*55. (Se requiere Cálculo). Supongamos que $f(x)$ es $o(g(x))$. ¿Se cumple que $2^{f(x)}$ es $o(2^{g(x)})$?

*56. (Se requiere Cálculo). Supongamos que $f(x)$ es $o(g(x))$. ¿Se cumple que $\log |f(x)|$ es $o(\log |g(x)|)$?

57. (Se requiere Cálculo). Las dos partes de este problema describen la relación entre las funciones O y o.
   a) Muestra que si $f(x)$ y $g(x)$ son funciones tales que $f(x)$ es $o(g(x))$, entonces $f(x)$ es $O(g(x))$.
   b) Muestra que si $f(x)$ y $g(x)$ son funciones tales que $f(x)$ es $O(g(x))$, no tiene por qué cumplirse necesariamente que $f(x)$ es $o(g(x))$.

58. (Se requiere Cálculo). Muestra que si $f(x)$ es un polinomio de grado n y $g(x)$ es un polinomio de grado m, donde $m > n$, entonces $f(x)$ es $o(g(x))$.

59. (Se requiere Cálculo). Muestra que si $f_1(x)$ es $O(g(x))$ y $f_2(x)$ es $o(g(x))$, entonces $f_1(x) + f_2(x)$ es $O(g(x))$.

60. (Se requiere Cálculo). Sea H_n el n-ésimo **número armónico**

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}.$$

Demuestra que H_n es $O(\log n)$. (*Indicación:* Establece primero la desigualdad

$$\sum_{j=2}^{n} \frac{1}{j} < \int_1^n \frac{1}{x}\,dx$$

demostrando que la suma de las áreas de los rectángulos de altura $1/j$ y base desde $x = j - 1$ a $x = j$, para $j = 2, 3, \ldots, n$, es menor que el área bajo la curva $y = 1/x$ desde 2 hasta n).

*61. Demuestra que $n \log n$ es $O(\log n!)$.

☞ 62. Determina si $\log(n!)$ es o no $\Theta(n \log n)$. Justifica tu respuesta.

Sean $f(x)$ y $g(x)$ funciones del conjunto de los números reales en el conjunto de los números reales. Decimos que las funciones f y g son **asintóticamente equivalentes** y escribimos $f(x) \sim g(x)$ si $\lim_{x \to \infty} f(x)/g(x) = 1$.

63. (Se requiere Cálculo). Para cada par de estas funciones, determina si f y g son asintóticamente equivalentes.
   a) $f(x) = x^2 + 3x + 7$, $g(x) = x^2 + 10$
   b) $f(x) = x^2 \log x$, $g(x) = x^3$
   c) $f(x) = x^4 + \log(3x^8 + 7)$, $g(x) = (x^2 + 17x + 3)^2$
   d) $f(x) = (x^3 + x^2 + x + 1)^4$, $g(x) = (x^4 + x^3 + x^2 + x + 1)^3$
   e) $f(x) = \log(x^2 + 1)$, $g(x) = \log x$
   f) $f(x) = 2^{x+3}$, $g(x) = 2^{x+7}$
   g) $f(x) = 2^{2^x}$, $g(x) = 2^{x^2}$

2.3 Complejidad de algoritmos

INTRODUCCIÓN

¿Cuándo proporciona un algoritmo una solución satisfactoria a un problema? Primero, debe producir siempre la respuesta correcta. En el Capítulo 3 se estudiará cómo se puede demostrar esto. En segundo lugar, debería ser eficiente. En esta sección estudiaremos la eficiencia de los algoritmos.

¿Cómo se puede analizar la eficiencia de los algoritmos? Una medida de eficiencia es el tiempo que requiere un ordenador para resolver un problema utilizando un algoritmo para valores de entrada de un tamaño especificado. Una segunda medida es la cantidad de memoria que se necesita de nuevo para valores de entrada de un tamaño dado.

Preguntas como las anteriores están ligadas a la **complejidad computacional** del algoritmo. Un análisis del tiempo requerido para resolver un problema de un tamaño particular está relacionado con la **complejidad en tiempo** del algoritmo. Análogamente, un análisis de la memoria de ordenador requerida involucra la **complejidad en espacio** del algoritmo. Las consideraciones de complejidad en tiempo y en espacio de un algoritmo son esenciales en la implementación de algoritmos. Obviamente, es importante saber si un algoritmo producirá su respuesta en un milisegundo, en un minuto o en un millón de años. De forma similar, debemos tener suficiente memoria disponible para poder resolver el problema, por lo que la complejidad en espacio debe tenerse también en cuenta.

Las consideraciones sobre complejidad en espacio están ligadas a las estructuras de datos usadas en la implementación del algoritmo. Como las estructuras de datos no se tratan con detalle en este libro, no consideraremos la complejidad en espacio. Restringiremos nuestra atención

a la complejidad en tiempo, de manera que, a menos que se diga lo contrario, cuando hablemos de complejidad nos estaremos refiriendo a la complejidad en tiempo.

COMPLEJIDAD EN TIEMPO

La complejidad en tiempo de un algoritmo se puede expresar en términos del número de operaciones que realiza el algoritmo cuando los datos de entrada tienen un tamaño particular. Las operaciones utilizadas para medir la complejidad pueden ser la comparación de enteros, la suma, multiplicación o división de enteros, así como cualquier otra operación básica.

La complejidad se describe en términos del número de operaciones requeridas, en lugar del tiempo de cálculo real, debido a que distintos ordenadores necesitan tiempos diferentes para realizar las mismas operaciones básicas. Los ordenadores más rápidos pueden realizar operaciones básicas con bits (por ejemplo, sumar, multiplicar, comparar o intercambiar las posiciones de dos bits) en 10^{-9} segundos (1 nanosegundo), mientras que un ordenador personal puede requerir 10^{-6} segundos (un microsegundo), mil veces más, para realizar la misma operación. Además, es bastante complicado descomponer todas las operaciones que desarrolla un algoritmo en las operaciones elementales que realiza un ordenador.

Ilustramos a continuación la complejidad del Algoritmo 1 de la Sección 2.1, que busca el elemento máximo de un conjunto.

EJEMPLO 1 Describe la complejidad del Algoritmo 1 de la Sección 2.1 para encontrar el elemento máximo de un conjunto.

Ejemplos adicionales

Solución: Se usará como medida de la complejidad en tiempo del algoritmo el número de comparaciones, pues la comparación es la operación básica utilizada.

Para encontrar el máximo elemento de un conjunto con n elementos, listados en orden arbitrario, primero se fija el máximo preliminar como el elemento inicial de la lista. Luego, tras realizar una comparación para determinar si se ha llegado al final de la lista, se comparan el máximo preliminar y el segundo elemento de la lista, actualizando el valor del máximo si el segundo término es mayor que el primero. Este procedimiento se repite utilizando dos comparaciones adicionales por cada término: una para determinar si se ha llegado al fin de la lista y otra para determinar si procede actualizar el valor provisional del máximo. Como se hacen dos comparaciones desde el segundo hasta el n-ésimo elemento, y una más para salir del bucle cuando $i = n + 1$, se realizan exactamente $2(n - 1) + 1 = 2n - 1$ comparaciones al aplicar el algoritmo. Por tanto, el algoritmo para encontrar el máximo de un conjunto de n elementos tiene una complejidad $\Theta(n)$, medido en términos del número de comparaciones realizadas. ◀

Ahora analizaremos la complejidad en tiempo de los algoritmos de búsqueda.

EJEMPLO 2 Calcula la complejidad en tiempo del algoritmo de búsqueda lineal.

Solución: Para medir la complejidad en tiempo utilizaremos de nuevo el número de comparaciones realizadas por el algoritmo. En cada paso del bucle del algoritmo se llevan a cabo dos comparaciones: una para ver si se ha alcanzado el final de la lista y la segunda para comparar el elemento x con un término de la lista. Finalmente, fuera del bucle se hace una comparación más. Por tanto, si $x = a_i$, se hacen $2i + 1$ comparaciones. El mayor número de comparaciones se alcanza cuando el elemento no está en la lista, a saber, $2n + 2$. En este caso, se hacen $2n$ comparaciones para determinar que $x \neq a_i$, para $i = 1, 2, \ldots, n$, una comparación adicional para salir del bucle y una más fuera del bucle. Por tanto, cuando x no figura en la lista, se hace un total de $2n + 2$ comparaciones. Así, una búsqueda lineal requiere a lo más $O(n)$ comparaciones. ◀

COMPLEJIDAD DEL PEOR CASO El tipo de análisis de complejidad hecho en el Ejemplo 2 es un análisis del **peor caso.** Por comportamiento de un algoritmo en el peor caso entendemos el mayor número de operaciones que hace falta para resolver el problema dado utilizando el algo-

ritmo para unos datos de entrada de un determinado tamaño. Los análisis del peor caso nos dicen cuántas operaciones tienen que realizar los algoritmos para garantizar que producirán una solución.

EJEMPLO 3 Calcula la complejidad del algoritmo de búsqueda binaria.

Solución: Por simplicidad, suponemos que hay $n = 2^k$ elementos en la lista $a_1, a_2, \ldots, a_n$, donde k es un entero no negativo. Observa que $k = \log n$. (Si n, el número de elementos de la lista, no es potencia de 2, la lista se puede considerar parte de otra más larga con 2^{k+1} elementos, donde $2^k < n < 2^{k+1}$. Aquí, 2^{k+1} es la menor potencia de 2 mayor que n).

En cada paso del algoritmo, i y j, las posiciones del primero y último término de la lista restringida considerada en este paso se comparan para ver si la lista restringida tiene más de un término. Si $i < j$, se realiza una comparación para determinar si x es mayor que el término central de la lista restringida.

En el primer paso, la búsqueda se restringe a una lista con 2^{k-1} términos. Hasta aquí se han hecho dos comparaciones. Este procedimiento continúa realizando dos comparaciones en cada paso para restringir la búsqueda a una lista con la mitad de términos. En otras palabras, se hacen dos comparaciones en el primer paso del algoritmo cuando la lista tiene 2^k elementos, dos más cuando la búsqueda se ha reducido a una lista con 2^{k-1} elementos, dos más cuando la búsqueda se reduce a 2^{k-2} elementos y así sucesivamente. Las dos últimas comparaciones se hacen cuando la búsqueda se ha reducido a una lista con 2^1 elementos. Finalmente, cuando sólo queda un término en la lista, una comparación basta para saber que no hay términos adicionales, y se necesita una más para determinar si este término es x.

Por tanto, a lo más, se requiere $2k + 2 = 2 \log n + 2$ comparaciones para realizar una búsqueda binaria cuando la lista de entrada tiene 2^k elementos. (Si n no es potencia de 2, la lista original se amplía a una con 2^{k+1} términos, donde $k = \lfloor \log n \rfloor$, y la búsqueda requiere a lo más $2\lceil \log n \rceil + 2$ comparaciones). Consecuentemente, una búsqueda binaria requiere como máximo $\Theta(\log n)$ comparaciones. De este análisis se concluye que el algoritmo de búsqueda binaria es más eficiente, en el peor caso, que la búsqueda lineal. ◀

COMPLEJIDAD DEL CASO PROMEDIO Otro tipo importante de análisis de complejidad, además del peor caso, es el denominado análisis del **caso promedio.** En este tipo de análisis se busca el número promedio de operaciones realizadas para solucionar un problema considerando todas las posibles entradas de un tamaño determinado. El análisis de la complejidad del caso promedio es, generalmente, mucho más complicado que el análisis del peor caso. No obstante, podemos hacer este análisis para el algoritmo de búsqueda lineal sin gran dificultad, como se muestra en el Ejemplo 4.

EJEMPLO 4 Describe el comportamiento en el caso promedio del algoritmo de la búsqueda lineal suponiendo que el elemento x está en la lista.

Solución: Hay n tipos de posibles entradas cuando sabemos que x está en la lista. Si x es el primer elemento de la lista, se necesitan tres comparaciones, una para determinar si se ha alcanzado el final de la lista, una para comparar x con el primer término y una fuera del bucle. Si x es el segundo término de la lista, se necesitan dos comparaciones más, de tal forma que se hacen un total de cinco comparaciones. En general, si x es el término i-ésimo de la lista, se realizarán dos comparaciones en cada uno de los i pasos del bucle, y uno fuera del bucle de tal forma que se necesita un total de $2i + 1$ comparaciones. Por tanto, el número promedio de comparaciones realizadas es igual a

$$\frac{3+5+7+\ldots+(2n+1)}{n} = \frac{2(1+2+3+\ldots+n)+n}{n}.$$

En la Sección 3.3 demostraremos que

$$1+2+3+\ldots+n = \frac{n(n+1)}{2}.$$

Por tanto, el número promedio de comparaciones realizadas por el algoritmo de búsqueda lineal (cuando se sabe que x está en la lista) es

$$\frac{2[n(n+1)/2]}{n}+1=n+2,$$

que es $\Theta(n)$. ◀

Observación: En el análisis del Capítulo 4 se asume que x está en la lista en la que se busca y puede estar en cualquier posición. También se puede hacer un análisis del caso promedio de este algoritmo cuando es posible que x no esté en la lista (véase el Problema 13 al final de esta sección).

Observación: Generalmente, las comparaciones que se necesitan para determinar si hemos llegado al final de un bucle no se tienen en cuenta. Aunque en los ejemplos anteriores las hemos considerado, a partir de ahora las ignoraremos.

COMPLEJIDAD DEL PEOR CASO DE DOS ALGORITMOS DE ORDENACIÓN Analizamos la complejidad de la ordenación por el método de la burbuja y la ordenación por inserción de los Ejemplos 5 y 6.

EJEMPLO 5 ¿Cuál es la complejidad en el peor caso de la ordenación por el método de la burbuja en términos del número de comparaciones realizadas?

Solución: La ordenación por el método de la burbuja (descrita en el Ejemplo 4 de la Sección 2.1) ordena una lista desarrollando una secuencia de pasadas sobre la lista. Durante cada pasada, este método de ordenación compara sucesivamente elementos adyacentes, intercambiándolos cuando es necesario. Cuando comienza la pasada i-ésima, el método garantiza que los $i-1$ elementos mayores están en sus posiciones correctas. Durante esta pasada se hacen $n-i$ comparaciones. Por tanto, el número total de comparaciones realizadas por el método de la burbuja para ordenar una lista de n elementos es

$$(n-1)+(n-2)+\ldots+2+1=\frac{(n-1)n}{2}$$

empleando una fórmula para la suma que se demostrará en la Sección 3.3. Observa que la ordenación por el método de la burbuja hace siempre este número de comparaciones, porque continúa incluso si la lista está completamente ordenada en algunos pasos intermedios. Consecuentemente, este método realiza $(n-1)n/2$ comparaciones, por lo que tiene complejidad en el peor caso $\Theta(n^2)$ en términos del número de comparaciones realizadas. ◀

EJEMPLO 6 ¿Cuál es la complejidad en el peor caso de la ordenación por inserción en términos del número de comparaciones realizadas?

Solución: La ordenación por inserción (descrita en el Ejemplo 5 de la Sección 2.1) coloca el elemento j-ésimo en la posición correcta entre los primeros $j-1$ términos que ya hemos puesto en el orden correcto. Esto se hace usando una técnica de búsqueda lineal, comparando el elemento j-ésimo con términos sucesivos hasta que se encuentre un elemento que sea mayor o igual que él, o se compara a_j consigo mismo y se para porque a_j no es menor que él mismo. Consecuentemente, en el peor caso, se requiere j comparaciones para insertar el elemento j-ésimo en la posición correcta. Así, el número total de comparaciones realizadas para ordenar por inserción una lista de n elementos es

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

empleando la fórmula de la suma de enteros consecutivos que se demostrará en la Sección 3.3 y teniendo en cuenta que el primer término, 1, no se considera en esta suma. Observa que el número

de comparaciones realizadas en una ordenación por inserción depende considerablemente de la disposición de los elementos menores en la lista. Concluimos que la ordenación por inserción tiene una complejidad en el peor caso $\Theta(n^2)$. ◀

EL SIGNIFICADO DE LA COMPLEJIDAD DE UN ALGORITMO

La Tabla 1 muestra la terminología comúnmente utilizada para describir la complejidad de los algoritmos. Por ejemplo, un algoritmo que encuentra el mayor de los cien primeros términos de una lista de n elementos ($n \geq 100$) aplicando el Algoritmo 1 tiene una **complejidad constante,** puesto que utiliza 99 comparaciones, no importa el valor de n (como podrá verificar el lector). El algoritmo de búsqueda lineal tiene **complejidad lineal** (en el peor caso o en el caso promedio) y el algoritmo de búsqueda binaria tiene **complejidad logarítmica** (en el peor caso). Muchos algoritmos importantes tienen complejidad $n \log n$, como la ordenación por mezcla, que presentaremos en el Capítulo 3.

Tabla 1. Terminología comúnmente utilizada para la complejidad de algoritmos.

Complejidad	*Terminología*
$O(1)$	Complejidad constante
$O(\log n)$	Complejidad logarítmica
$O(n)$	Complejidad lineal
$O(n \log n)$	Complejidad $n \log n$
$O(n^b)$	Complejidad polinómica
$O(b^n)$, donde $b > 1$	Complejidad exponencial
$O(n!)$	Complejidad factorial

Un algoritmo tiene **complejidad polinómica** si tiene complejidad $O(n^b)$, donde b es un racional mayor o igual que 1. Por ejemplo, el algoritmo de ordenación por el método de la burbuja es un algoritmo de complejidad polinómica porque usa $O(n^2)$ comparaciones en el peor caso. Un algoritmo tiene **complejidad exponencial** si su complejidad es $O(b^n)$, donde $b > 1$. El algoritmo que determina si una fórmula proposicional con n variables puede ser satisfecha mediante la comprobación de todas las posibles asignaciones de valores de verdad es un algoritmo con complejidad exponencial porque utiliza $O(2^n)$ operaciones. Finalmente, un algoritmo tiene **complejidad factorial** si su complejidad es $O(n!)$. El algoritmo que genera todas las maneras en las que un viajante podría visitar n ciudades tiene complejidad factorial; discutiremos este algoritmo en el Capítulo 8.

Un problema que se puede resolver utilizando un algoritmo con complejidad polinómica en el peor caso se llama **tratable,** pues se espera que el algoritmo produzca la solución al problema para una entrada de tamaño razonable en un tiempo relativamente corto. Sin embargo, si el polinomio de la estimación en notación O tiene un grado alto (como, por ejemplo, 100) o si los coeficientes son extremadamente grandes, el algoritmo puede necesitar un tiempo muy grande para solucionar el problema. Por tanto, el que ese problema se pueda solucionar utilizando un algoritmo con complejidad polinómica en el peor caso no es una garantía de que el problema se pueda solucionar en un tiempo razonable, incluso para valores de entrada pequeños. Afortunadamente, en la práctica, el grado y los coeficientes de los polinomios de tales estimaciones son pequeños.

La situación es mucho peor para problemas que no se pueden resolver utilizando un algoritmo con complejidad polinómica en el peor caso. Tales problemas se llaman **intratables.** Generalmente, pero no siempre, se requiere un tiempo extremadamente largo para resolver un problema en el peor caso incluso para un valor pequeño de entrada. En la práctica, no obstante, hay situaciones en las que un algoritmo puede resolver un problema mucho más rápidamente en la mayoría de los casos que en el peor caso. Cuando podamos permitir que algunos casos, quizá pocos, no se puedan resolver en un tiempo razonable, la complejidad en tiempo en el caso promedio es

una medida más acertada del tiempo que llevaría resolver un problema que el peor caso . Muchos de los problemas importantes en la industria son presentados como intratables, pero en la práctica pueden resolverse para, esencialmente, todos los conjuntos de datos que aparecen en la vida real. Otra forma en la que se manejan los problemas intratables cuando aparecen en las aplicaciones prácticas es buscar soluciones aproximadas del problema en vez de soluciones exactas. Puede darse el caso de que existan algoritmos rápidos para encontrar soluciones aproximadas, quizá incluso con la garantía de que no difieren mucho de las soluciones exactas.

Existen algunos problemas para los cuales incluso se puede demostrar que no existen algoritmos que puedan resolverlos. Tales problemas se llaman **no resolubles** o **irresolubles** (en oposición a los problemas **resolubles**, para los cuales existen algoritmos que los resuelven). La primera demostración de que hay problemas irresolubles se debe al gran matemático e informático Alan Turing. El problema que Turing demostró que era irresoluble es el **problema de la parada.** Este problema toma como entrada un programa junto con la entrada de este programa. El problema pregunta si el programa se interrumpirá cuando se ejecute con la entrada del programa. Estudiaremos este problema en la Sección 3.1. (En el Capítulo 11 se puede encontrar una biografía de Alan Turing y una descripción de algunos de sus otros trabajos).

El estudio de la complejidad de algoritmos va mucho más allá de lo que podemos describir aquí. Ten en cuenta, no obstante, que se cree que muchos problemas resolubles tienen la propiedad de que no hay ningún algoritmo con complejidad polinómica en el peor caso que los resuelva, pero que en caso de conocerse alguna solución puede comprobarse que ésta, efectivamente, resuelve el problema en tiempo polinómico. Se dice que un problema pertenece a la **clase NP** si se puede comprobar en tiempo polinómico que una solución efectivamente lo es (los problemas tratables se dicen que pertenecen a la **clase P**)*. Hay también una clase importante de problemas, llamados **problemas NP-completos,** con la propiedad de que si alguno de estos problemas se puede resolver haciendo uso de un algoritmo con complejidad polinómica en el peor caso, entonces todos ellos se pueden resolver por medio de algoritmos con complejidad polinómica en el peor caso.

Enlaces

El problema de la satisfabilidad es un ejemplo de problema NP-completo –podemos verificar rápidamente que una asignación de valores de verdad a las variables de una fórmula proposicional la hace verdadera, pero no se ha descubierto un algoritmo con complejidad en tiempo polinómico que genere tal asignación de valores de verdad–. [Por ejemplo, una búsqueda exhaustiva sobre todos los posibles valores de verdad requiere $\Theta(2^n)$ operaciones con bits, donde n es el número de variables de la fórmula]. Además, si se conociese un algoritmo de complejidad polinómica para resolver este problema, podríamos encontrar algoritmos de complejidad polinómica para resolver todos los problemas de esta clase (y hay muchos problemas importantes en esta clase).

A pesar del gran esfuerzo invertido en investigación, no se han encontrado algoritmos con complejidad computacional polinómica en el peor caso para esta clase de problemas. Es generalmente aceptado, aunque no se ha demostrado, que ningún problema NP-completo puede resolverse en un tiempo polinómico. Para más información acerca de la complejidad de algoritmos, consulta las referencias para esta sección al final del libro, en particular [CoLeRi90].

Observa que una estimación en notación O para la complejidad en tiempo de un algoritmo expresa cómo el tiempo requerido para resolver un problema cambia a medida que el tamaño de la entrada crece. En la práctica, se usa la mejor estimación que se pueda encontrar (esto es, la función de referencia más pequeña posible). No obstante, la estimación con la notación O de la complejidad no ofrece información sobre el tiempo de computación real utilizado. Una razón es que la estimación de la función O para la función $f(n)$ depende de dos constantes C y k ($f(n)$ es $O(g(n))$ si $f(n) \leq Cg(n)$, cuando $n > k$, donde C y k son constantes). Sin conocer las constantes C y k, la estimación no se puede utilizar para dar una cota superior del número de operaciones realizadas. Además, como se observó anteriormente, el tiempo requerido para completar una operación depende del tipo de operación y del ordenador utilizado. (También se debe tener en cuenta que la notación O estima la complejidad del algoritmo proporcionando una cota superior, pero no inferior, del tiempo requerido en el peor caso para resolver un problema en función del tamaño de la entrada. Para proporcionar una cota inferior se debería utilizar la notación Zeta. No obstante, por simplicidad, utilizaremos la notación O para estimar la complejidad de algoritmos, aun sabiendo que la notación Zeta proporcionará siempre más información).

* NP proviene de la traducción inglesa de la expresión tiempo *polinómico no determinista*.

Podemos determinar el tiempo real requerido por un algoritmo para solucionar un problema de tamaño especificado si todas las operaciones utilizadas se pueden reducir a las operaciones con bits que realiza el ordenador. La Tabla 2 muestra el tiempo necesario para resolver problemas de varios tamaños con un algoritmo que realiza el número de operaciones con bits que se indica. Con un asterisco se indican tiempos mayores a 10^{100} años. (En la Sección 2.5 estudiaremos el número de operaciones con bits que se necesitan para sumar y multiplicar dos enteros). Para la construcción de esta tabla se ha considerado que cada operación con bits requiere 10^{-9} segundos, el tiempo que invierte el ordenador más rápido a día de hoy. En el futuro, estos tiempos decrecerán con el aumento de velocidad de los ordenadores de nueva generación.

Tabla 2. Tiempo de ordenador utilizado por los algoritmos.

Tamaño del problema	*Operaciones con bits utilizadas*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-9} s	10^{-8} s	3×10^{-8} s	10^{-7} s	10^{-6} s	3×10^{-3} s
10^2	7×10^{-9} s	10^{-7} s	7×10^{-7} s	10^{-5} s	4×10^{13} años	*
10^3	$1{,}0 \times 10^{-8}$ s	10^{-6} s	1×10^{-5} s	10^{-3} s	*	*
10^4	$1{,}3 \times 10^{-8}$ s	10^{-5} s	1×10^{-4} s	10^{-1} s	*	*
10^5	$1{,}7 \times 10^{-8}$ s	10^{-4} s	2×10^{-3} s	10 s	*	*
10^6	2×10^{-8} s	10^{-3} s	2×10^{-2} s	17 min	*	*

Es importante saber cuánto tiempo necesitará un ordenador para resolver un problema. Por ejemplo, si un algoritmo requiere 10 horas, puede valer la pena invertir el tiempo (y el dinero) requerido para resolver el problema. Pero si un algoritmo requiere diez mil millones de años para resolverlo, no sería razonable utilizar recursos para implementar este algoritmo. Uno de los fenómenos más interesantes de la tecnología moderna es el tremendo aumento de la memoria y la velocidad de los ordenadores. Otro factor importante que reduce el tiempo necesario para resolver los problemas con el ordenador es el **procesado en paralelo,** que es la técnica de realizar secuencias de operaciones simultáneamente.

Los algoritmos eficientes, incluidos muchos algoritmos con complejidad polinómica, son los que más se benefician de los avances tecnológicos significativos. Sin embargo, estos avances ayudan poco a superar la problemática de algoritmos con complejidad en tiempo exponencial o factorial. Debido al avance en la velocidad de cálculo, aumento de la memoria disponible y del uso del procesado en paralelo, muchos problemas que se consideraban imposibles de resolver hace cinco años ahora se resuelven de manera rutinaria, y de hecho, dentro de cinco años, esta afirmación volverá a ser cierta.

Problemas

1. ¿Cuántas comparaciones se hacen en el algoritmo dado en el Problema 16 de la Sección 2.1 para encontrar el menor número natural de una sucesión de n números naturales?

2. Escribe un algoritmo que ponga los cuatro primeros términos de una lista de longitud arbitraria en orden creciente. Muestra que este algoritmo tiene complejidad $O(1)$ en términos del número de comparaciones realizadas.

3. Supongamos que sabemos que un elemento está entre los cuatro primeros de un una lista de 32. ¿Qué tipo de búsqueda localizaría antes este elemento: una búsqueda lineal o una binaria?

4. Determina el número de multiplicaciones necesarias para calcular x^{2^k}, comenzando por x y elevando al cuadrado sucesivamente (para encontrar x^2, x^4 y así sucesivamente). ¿Es éste un método más eficiente para calcular x^{2^k} que multiplicar x por sí mismo el número de veces apropiado?

5. Da una estimación en notación O del número de comparaciones realizadas por el algoritmo que determina el nú-

mero de unos de una cadena de bits examinando cada bit de la cadena para determinar si es un uno (véase el Problema 25 de la Sección 2.1).

*6. a) Muestra que este algoritmo determina el número de bits 1 en la cadena S:

```
procedure cuenta_bit(S: cadena de bits)
cuenta := 0
while S ≠ 0
begin
  cuenta := cuenta + 1
  S := S ∧ (S − 1)
end {cuenta es el número de unos en S}
```

Aquí $S-1$ es la cadena de bits obtenida al cambiar el bit 1 más a la derecha de S por un 0 y todos los bits 0 a la derecha de éste por unos. [Recuerda que $S \wedge (S-1)$ es la operación bit *AND* entre S y $S-1$].

b) ¿Cuántas operaciones bit *AND* se necesitan para encontrar el número de bits 1 en S?

7. El algoritmo convencional para evaluar el polinomio $a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ en $x = c$ se puede expresar en pseudocódigo como

```
procedure polinomio(c, a_0, a_1, ..., a_n: números
  reales)
potencia := 1
y := a_0
for i := 1 to n
begin
  potencia := potencia * c
  y := y + a_i * potencia
end {y = a_n c^n + a_{n-1} c^{n-1} + ... + a_1 c + a_0}
```

donde el valor final de y es el valor del polinomio en $x = c$.

a) Evalúa $3x^2 + x + 1$ en $x = 2$ siguiendo el algoritmo paso a paso.

b) ¿Exactamente cuántas sumas y multiplicaciones se hacen para evaluar el polinomio de grado n en $x = c$? (No cuentes el número de sumas utilizadas en el incremento de la variable del bucle).

8. Hay un algoritmo más eficiente (en términos del número de multiplicaciones y sumas realizadas) para evaluar polinomios que el algoritmo convencional descrito en el problema anterior. Se llama **método de Horner.** Este pseudocódigo muestra cómo se usa este método para calcular los valores de $a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ en $x = c$:

```
procedure Horner(c, a_0, a_1, a_2, ..., a_n: números
  reales)
y := a_n
for i := 1 to n
  y := y * c + a_{n-i}
{y = a_n c^n + a_{n-1} c^{n-1} + ... + a_1 c + a_0}
```

a) Evalúa $3x^2 + x + 1$ en $x = 2$ siguiendo paso a paso el algoritmo.

b) ¿Cuántas sumas y multiplicaciones se realizan exactamente para evaluar el polinomio de grado n en $x = c$? (No cuentes el número de sumas realizadas en el incremento de la variable del bucle).

9. ¿De qué tamaño puede ser un problema que se debe resolver en un segundo usando un algoritmo que requiere $f(n)$ operaciones con bits, donde cada operación se hace en 10^{-9} segundos, con los siguientes valores para $f(n)$?

a) $\log n$ b) n c) $n \log n$
d) n^2 d) 2^n e) $n!$

10. ¿Cuánto tiempo necesita un algoritmo para resolver un problema de tamaño n si el algoritmo utiliza $2n^2 + 2^n$ operaciones con bits y cada una requiere 10^{-9} segundos, con los siguientes valores para n?

a) 10 b) 20
c) 50 c) 100

11. ¿Cuánto tiempo necesita un algoritmo que realiza 2^{50} operaciones con bits si cada operación requiere estos tiempos?

a) 10^{-6} segundos
b) 10^{-9} segundos
c) 10^{-12} segundos

12. Determina el número mínimo de comparaciones, o el comportamiento en el mejor caso,

a) requeridas para encontrar el máximo de una sucesión de n enteros usando el Algoritmo 1 de la Sección 2.1;
b) realizadas para localizar un elemento de una lista de n términos con una búsqueda lineal;
c) realizadas para localizar un elemento de una lista de n términos con una búsqueda binaria.

13. Analiza el comportamiento en el caso promedio del algoritmo de búsqueda lineal si exactamente la mitad de las veces el elemento x no está en la lista y si, cuando x está en la lista, puede estar en cualquier posición.

14. Un algoritmo se dice que es **óptimo** para la solución de un problema con respecto a una operación especificada si no hay ningún otro algoritmo que lo resuelva con un número inferior de operaciones.

a) Muestra que el Algoritmo 1 de la Sección 2.1 es un algoritmo óptimo con respecto al número de comparaciones de enteros. (*Indicación:* No se cuentan las comparaciones realizadas en el incremento de la variable del bucle).
b) ¿Es el algoritmo de búsqueda lineal óptimo respecto al número de comparaciones de enteros (sin incluir las comparaciones realizadas en el incremento de la variable del bucle)?

15. Describe la complejidad en el peor caso, medida en términos de comparaciones, del algoritmo de búsqueda ternaria descrito en el Problema 27 de la Sección 2.1.

16. Describe la complejidad en el peor caso, medida en términos de comparaciones, del algoritmo descrito en el Problema 28 de la Sección 2.1.

17. Analiza la complejidad en el peor caso del algoritmo que propusiste en el Problema 29 de la Sección 2.1 para localizar una moda en una lista no decreciente de enteros.

18. Analiza la complejidad en el peor caso del algoritmo que propusiste en el Problema 30 de la Sección 2.1 para obtener todas las modas de una lista no decreciente de enteros.

19. Analiza la complejidad en el peor caso del algoritmo que propusiste en el Problema 31 de la Sección 2.1 para encontrar el primer término de una sucesión de enteros que es igual a algún término previo.

20. Analiza la complejidad en el peor caso del algoritmo que propusiste en el Problema 32 de la Sección 2.1 para obtener todos los términos de una sucesión que son mayores que la suma de todos los anteriores a él.

21. Analiza la complejidad en el peor caso del algoritmo que propusiste en el Problema 33 de la Sección 2.1 para determinar el primer término de una sucesión menor que el elemento que le precede inmediatamente.

22. Analiza la complejidad en el peor caso, en términos del número de comparaciones, del algoritmo del Problema 5 de la Sección 2.1, que determina todos los valores que aparecen más de una vez en una lista ordenada de enteros.

23. Analiza la complejidad en el peor caso, en términos del número de comparaciones, del algoritmo del Problema 9 de la Sección 2.1, que determina si una cadena es un palíndromo.

24. ¿Cuántas comparaciones realiza la ordenación por selección (véase el preámbulo al Problema 41 de la Sección 2.1) para ordenar n elementos? Utiliza tu respuesta para dar una estimación en notación O de la complejidad de la ordenación por selección en términos del número de comparaciones.

25. Obtén una estimación en notación O para la complejidad en el peor caso en términos del número de comparaciones realizadas y el número de términos intercambiados en la ordenación por inserción binaria descrita en el preámbulo al Problema 27 de la Sección 2.1.

26. Muestra que el algoritmo voraz para dar cambio de n céntimos usando monedas de 25, 10, 5 y 1 céntimos tiene complejidad $O(n)$ medida en términos de las comparaciones requeridas.

27. Describe cómo cambia el número de comparaciones realizadas en los siguientes algoritmos para buscar un elemento de una lista cuando el tamaño de la lista se duplica de n a $2n$, para n entero positivo
    a) el de búsqueda lineal;
    b) el de búsqueda binaria.

28. Describe cómo cambia el número de comparaciones realizadas en los siguientes algoritmos cuando el tamaño de la lista que debe ser ordenada se duplica de n a $2n$, para n entero positivo, para los siguientes algoritmos de ordenación
    a) el método de la burbuja;
    b) ordenación por inserción;
    c) ordenación por selección (descrito en el preámbulo del Problema 41 de la Sección 2.1);
    c) ordenación por inserción binaria (descrito en el preámbulo del Problema 47 de la Sección 2.1).

2.4 Enteros y división

INTRODUCCIÓN

La parte de la matemática discreta que estudia los enteros y sus propiedades pertenece a la rama de las matemáticas conocida como **teoría de números.** Esta sección es el comienzo de una introducción en tres secciones a la teoría de números. En esta sección repasaremos algunos conceptos básicos de la teoría de números, entre los que se incluyen la divisibilidad, el máximo común divisor y la aritmética modular. En la Sección 2.5 describiremos varios algoritmos importantes de teoría de números, enlazando el material de las Secciones 2.1 y 2.3 sobre algoritmos y su complejidad con las nociones que se presentan en esta sección. Por ejemplo, introduciremos algoritmos para encontrar el máximo común divisor de dos enteros positivos y para desarrollar la aritmética computacional utilizando expresiones binarias (en base 2). Finalmente, en la Sección 2.6 continuaremos el estudio de la teoría de números presentando algunos resultados importantes y sus aplicaciones a la aritmética computacional y a la criptología, el estudio de los mensajes secretos.

Las ideas que desarrollaremos en esta sección se basan en la noción de divisibilidad. Un concepto importante basado en la divisibilidad es el de número primo. Un primo es un entero mayor que 1 que es divisible sólo por 1 y por él mismo. Determinar si un número es primo es importante en las aplicaciones a la criptología. Un importante teorema de la teoría de números, el Teorema