log n \tag{2}$$

para todo entero positivo n. Ya que el paso 4 (junto con sus subpasos 5–10) se ejecutan sólo una vez cuando $n = 1$, tenemos la desigualdad 2.

Algoritmo 3 Algoritmo de búsqueda binaria

Entrada: Una palabra dada $CLAVE$ y una lista de palabras $W(1)$, $W(2)$, $\ldots$, $W(n)$ ordenadas alfabéticamente.

Salida: Determina si $CLAVE$ aparece en la lista y, en tal caso, regresa su posición en la lista.

1: **function** BÚSQUEDA_BINARIA(CLAVE, ($W(1)$, $W(2)$, $\ldots$, $W(n)$))
2: $f \leftarrow 1$ ▷ f representa la primera posición de las palabras en la sublista considerada
3: $l \leftarrow n$ ▷ l representa la última posición de la palabras palabras en la sublista considerada
4: **while** $f < l$ **do**
5: $k \leftarrow \lfloor \frac{f+l}{2} \rfloor$ ▷ $W(k)$ es la palabra de en medio entre $W(f)$ y $W(l)$
6: **if** $CLAVE = W(K)$ **then**
7: **return** k y termina.
8: **if** $CLAVE < W(K)$ **then** ▷ Si $CLAVE$ precede a la palabra de en medio de la sublista actual entonces la nueva lista consiste en las palabras entre $W(f)$ y $W(k-1)$.
9: $l \leftarrow k - 1$.
10: **if** $CLAVE > W(K)$ **then** ▷ Si $CLAVE$ sucede a la palabra de en medio de la lista actual entonces la nueva sublista consiste de las palabras entre $W(k+1)$ y $W(l)$.
11: $f \leftarrow k + 1$.
12: **return** «No aparece en la lista.»

Supongamos que la desigualdad 2 se satisface para todo entero n con $1 \leq n \leq k$, para algún entero positivo k. Mostraremos que

$$B(k+1) \leq 1 + \log(k+1).$$

La desigualdad 1 implica que

$$B(k+1) \leq 1 + B\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right).$$

Sin embargo,

$$B\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right) \leq 1 + \log\left(\left\lfloor \frac{k+1}{2} \right\rfloor\right) \leq 1 + \log\left(\frac{k+1}{2}\right),$$

donde la primera desigualdad se sigue de la hipótesis de inducción. Por lo tanto,

$$B(k+1) \leq 2 + \log(k+1) - \log 2 = 1 + \log(k+1),$$

lo que da el resultado esperado. Así, tenemos que $B(n) = \mathcal{O}(\log n)$. Ya que cada vez que ejecutamos el paso 4, necesitamos de a lo más dos comparaciones y dos asignaciones, la complejidad resultante del paso 4 es del orden de $B(n)$, es decir, $\mathcal{O}(\log n)$.

3 Algoritmos de ordenación

En los algoritmos de búsqueda que previamente escribimos, supusimos que la lista dada estaba en orden alfabético. Ahora describiremos dos algoritmos para ordenar una lista dada de palabras en orden alfabético. El primero de estos, llamado de *búsqueda secuencial* o *algoritmo de ordenación por selección*, usa el algoritmo 1 para localizar la primera palabra, alfabéticamente, en la lista. Véase el algoritmo 4.

Ilustraremos el algoritmo de ordenación con la lista: $W(1) = CERCA$, $W(2) = PUERTA$, $W(3) = CARRO$ y $W(4) = LADO$. Ya que $n = 4$, haremos por los pasos 2, 3 y 4 tres veces, que corresponden a $k = 1$, $k = 2$ y $k = 3$.

Lista original (n=4)	**Paso 2 (k=1)**	**Paso 2 (k=2)**	**Paso 2 (k=3)**
CERCA	CARRO	CARRO	CARRO
PUERTA	PUERTA	CERCA	CERCA
CARRO	CERCA	PUERTA	LADO
LADO	LADO	LADO	PUERTA

Algoritmo 4 Algoritmo de ordenación secuencial

Entrada: Una lista de n palabras $(W(1), W(2), \ldots, W(n))$.

Salida: Regresa la lista ordenada alfabéticamente.

1: **function** ORDENACIÓN_SECUENCIAL$((W(1), W(2), \ldots, W(n)))$
2: **for** $k \leftarrow 1$ **to** $n-1$
3: $(PRIMERA, p) \leftarrow primera\_palabra(W(k), W(k+1), \ldots, W(n))$
4: $W(p) \leftarrow W(k)$ ▷ Intercambiaremos la primera palabra, alfabéticamente, en la sublista por la primera palabra en la sublista
5: $W(k) \leftarrow PRIMERA$
6: **next**
7: **return** $W(1), W(2), \ldots, W(n)$.

Para determinar la complejidad del algoritmo 4, primero recordemos que el algoritmo 1 requiere de a lo más $n-1$ comparaciones. Cuando aplicamos el algoritmo 1 por segunda vez, la sublista tiene $n-1$ palabras por lo que requeriremos de a lo más $n-2$ comparaciones. De esta manera, necesitamos de a lo más

$$(n-1) + (n-2) + \cdots + 3 + 2 + 1 = \binom{n}{2} = \frac{n(n-1)}{2}$$

comparaciones. Por lo tanto, el algoritmo 4 tiene complejidad $\mathcal{O}(n^2)$.

Otro algoritmo común para ordenar alfabéticamente una lista de palabras empieza al final de la lista y compara cada par de palabras consecutivas en cada ejecución sobre la lista. Si dos de las palabras no están en orden alfabético entonces intercambiamos las palabras.

Lista orig.	**Primera corrida**			**Segunda**		**Tercera**
CERCA	CERCA	CERCA	CARRO	CARRO	CARRO	CARRO
PUERTA	PUERTA	CARRO	CERCA	CERCA	CERCA	CERCA
CARRO	CARRO	PUERTA	PUERTA	LADO	LADO	LADO
LADO	LADO	LADO	LADO	PUERTA	PUERTA	PUERTA

Este algoritmo se llama de *ordenación de burbuja* ya que las primeras palabras, en orden alfabético, «suben» como burbujas hasta arriba de la lista. También se le llama algoritmo de *ordenación por intercambio*. Véase el algoritmo 5.

Cuando usamos el algoritmo de burbuja, la primera pasada sobre una lista de n palabras requiere de $n-1$ comparaciones. Una vez que hemos puesto hasta arriba la primera palabra en orden alfabético, la siguiente pasada va sobre las últimas $n-1$ palabras y requiere de $n-2$ comparaciones.

Algoritmo 5 Algoritmo de ordenación de burbuja

Entrada: Una lista de n palabras $W(1)$, $W(2)$, ..., $W(n)$.

Salida: Regresa la lista ordenada alfabéticamente.

```
1: function ORDENACIÓN_DE_BURBUJA((W(1), W(2), ..., W(n)))
2:     for j ← 1 to n − 1                                  ▷ Una vez que ordenamos alfabéticamente las primeras j − 1 palabras, sólo nos quedan por ordenas las palabras restantes W(j), W(j + 1), ..., W(n)
3:         for i ← 1 to n − j                              ▷ Checamos las palabras consecutivas en la lista W(j), W(j + 1), ..., W(n) desde abajo hasta arriba y hacemos los intercambios necesarios.
4:             if W(n + 1 − i) < W(n − i) then
5:                 TEMP ← W(n − i)
6:                 W(n − i) ← W(n + 1 − i)
7:                 W(n + 1 − i) ← TEMP.
8:         next i
9:     next j
10:    return (W(1), W(2), ..., W(n)).
```

Al continuar de esta forma, llegamos a que el número total de comparaciones es de a lo más

$$(n-1)+(n-2)+\cdots+3+2+1=\frac{n(n-1)}{2},$$

por lo que su complejidad es $\mathcal{O}(n^2)$, la misma complejidad que la del algoritmo de ordenación secuencial.

Otro algoritmo de ordenación utilizar una técnica de divide-y-vencerás y es conocido como *ordenación por mezcla*. Si $n \geq 2$, este algoritmo primero divide una lista dada $L = (W(1), W(2), \ldots, W(n))$ de n palabras en dos sublistas L_1 y L_2 de aproximadamente la mitad de la longitud de L. En concreto, $L_1 = (W(1), W(2), \ldots, W(\lfloor \frac{n}{2} \rfloor))$ y $L_2 = (W(\lfloor \frac{n}{2} \rfloor + 1), W(\lfloor \frac{n}{2} \rfloor + 2), \ldots, W(n))$. El algoritmo ordena L_1 y L_2 alfabéticamente llamándose a sí mismo. Finalmente, las dos lista alfabéticamente ordenadas se «mezclan». Los primeros elementos de cada una de estas listas se comparan, la primera palabra en orden alfabético de estas dos listas será la primera palabra en L y la borramos de la lista en la que aparece. Repetimos esta operación con el resto de las sublistas para determinar la palabra que sigue en orden alfabético en L. El algoritmo se detiene cuando las dos sublistas quedan vacías. Notemos que a los necesitamos $n - 1$ comparaciones para mezclar ambas sublistas L_1 y L_2. Como ejercicio, probarán que el algoritmo de

ordenación por mezcla requiere a lo más $\mathcal{O}(n \log n)$ comparaciones.

Existen otros algoritmos de ordenación como la ordenación rápida y la ordenación por montíuclos. El nombre de «rápida» es un equívoco ya que su complejidad es $\mathcal{O}(n^2)$, la misma que la búsqueda secuencial y la ordenación por burbuja. Por otro lado, la ordenación por montículo tiene complejidad $\mathcal{O}(n \log n)$. Por supuesto, todos estos algoritmos de ordenación son eficientes por definición.