Capítulo 1

Algoritmos

1. Introducción
2. Algoritmos y máquina de Turing
3. Lenguaje algorítmico
4. Análisis de algoritmos
5. Comparación de algoritmos
6. Clasificación de algoritmos

El objetivo de este capítulo es proporcionar el marco formal y las herramientas básicas que permitan la descripción y el estudio de los algoritmos que se presentarán al largo de este libro. Después de una introducción general al concepto de algoritmo, este se precisa en la sección 2 dentro del modelo de cómputo de la máquina de Turing en el que se consideran los algoritmos. A continuación se presenta la notación o lenguaje algorítmico que se usará, y en la sección 4 se dan los fundamentos del análisis de algoritmos, que nos permitirá realizar la comparación entre diferentes algoritmos de resolución de un mismo problema (sección 5) e introducir la clasificación general de algoritmos de la sección 6, objetivo fundamental del capítulo.

1.1 Introducción

El concepto general de algoritmo es seguramente conocido por el lector. El Diccionario de la Lengua Española de la Real Academia Española define el término *algoritmo* como “conjunto ordenado y finito de operaciones que permite hallar la solución de un problema”. Básicamente un algoritmo es, entonces, una descripción de cómo se realiza una tarea: la resolución de un problema concreto. Por procedimiento entendemos una secuencia de instrucciones tales que hechas en el orden adecuado llevan al resultado deseado. Otra característica que se desprende

de la definición es que un mismo algoritmo puede resolver todos los problemas de una misma clase. En un sentido general, la noción de algoritmo no es exclusiva de la matemática. También tienen una relación muy próxima al algoritmo una receta de cocina, una partitura musical, una guía turística de una ciudad o un museo, un manual de instrucciones de un electrodoméstico, etc.

El primer punto que se debe considerar es la identificación de los problemas que resolverá el algoritmo. Consideraremos problemas bien definidos y adecuados para ser solucionados mediante el uso de computadores digitales. También se debe tratar de problemas que pueden caracterizarse por su dimensión. Para cada tamaño puede haber diferentes *instancias* del problema. Por ejemplo, si el problema considerado es la ordenación de un conjunto de elementos, la dimensión del problema sería la cardinalidad del conjunto y para una dimensión fijada se pueden dar instancias diferentes que corresponderán al conjunto concreto de elementos que se considere.

El algoritmo debe tener en cuenta el *modelo* abstracto dentro del cual tratemos el problema. No usaremos el mismo modelo si tenemos que ordenar un conjunto de libros por temas que si queremos ordenar un conjunto de números naturales. El modelo, de hecho, es un inventario de las herramientas a nuestra disposición para tratar el problema y una descripción de la manera de usarlas.

Para la solución del problema se tratará de codificar la instancia considerada para que constituya la entrada del algoritmo. El algoritmo produce una salida que, una vez decodificada, es la solución del problema. Solucionar el problema quiere decir ser capaces de realizar este proceso para cualquier tamaño e instancia del problema.

Habitualmente, los algoritmos se escriben pensando en el ejecutor o procesador del algoritmo (máquina o persona). El procesador debe ser capaz de interpretarlo y de ejecutarlo. Si se trata de una máquina, será preciso que esté en su lenguaje específico. De esta manera hay una serie de pasos intermedios entre la descripción del algoritmo y el programa final que lo ejecuta; véase la figura 1.1.

Una característica importante deseable para un algoritmo es su transportabilidad, consecuencia de su generalidad, la cual debe permitir la adaptación del algoritmo para que pueda ser ejecutado por máquinas muy diferentes.

Podemos ahora precisar un poco más el concepto de algoritmo: Además de estar constituido por una secuencia de operaciones que llevan a la resolución de un tipo concreto de problemas dentro de un modelo prefijado, en un algoritmo es preciso que se verifique:

Existencia de un conjunto de entradas. Debe existir un conjunto específico de objetos cada uno de los cuales son los datos iniciales de un caso particular del problema que resuelve el algoritmo. Este conjunto se llama conjunto de entradas del problema.

Definibilidad. Cada paso debe ser definible de forma precisa y sin ninguna ambigüedad.

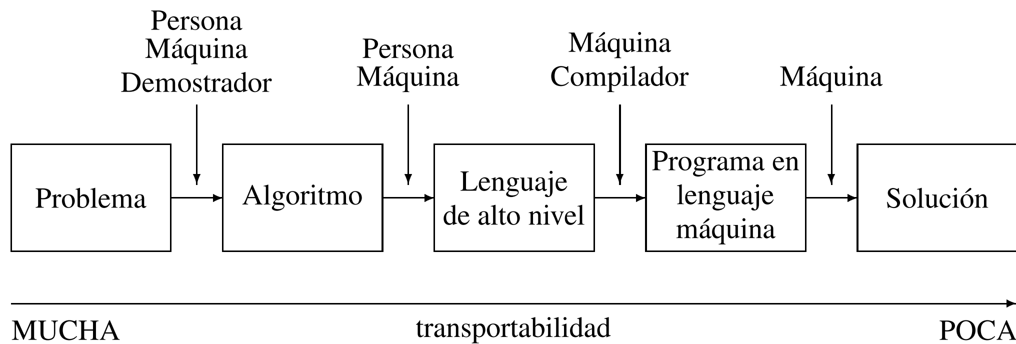

Figura 1.1: Proceso de solución de un problema mediante un algoritmo

Efectividad. Todas las operaciones a realizar en el algoritmo han de ser suficientemente básicas para que se puedan hacer en un tiempo finito en el procesador que ejecuta el algoritmo.

Finitud. El algoritmo debe acabar en un número finito de pasos. Un *método de cálculo* puede no tener esta restricción.

Corrección. El algoritmo debe ser capaz de encontrar la respuesta correcta al problema planteado.

Predecibilidad. Siempre consigue el mismo resultado para un mismo conjunto de entradas.

Existencia de un conjunto de salidas. El resultado del algoritmo asociado al conjunto de entradas.

También se consideran características importantes de un algoritmo la *claridad* y la *concisión*: Un algoritmo claro y breve resulta más sencillo de programar, a la vez que es más sencillo comprobar si es correcto.

Con esta definición comprobamos que una receta de cocina no encaja totalmente con la noción de algoritmo. La receta tiene un conjunto de entradas (ingredientes) y de salidas (el plato guisado); las instrucciones pueden, en principio, hacerse efectivas (se supone que se dispone de los utensilios convenientes—el *hardware*—y que el cocinero no es torpe); se verifica la finitud, pero falla la definibilidad, ya que son ambigüas frases típicas de una receta como 'añadir una *pizca* de sal', 'revolver *lentamente*', o bien, '*esperar* que se haya reducido el líquido', y también falla su predecibilidad.

Un mismo problema se puede resolver usando algoritmos diferentes. Los algoritmos podrán tener una complejidad diferente, la cual afectará al tiempo de ejecución y a la ocupación

de memoria, y posiblemente se distinguirán también por la exactitud de los resultados a que llevan.

Consideremos, por ejemplo, el cálculo del máximo común divisor de dos enteros. Recordemos que el máximo común divisor, mcd, de dos enteros positivos es el entero positivo más grande que los divide. Por ejemplo $\mathrm{mcd}(225,945) = 45$. Si alguno de los enteros es cero, el mcd se define como el otro entero. Así $\mathrm{mcd}(14,0) = 14$ y $\mathrm{mcd}(0,0) = 0$.

Un primer procedimiento para encontrarlo puede consistir en descomponer cada uno de los números en primos y considerar el producto de las potencias más bajas de cada primo común. En el caso de 180 y 380 escribiremos $180 = 2^2 \cdot 3^2 \cdot 5$ y $380 = 2^2 \cdot 5 \cdot 19$ y por tanto el $\mathrm{mcd}(180,380) = 2^2 \cdot 5 = 20$. Este método, si bien es el que se suele enseñar en las escuelas, es muy ineficiente cuando los números considerados son grandes (más de 5 dígitos), dado que la descomposición en primos es difícil. El mismo problema puede ser resuelto, como estudiaremos en la sección 4, usando el clásico algoritmo de Euclides, que consiste en dividir el número más grande por el más pequeño y retener el resto. De forma sucesiva se hace el cociente del divisor y el resto anteriores hasta que el resto sea cero. En este caso, el último resto calculado diferente de cero es el máximo común divisor de los dos números iniciales. Aplicándolo al mismo ejemplo de antes: 380 dividido por 180 tiene por resto 20. 180 dividido por 20 tiene resto cero. Así, el máximo común divisor de 380 y 180 es 20. Más adelante entraremos en detalle sobre el análisis de algoritmos, el objetivo del cual es precisamente el estudio y la comparación de algoritmos.

Hay otros puntos en relación a la algorítmica que son importantes pero en los cuales no entraremos. Mencionemos en primer lugar la cuestión del diseño de algoritmos: ¿Existen algoritmos para diseñar algoritmos? ¿Todo proceso tiene un algoritmo que lo describe? Este tipo de cuestiones lleva a un área importante: la teoría de la computabilidad. Otro aspecto es la comprobación de lo que realmente hace un algoritmo. Métodos como la especificación formal (por ejemplo, el *Vienna Development Method* o **Z**) son útiles para este tipo de estudios.

1.2 Algoritmos y máquina de Turing

La máquina de Turing no es un objeto físico, sino un artificio matemático que nos proporciona un modelo de computación en el cual encuadramos el análisis de nuestros algoritmos. Es preciso decir que hay otros modelos igualmente válidos, pero no tan comunes en la literatura. Destacamos las *máquines de acceso aleatorio*, mucho más realistas que la de Turing en el sentido que emulan en su estructura un ordenador real, véase por ejemplo [2] o [6]. Todos los modelos son equivalentes, pero la máquina de Turing tiene un carácter más elemental que hace más sencillo comprender su aplicación en el estudio de algoritmos.

Una *máquina de Turing* consiste en un cabezal de lectura/escritura por el cual pasa una

cinta infinita que puede moverse hacia adelante y hacia atrás. La cinta se encuentra dividida en casillas que pueden estar vacías o llevar un símbolo de un determinado alfabeto. De hecho, con un solo símbolo (además de la posibilidad de dejar la casilla vacía) es suficiente, ya que cualquier otro alfabeto lo podríamos expresar en secuencias de este símbolo y casillas vacías. En nuestro ejemplo, figura 1.2, se usa un solo símbolo (un cuadrado negro).

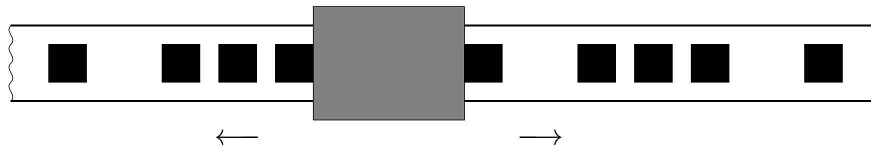

Figura 1.2: Máquina de Turing

El cabezal se encuentra en cada instante en un estado concreto de entre un número finito de estados posibles diferentes. La máquina funciona paso a paso y cada uno de los pasos consiste en situar el cabezal delante de una casilla y, después de leer el contenido de la casilla, realizar, de acuerdo con el resultado de la lectura y el estado interno de la máquina, las tres acciones siguientes: borrar la casilla y dejarla vacía, o bien, escribir un símbolo que puede ser el mismo que había antes, pasar a un nuevo estado (que puede ser el mismo) y finalmente mover la cinta una casilla en una de las dos direcciones posibles, o bien, acabar el cómputo. La conducta global de la máquina viene determinada por un *conjunto de instrucciones*, las cuales indican, para cada posible estado y símbolo leído, las tres acciones que es preciso hacer. Si se han de dar datos iniciales, éstos se escriben en la cinta de acuerdo con el sistema de codificación considerado. El cabezal se sitúa en la casilla inicial y, una vez se acaba el cómputo, la máquina entra en un estado especial de *parada* y deja de funcionar. La posible respuesta se encontrará escrita en la misma cinta a partir de la posición donde se encuentre el cabezal. Cada instrucción se puede representar con una *quíntupla* $(e_a, s_a; e_d, s_d, m)$, donde e_a indica el estado de la máquina cuando se lee el símbolo s_a, y e_d es el estado de la máquina después de dejar escrito el símbolo s_d en la casilla procesada y mover la cinta a la derecha o la izquierda según indica m.

Ya que tanto el conjunto de estados como el de símbolos es finito, cualquier cómputo puede ser especificado completamente por un conjunto finito de quíntuplas. Supongamos también que el cómputo es determinístico en el sentido que, dados el estado de la máquina y el símbolo de la cinta antes de cualquier acción, existe una quíntupla que determina el nuevo estado de la máquina, el símbolo a escribir en la cinta y el movimiento que debe hacer. Se suele decir entonces que la máquina de Turing es *determinista*.

Gracias a la máquina de Turing, un algoritmo puede ser definido de forma precisa como

una secuencia de instrucciones que determina totalmente la conducta de la máquina para la resolución del problema considerado. La llamada *hipótesis de Church–Turing* dice que todo algoritmo puede ser descrito (o implantado) en una máquina de Turing.

Ejemplo 1.1. En este ejemplo, el alfabeto que se usará tiene un solo símbolo, el 1. Los enteros positivos se representan por una secuencia de unos. El número total de unos es el entero considerado (representación unaria). La máquina tiene cinco estados posibles etiquetados 0,1,2,3 y A (el estado especial de parada). El programa tiene por objeto determinar si un cierto entero es par o impar. Si es par, la máquina escribirá un 1 y se parará. Si es impar dejará la casilla vacía y también se parará. La salida del programa, 1 o vacío, se encontrará en la cinta a continuación del entero con una casilla vacía como separador. Se supone que el cabezal se encuentra, en el momento de iniciar la computación, sobre el primer dígito del entero, y que el resto de dígitos están a la derecha de éste. La figura 1.3 especifica las quíntuplas que forman el conjunto de instrucciones y la acción del programa. Si la entrada es el entero 4 (1111 en la notación unaria empleada), D y E expresan el movimiento de la cinta hacia la derecha o la izquierda, el símbolo $*$ indica la irrelevancia de expresar el contenido de la posición correspondiente de la quíntupla y el símbolo b indica que la casilla está en blanco. La posición del cabezal viene dada por la flecha.

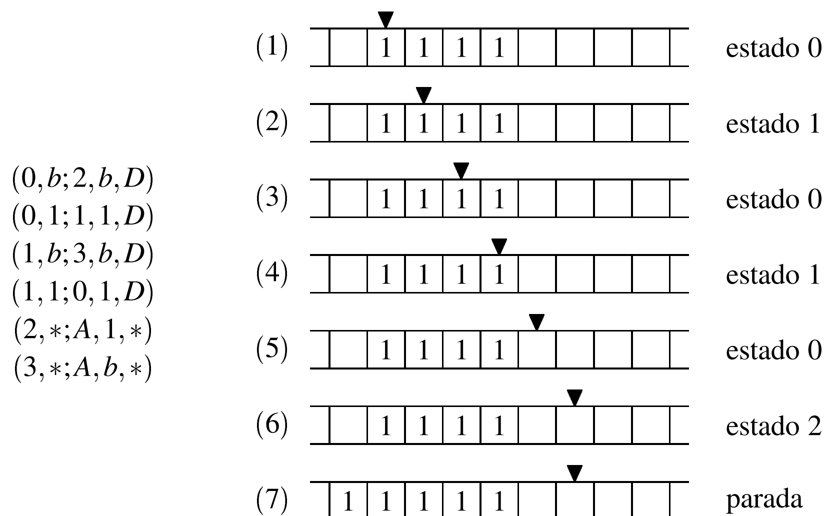

Figura 1.3: Conjunto de intrucciones y ejecución de un programa para determinar la paridad del entero 4 en una máquina de Turing

Ejemplo 1.2. El conjunto de instrucciones de la figura 1.4 describe una implantación del al-

goritmo de Euclides en una máquina de Turing. Se supone que los dos enteros están escritos en la cinta en representación unaria y separados por una casilla vacía. El cabezal se encuentra, en el momento de iniciar el cómputo, sobre esta casilla. Se deja al lector la comprobación de su funcionamiento.

```
(0,b;0,b,D)     (0,1;1,1,E)     (1,b;2,1,D)     (1,1;1,1,E)
(2,b;10,b,D)    (2,1;3,b,D)     (3,b;4,b,D)     (3,1;3,1,D)
(4,b;4,b,D)     (4,1;5,b,D)     (5,b;7,b,E)     (5,1;6,1,E)
(6,b;6,b,E)     (6,1;1,1,E)     (7,b;7,b,E)     (7,1;8,1,E)
(8,b;9,b,E)     (8,1;8,1,E)     (9,b;2,b,D)     (9,1;1,1,E)
(10,b;A,b,*)    (10,1;10,1,D)
```

Figura 1.4: Conjunto de intrucciones que describen la implantación del algoritmo de Euclides en una máquina de Turing

1.3 Lenguaje algorítmico

Un algoritmo puede ser expresado o formulado de maneras muy diferentes. Podemos usar únicamente el idioma habitual o emplear presentaciones gráficas más o menos complicadas e independientes de un lenguaje natural como, por ejemplo, un diagrama de flujos. Muy a menudo se usan metalenguajes que combinan una descripción en un lenguaje natural con unos símbolos o palabras clave que expresan algunas de las acciones bien definidas que es preciso efectuar.

De hecho, y pensando en el ejemplo de la receta de cocina, ésta equivaldría al algoritmo, pero podría ser escrita en catalán, francés o ruso (diferentes *lenguajes de 'programación'*), a pesar que el resultado obtenido, por ejemplo un pastel, sería equivalente fuera cual fuese el lenguaje empleado. Por otra parte, aunque en una receta de cocina no se acostumbra a usar un metalenguaje, sí que se suele presentar de forma estructurada ordenando los ingredientes uno bajo el otro, indicando claramente los diferentes pasos, etc.

En este libro usamos esencialmente el lenguaje natural con algunas palabras y símbolos que tienen el papel de metalenguaje. La indentación del texto también es importante y permite distinguir las diferentes partes y estructuras del algoritmo. La tabla 1.1 muestra algunos de los principales elementos de este metalenguaje.

Nos hemos decidido por esta aproximación, porque al mismo tiempo que permite dar una cierta estructura visual al algoritmo que facilita su lectura y su comprensión se evita la complejidad a que lleva muy a menudo un diagrama de flujos. Al mismo tiempo es suficientemente

Tabla 1.1: Principales elementos de la notación algorítmica que se usa en este libro

---

Asignación

variable ← *expresión*

El valor de *expresión* pasa a ser el nuevo valor de *variable*.

Decisión

Si *condición* **entonces hacer** $instrucción_1$ **si no hacer** $instrucción_2$

Si es cierta la *condición* se ejecuta la $instrucción_1$, si no se ejecuta la $instrucción_2$.

Repetición

Hacer $instrucción_1 \ldots instrucción_n$ **hasta que** *condición*

Hasta que sea cierta la *condición* se ejecutan todas las instrucciones desde $instrucción_1$ hasta $instrucción_n$.

También hay otras maneras de hacer la repetición:

Para $variable = val_{inicial}$ **hasta** val_{final} **hacer** $instrucción_1 \ldots instrucción_n$.

Repetir $val_{final} - val_{inicial}$ veces la ejecución de todas las instrucciones desde $instrucción_1$ hasta $instrucción_n$.

Repetir hasta *condición* $instrucción_1 \ldots instrucción_n$

Repetir, hasta que sea cierta *condición*, todas las instrucciones desde $instrucción_1$ hasta $instrucción_n$.

Mientras *condición* **hacer** $instrucción_1 \ldots instrucción_n$

Repetir, mientras sea cierta *condición*, todas las instrucciones desde $instrucción_1$ hasta $instrucción_n$.

---

concreto para poder ser ejecutado con facilidad. Todos los algoritmos que se describen en este libro, de hecho, tendrían que poderse programar sin demasiadas dificultades en cualquier lenguaje estructurado de alto nivel como *C* o *Pascal*.

Finalmente, un elemento importante en el lenguaje algorítmico y que usamos a menudo es la *recursividad*. Los algoritmos que consideramos usualmente son por su propia definición *iterativos* (se encaminan hacia la solución paso a paso). Un algoritmo se llama *recursivo* si se define a sí mismo. En un algoritmo iterativo, una parte de él puede ejecutarse diversas veces (por ejemplo, en una estructura repetitiva); sin embargo, en un algoritmo recursivo se realiza la reejecución del algoritmo completo desde el comienzo (normalmente con un conjunto diferente de entradas). También usaremos a veces *procedimientos* que se pueden considerar como algoritmos y que se llaman desde dentro del algoritmo principal. Lógicamente pueden existir procedimientos recursivos; por extensión también se llama *algoritmo recursivo* aquel que contiene algún procedimiento recursivo.

1.4 Análisis de algoritmos

Para resolver un problema determinado y una vez decidido el modelo para el tratamiento del problema, pueden existir diversos algoritmos. El análisis de algoritmos es entonces esencial para la realización práctica y eficiente del algoritmo, como por ejemplo su programación en un ordenador. Siempre es de interés perfeccionar el método de resolución de un problema allá donde signifique un esfuerzo menor y, si tenemos en cuenta la figura esquemática del proceso que nos lleva del problema a la solución (Fig. 1.1), lógicamente será más sencillo mejorar el algoritmo que la codificación en lenguaje máquina.

Una manera de comparar dos algoritmos que resuelven un mismo problema podría ser representarlos en un determinado lenguaje de programación, encontrar la solución al problema usando el mismo ordenador y medir el tiempo que tardan en encontrar la solución. Como el tiempo que tarda el ordenador en encontrar la solución es proporcional al número de operaciones elementales que este debe efectuar (sumas, productos, etc.), se suele llamar *tiempo de ejecución* a este número de operaciones.

De los diferentes parámetros que caracterizan un algoritmo, uno de los que nos puede interesar más es precisamente el *tiempo*, o número de operaciones básicas que necesita el algoritmo para solucionar el problema en función del tamaño de la entrada. Otro parámetro que a veces se considera es el *espacio* de memoria que necesita el algoritmo para su ejecución. Como normalmente estaremos más interesados en la rapidez que en el almacenamiento en memoria, cuando analizemos un algoritmo estudiaremos, si no se dice lo contrario, el tiempo de ejecución.

Estos dos parámetros, así como también la misma comparación de algoritmos, mantienen su sentido y son perfectamente tratables en el contexto de la máquina de Turing: La comple-

jidad temporal de una máquina de Turing determinista en función del número de casillas que ocupa la entrada vendrá dada, en este caso, por el número máximo de pasos que puede llegar a hacer la máquina considerando todas las posibles entradas del mismo tamaño. La complejidad espacial viene dada por la distancia máxima—en número de casillas—que puede desplazarse la cinta por delante del cabezal a partir de la casilla inicial.

Además, por ejemplo, si un cierto algoritmo tiene una complejidad temporal polinómica en términos de una máquina de Turing, mantiene la complejidad polinómica usando argumentos más informales basados en el número de operaciones. Por esta razón usaremos habitualmente esta manera de analizar la eficiencia de un algoritmo para evitar la carga que supone trabajar con máquinas de Turing.

Vamos a verlo en un ejemplo: Consideremos dos posibles algoritmos para el cálculo de x^n donde x es un real y n un natural. El primero simplemente va haciendo un bucle acumulando el producto de x por sí mismo n veces:

Entrada: x: real, n: entero.
Algoritmo x^n, (1)
 1. $y \leftarrow 1.0$, $i \leftarrow 0$
 2. $y \leftarrow yx$, $i \leftarrow i+1$
 3. **Si** $i = n$ **entonces salir**
  sino ir al paso 2
Salida: y.

Observemos que, si calculamos por ejemplo x^{10000}, este algoritmo efectuará 10000 veces el bucle principal.

El segundo algoritmo es conceptualmente más complicado. Para calcular x^n usa la representación binaria de n para determinar cuales de las potencias $x, x^2, x^4, \ldots$ son necesarias para construir x^n y así considerar sólo su producto. Usa dos operadores, `and` y `shr`, que muchos lenguajes de programación tienen incorporados. El primero actúa bit a bit sobre la representación binaria de los dos enteros considerados y retorna el entero que resulta de tomar 1 si los dos bits son 1 y 0 en cualquier otro caso. Por ejemplo: $923 \texttt{ and } 123 = 1110011011_2 \texttt{ and } 0001111011_2 = 0000011011_2 = 27$. El segundo desplaza, en la reprentación binaria, todos los dígitos una posición hacia la derecha añadiendo un 0 a la izquierda y pierde el dígito de la derecha. Equivale a hacer la división entera por 2. Por ejemplo: $\texttt{shr } 235 = \texttt{shr } 11101011_2 = 01110101_2 = 117$.

Entrada: x: real, n: entero.

Algoritmo x^n, (2)

1. $y \leftarrow 1.0$, $z \leftarrow x$.
2. **Si** $(n\ \texttt{and}\ 1) \neq 0$ **entonces** $y \leftarrow yz$.
3. $z \leftarrow zz$.
4. $n \leftarrow\ \texttt{shr}\ n$.
5. **Si** $n \neq 0$ **entonces** ir al paso 2.

Salida: y.

Aplicándolo también para calcular x^{10000} y como $10000 = 10011100010000_2$, es decir, tiene 14 dígitos binarios, el algoritmo sólo hace 14 veces su bucle principal. Así, en general, el primer algoritmo efectúa un número de pasos aproximadamente igual a n, mientras que, para el segundo, el número de pasos es proporcional al número de dígitos que tiene n representado en binario ($\log_2 n$). El segundo algoritmo necesita un tiempo de ejecución mucho menor que el primero para conseguir el mismo resultado.

En este punto conviene introducir una notación útil para estimar la eficiencia de los algoritmos. Cuando describimos que el número de operaciones, $f(n)$, que efectúa un algoritmo depende de n, se pueden ignorar contribuciones pequeñas. Lo que es preciso conocer es un orden de magnitud para $f(n)$ válido para todo n salvo, quizá, un número finito de casos especiales. Así:

Definición 1.3. Sea f una función de $\mathbb{N}$ en $\mathbb{N}$. Decimos que $f(n)$ es $O(g(n))$ si existe una constante k positiva tal que $f(n) \leqslant kg(n)$ para todo $n \in \mathbb{N}$ (salvo posiblemente un número finito de excepciones).

Con esta notación se dice que el primer algoritmo para el cálculo de x^n es $O(n)$ y el segundo $O(\log n)$.

Podemos ahora dar los pasos que es preciso efectuar en el análisis de algoritmos:

1. Describir el algoritmo de forma precisa.
2. Definir el tamaño n de una instancia característica del problema.
3. Calcular $f(n)$, número de operaciones que efectúa el algoritmo para resolver el problema.

De hecho, incluso para un mismo tamaño de datos n, el algoritmo puede tener, como veremos, comportamientos muy diferentes dependiendo de la instancia considerada. En este caso será preciso hacer tres tipos diferentes de análisis:

1. Análisis del caso más favorable: Qué rapidez posee el algoritmo bajo las condiciones más favorables.

2. Análisis del caso medio: Cuál es el rendimiento medio del algoritmo considerando todos los posibles conjuntos de datos iniciales (normalmente asumiendo que todos son igualmente probables).

3. Análisis del caso más desfavorable: Qué rapidez posee en el peor caso (es decir, para datos de entrada que hacen que el algoritmo tenga el peor rendimiento).

La primera opción no suele ser demasiado útil. La segunda es, en general, complicada de calcular y podría ser poco significativa si muchos de los conjuntos de datos afectan poco o son improbables en la práctica. El estudio del caso más desfavorable acostumbra a ser el más utilizado. De todas maneras es preciso tratarlo, teniendo presente siempre el problema que se está solucionando.

Vamos a utilizar nuevamente el algoritmo de Euclides para ilustrar ahora el análisis de algoritmos. Este algoritmo, como bien sabemos, encuentra el máximo común divisor de un par de números enteros positivos.

La recursión juega un papel importante en el algoritmo de Euclides. El punto esencial está en el hecho de que cualquier factor común de m y n lo es también de $m-n$, y también que cualquier factor común de n y $m-n$ lo es de m, ya que $m = n+(m-n)$. Así $\mathrm{mcd}(m,n) = \mathrm{mcd}(m-n,n)$.

Consideremos los enteros m y n y vamos a encontrar su mcd. Supongamos que dividiendo m por n encontramos un cociente q_1 y un resto r_1, es decir, $m = nq_1 + r_1$ con $q_1 \geqslant 0$ y $0 \leqslant r_1 < n$. Como $r_1 = m - nq_1$, aplicando q_1 veces este razonamiento, obtendremos que $\mathrm{mcd}(m,n) = \mathrm{mcd}(n,r_1)$. Repitiendo el proceso encontramos:

$$\begin{array}{rclrcl} m & = & nq_1 + r_1 & 0 & \leqslant & r_1 < n \\ n & = & r_1q_2 + r_2 & 0 & \leqslant & r_2 < r_1 \\ r_1 & = & r_2q_3 + r_3 & 0 & \leqslant & r_3 < r_2 \\ & \vdots & & & \vdots & \\ r_{k-1} & = & r_kq_{k+1} + r_{k+1} & 0 & \leqslant & r_{k+1} < r_k \\ r_k & = & r_{k+1}q_{k+2} & & & \end{array}$$

Y por tanto:

$$\mathrm{mcd}(m,n) = \mathtt{mcd}(n,r_1) = \mathrm{mcd}(r_1,r_2) = \cdots = \mathrm{mcd}(r_k,r_{k+1}) = r_{k+1}$$

Por ejemplo, los restos sucesivos que encontramos cuando se calcula el mcd de 315 y 91 son 42 y 7 y entonces $\mathrm{mcd}(315,91) = 7$.

Dar el algoritmo es ahora directo:

Entrada: m,n: enteros.
Algoritmo Euclides
  1. Dividir m por n. Sea r el resto ($0 \leqslant r < n$).
  2. **Si** $r = 0$ **entonces salir**.
  3. $m \leftarrow n$, $n \leftarrow r$. Ir al paso 1.
Salida: n.

Analizaremos este algoritmo estudiando el peor caso. Este pasará cuando en cada paso decrece mínimamente la sucesión. Así comencemos por el final: El último valor será 0. El penúltimo 1. Ahora tiene que seguir el valor más pequeño tal que dividido por 1 dé 0 de resto: 1. A continuación el valor más pequeño que dividido por 1 dé 1 de resto: 2. La sucesión que construimos será:

$$0,1,1,2,3,5,8,13,21,34,55,89,144,\ldots$$

Curiosamente, esta sucesión es la conocida *sucesión de Fibonacci*[1]. El peor caso para el algoritmo de Euclides corresponde, entonces, a considerar dos números de Fibonacci sucesivos. Procuremos estimar el orden del algoritmo observando la sucesión: Si los enteros iniciales considerados son los primeros términos de la sucesión, con 2 dígitos son precisos 6 pasos, y con 3 dígitos son precisos 12 pasos, para 100 dígitos son precisos unos 500 pasos. Así vemos que es de orden aproximadamente igual al número de dígitos de los enteros afectados. Para n grande, podemos ver fácilmente que el número medio de iteraciones para calcular $\mathrm{mcd}(m,n)$ es $O(\log n)$. En efecto, usando que el número de Fibonacci, número k es $n = F_k = (((1+\sqrt{5})/2)^k - ((1-\sqrt{5})/2)^k)/\sqrt{5}$—sección 4.3—, y considerando k grande, entonces n es proporcional a $((1+\sqrt{5})/2)^k$. Tomando logaritmos vemos que $k = O(\log n)$ y, como k es precisamente el índice del número, n es por tanto la cantidad de pasos a efectuar cuando se aplica el algoritmo.

El análisis resultará más sencillo en el caso del algoritmo de resolución de un problema que es en cierta manera clásico: el problema de las torres de Hanoi. Este juego, propuesto en 1883 por el matemático francés Édouard Lucas, consiste en tres palos, uno de los cuales contiene un número determinado de discos de diferentes tamaños puestos uno sobre el otro de mayor a menor. Se trata de pasar todos los discos a otro palo, dejándolos también de mayor a menor, y de acuerdo con las reglas:

1. En cada movimiento sólo se puede mover el disco de arriba de un palo a otro.

2. No se puede poner un disco sobre uno de diámetro más pequeño.

[1] Véase el capítulo 4.

El problema consiste en presentar un algoritmo óptimo para resolver el problema dando el número de movimientos que es preciso hacer.

Comenzaremos proponiendo un algoritmo recursivo.

Entrada: $N_discos, Palo_inicial, Palo_final$
Algoritmo HANOI
Procedimiento $H(n,r,s)$. `[Mueve n discos del palo r al s]`
Si $n = 1$ **entonces** mueve un disco de r a s.
sino hacer $H(n-1,r,t)$ `[t es el palo que no es ni r ni s]`
mueve un disco de r a s `[Sólo queda el de abajo de todo]`
$H(n-1,t,s)$
1. *H(N_discos,Palo_inicial,Palo_final)* `[Llamada principal]`
Salida: Los movimientos que se van haciendo.

La figura 1.5 muestra gráficamente cómo se ejecuta el algoritmo.

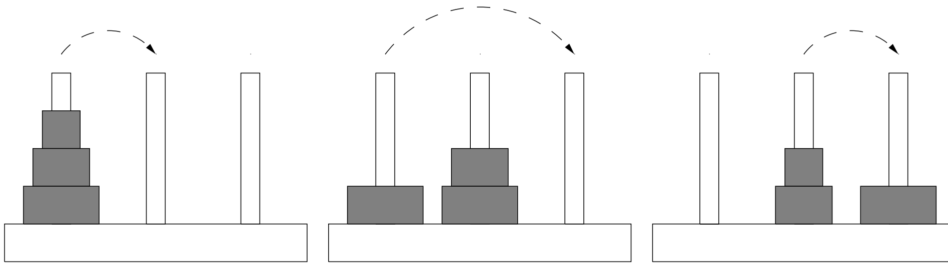

Figura 1.5: Resolución del problema de las torres de Hanoi en el caso de tres discos

Para estudiar su eficiencia vamos a determinar el total de movimientos $h(n)$ que efectúa el algoritmo cuando hay n discos en el palo p_1 y los queremos pasar al palo p_3. Está claro que $h(1) = 1$. El algoritmo dice que, si $n > 1$ y suponiendo que los discos $d_1, d_2, \ldots, d_n$ (d_n es el de abajo de todo) están en el palo p_1, lo que es preciso hacer es pasar los $n-1$ discos $d_1, d_2, \ldots, d_{n-1}$ al palo p_2 (esto son $h(n-1)$ movimientos), después pasar d_n a p_3 y finalmente pasar los $n-1$ discos de p_2 a p_3. El número total de movimientos será:

$$h(n) = 2h(n-1) + 1, \quad n \geqslant 2$$

Solucionar esta ecuación recursiva no es difícil y, de hecho, en el capítulo 4 se tratarán métodos para resolver este tipo de ecuaciones. Aquí veremos cómo podemos encontrar $h(n)$ por inducción. Observemos primero que $h(0) = 0$, $h(1) = 1$, $h(2) = 3$, $h(3) = 7$, $h(4) = 15$, $h(5) = 31$,

etc. Esto nos sugiere que quizá $h(n) = 2^n - 1$. Demostrémoslo. Supongámoslo cierto para $n-1$, es decir, que es cierto que $h(n-1) = 2^{n-1} - 1$. Demostraremos que es cierto para n. En efecto:

$$h(n) = 2h(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$$

En cuanto al comportamiento del algoritmo, resulta que, tal como se ha planteado el problema, el mejor caso coincide con el peor y con el medio. Así, el algoritmo es $O(2^n)$. Si lo hiciésemos a mano, y moviésemos 1 disco por segundo, en el caso de 8 discos tendríamos que hacer 255 movimientos y tardaríamos unos 4 minutos. Suponiendo un movimiento cada microsegundo, para mover 60 discos harían falta 36600 años. Ya vemos, entonces, que un análisis de este algoritmo nos lleva a detectar un comportamiento fundamentalmente diferente al del algoritmo de Euclides. Finalmente, y para este algoritmo, podemos tratar también la cuestión de la complejidad y demostrar que no es posible encontrar un algoritmo que resuelva el problema con menos movimientos. Supongamos que $\tilde{h}(n)$ sea el número mínimo de movimientos para el algoritmo mejor posible. Consideremos la relación entre $\tilde{h}(n+1)$ y $\tilde{h}(n)$. Para el caso con $n+1$ discos, el disco de abajo de todo se tendrá que mover en algún momento. Antes de hacerlo, los otros n discos habrán de estar en uno de los otros palos. Esto quiere decir que al menos se habrán hecho $\tilde{h}(n)$ movimientos y, similarmente, cuando se haya cambiado de sitio el disco de abajo, se tendrán que hacer al menos $\tilde{h}(n)$ movimientos más para poner el resto de discos encima de él. Por tanto $\tilde{h}(n+1)$ debe valer como mínimo $2\tilde{h}(n) + 1$. Dado que, lógicamente, $\tilde{h}(1) = 1$, resulta de la ecuación última que $\tilde{h}(n) \geqslant h(n)$ para todo n. Así, $\tilde{h}(n)$ es el número de movimientos para el mejor algoritmo.

1.5 Comparación de algoritmos

En la sección anterior hemos visto como es posible estudiar el comportamiento de un algoritmo y encontrar su complejidad temporal (o espacial) en función del tamaño de los datos de entrada. Dada, por ejemplo, la relación directa que hay entre la complejidad temporal de un algoritmo y el tiempo de cálculo de su implantación, podemos ver fácilmente qué tipo de problemas son tratables en la práctica. Si usásemos una máquina que realizase una operación básica cada microsegundo, podríamos escribir la tabla 1.2.

Observar que, aunque mejoremos mucho la rapidez de las máquinas, la tabla no se modifica esencialmente. Suponiendo que lleguemos a tener en un futuro próximo máquinas 1000 veces más rápidas, esto simplemente significaría dividir por 1000 los valores de la tabla, cosa que, evidentemente, no afecta el comportamiento asintótico. Es más importante, entonces, conseguir buenos algoritmos para que ninguna instancia del problema conlleve una resolución con un tiempo superior al polinómico.

Tabla 1.2: Tiempo de cálculo según la complejidad (una instrucción por microsegundo)

	$n = 10$	$n = 20$	$n = 40$	$n = 60$	$n = 1000$
n	0.00001 s	0.00002 s	0.00004 s	0.00006 s	0.001 s
n^2	0.0001 s	0.0004 s	0.0016 s	0.0036 s	1 s
n^3	0.001 s	0.008 s	0.064 s	0.216 s	17 m
2^n	0.001 s	1 s	12.7 días	36 534 años Cromagnon	
$n!$	3.629 s	77 094 años Neanderthal	$2.6\ 10^{34}$ años Edad del Universo $1.5\ 10^{10}$ años	$2.6\ 10^{68}$ años	

Así pues, la elección del algoritmo es una parte esencial asociada también a su análisis. Para ilustrar este hecho dedicaremos esta sección a comparar diversos algoritmos de ordenación de listas. El objetivo de los algoritmos es ordenar una lista de elementos de un conjunto según la relación de orden que hay. Esto incluye la ordenación alfabética de nombres, la ordenación de reales, etc.

El primer algoritmo considerado es el llamado *algoritmo de inserción*. Consiste en ir considerando uno a uno los elementos de la lista, a partir del segundo, e insertarlos en la posición que les corresponda comparando con los anteriores hasta encontrar su sitio.

Entrada: Una lista, $L = \{a_1, a_2, \ldots, a_n\}$, con n elementos no ordenados.
Algoritmo INSERCIÓN
  Para $i = 2$ **hasta** n
    $tmp \leftarrow a_i$
    $j \leftarrow i - 1$
    repetir hasta que $j = 0$ **o bien** $tmp \geqslant n_j$
      $n_{j+1} \leftarrow n_j$
      $j \leftarrow j - 1$
    $n_{j+1} \leftarrow tmp$
Salida: La lista L ordenada.

El número de comparaciones que efectúa el algoritmo es el aspecto crítico en su análisis. En el caso peor, cuando la lista esté ordenada al revés, serán precisas $1 + 2 + 3 + \cdots + (n-1) = n(n-1)/2$ comparaciones, esto es, un polinomio $O(n^2)$ en el tamaño de entrada.

¿Cuál es el número de comparaciones en el caso medio? El algoritmo de inserción es precisamente uno de aquellos casos en que el análisis del caso medio no es mucho más difícil de efectuar que el análisis del peor caso. Es preciso en primer lugar encontrar el valor medio de la posición en la cual insertar el elemento a_i, ya que esto nos da el número medio de comparaciones para cada i. Consideremos que todas las $i!$ posibles ordenaciones son igualmente probables. Con esto, la posición donde insertar a_i puede ser cualquiera entre la primera (a_i es menor que cualquiera de los $i-1$ valores ya ordenados) y la i (a_i es más grande que a_{i-1}). Si el sitio correcto donde insertar a_i es la posición j, entonces la cantidad de comparaciones es $i-j+1$ donde $j=2,3,\ldots,i$. Si a_i se tuviese que insertar en la posición $j=1$, entonces el número de comparaciones es $i-1$. El número medio de comparaciones resulta ser:

$$\begin{aligned}
\frac{1}{i}\left(i-1+\sum_{j=2}^{i}(i-j+1)\right) &= \frac{1}{i}\left[i-1+\sum_{k=1}^{i-1}k\right] \\
&= \frac{1}{i}\left(i-1+\frac{i(i-1)}{2}\right) \\
&= \frac{(i-1)(i+2)}{2i} = \frac{i^2+i-2}{2i} \\
&= \frac{i}{2}+\frac{1}{2}-\frac{1}{i}
\end{aligned}$$

Para encontrar el número total de comparaciones sólo es preciso sumar esto entre 2 y n y encontramos finalmente:

$$\frac{1}{2}\left[\frac{n(n+1)}{2}-1\right]$$

Otro algoritmo $O(n^2)$ es el *algoritmo de burbujas*. Este algoritmo es simple de describir. La idea es pasar por la lista intercambiando dos elementos sucesivos si están en orden incorrecto. Al final de la primera pasada, el elemento más grande quedará situado en el sitio correcto. Se hace ahora otra pasada y será el segundo más grande el que se situará en su sitio. En total serán precisas $n-1$ pasadas para conseguir ordenar la lista. También vemos que en la segunda pasada no es preciso llegar al final, sólo es preciso considerar los $n-1$ primeros elementos. Análogamente, $n-2$ en la tercera, etc. El número total de comparaciones es el mismo que con el algoritmo de inserción. El nombre del algoritmo proviene de la asociación con el hecho de que en cada pasada el elemento mayor se va desplazando hacia un extremo como si fuese una burbuja dentro de un líquido que subiese a la superficie.

Otro algoritmo conocido de ordenación es el algoritmo *mergesort*. Es un ejemplo clásico de la técnica algorítmica conocida como *dividir y vencer*. La idea consiste en dividir la lista en dos por la mitad y ordenar cada mitad recursivamente. Al final se combinan las dos listas manteniendo el orden creciente.

Damos primero el algoritmo de combinación de listas:

Entrada: Dos listas $L_1 = \{a_1, a_2, \dots, a_r\}$ y $L_2 = \{b_1, b_2, \dots, b_s\}$
Algoritmo COMBINA(L_1,L_2) `[`L_1 `y` L_2 `tienen` r `y` s `elementos ordenados]`

1. $i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow 1$.
2. **2.1**. **Si** $a_i \leqslant b_j$ **hacer** $c_k \leftarrow a_i$ y **Si** $i < r$ **hacer** $i \leftarrow i+1$.
   sino hacer $c_k \leftarrow b_j$ y **Si** $j < s$ **hacer** $j \leftarrow j+1$
   2.2. **Si** $i = r$ **hacer** $a_r \leftarrow b_s$ y **Si** $j = s$ **fer** $b_s \leftarrow a_r$.
3. $k \leftarrow k+1$.
4. **Si** $k \leqslant r+s$ volver al paso 2.

Salida: La lista ordenada $L = \{c_1, c_2, \dots, c_{r+s}\}$.

El algoritmo MERGESORT de ordenación de listas para el caso de una lista de naturales viene dado por:

Entrada: La lista $L = \{a_i, a_{i+1}, \dots, a_j\}$ con n naturales.
Algoritmo MERGESORT

1. **Si** L tiene un sólo elemento la lista está ordenada. **Salir**.
2. $k \leftarrow \lfloor \frac{i+j}{2} \rfloor$.
   $L_1 \leftarrow \{a_i, a_{i+1}, \dots, a_k\}$.
   $L_2 \leftarrow \{a_{k+1}, a_{k+2}, \dots, a_j\}$.
3. Ordenar por separado L_1 y L_2 usando MERGESORT. `[llamada recursiva]`
4. $L \leftarrow$ COMBINA(L_1, L_2).

Salida: La lista ordenada L.

Podemos estudiar un poco la complejidad de este algoritmo. Para la etapa de combinación de listas se hacen $n-1$ comparaciones. Así, el total de pasos vendrá dado por

$$M(n) = M(\left\lfloor \frac{n}{2} \right\rfloor) + M(\left\lceil \frac{n}{2} \right\rceil) + n - 1, \quad M(1) = 0$$

Se trata, como en el caso de las torres de Hanoi, de solucionar esta ecuación recursiva. La estudiaremos en el caso en que n sea una potencia de 2. De entrada, cambiamos $n-1$ por n (encontramos una cota superior, de todas maneras $n-1$ correspondía al caso peor de la combinación de listas). Como $n = 2^k$, introduciendo $m(k) = M(2^k)$ la ecuación nos quedará: $m(k) = 2m(k-1) + 2^k, \quad m(0) = 0$. Esta ecuación es fácil de resolver.

Ejercicio 1.4. Demostrar por inducción que la solución de la ecuación

$$m(k) = 2m(k-1) + 2^k, \quad m(0) = 0$$

es $m(k) = k2^k$.

Así resulta que el número total de pasos es $M(n) \leqslant n \log n$. No es difícil de ver que este resultado es válido para cualquier n aunque no sea una potencia de 2. En resumen, estamos ante un algoritmo de ordenación $O(n \log n)$. Un inconveniente del algoritmo es la ocupación de memoria. Observad que en la etapa de combinación se crea una nueva lista del mismo tamaño que la inicial (de hecho puede ser modificado per evitar esto, a costa de aumentar su complejidad).

Finalmente presentamos el algoritmo *quicksort*. También es de la familia de algoritmos de *dividir y vencer*. Mientras en el algoritmo anterior se trata de partir la lista lo más exactamente posible, ahora lo que se hace es partirla creando nuevas listas de forma que una sublista tenga elementos inferiores o iguales a la otra.

Entrada: La lista $L = \{a_1, a_2, \dots, a_n\}$ de n naturales.
Algoritmo QUICKSORT
1. Si L tiene un solo elemento la lista está ordenada. **Salir**.
2. Considerar a_1 el primer elemento de la lista.
3. Separar el resto de la lista en dos sublistas L_1 y L_2
de forma que L_1 contiene los elementos anteriores a a_1 y L_2 los posteriores.
4. Ordenar por separado L_1 y L_2 con QUICKSORT. `[llamada recursiva]`
5. Concatenar la primera lista, el elemento a_1 y la segunda lista.
Salida: La lista ordenada L.

Ejercicio 1.5. Escribir un algoritmo para separar una lista de acuerdo con el paso 3 del algoritmo QUICKSORT.

Se puede ver que QUICKSORT tiene una complejidad temporal—número de comparaciones—$O(n \log n)$ (como MERGESORT). ¿Cuál de los dos es preferible?

1. El peor caso de QUICKSORT se sabe que es de orden más grande que el peor de MERGESORT. En general, los casos peores son poco probables. Así, este aspecto lo hemos de tener en cuenta, pero no es decisivo.

2. La utilización de la memoria en QUICKSORT es de orden n, mientras que en MERGESORT es de $2n$. Ésta es una cuestión importante si queremos emplear ordenadores pequeños para tratar grandes cantidades de datos.

3. Si se hace un análisis de la complejidad temporal para el caso medio, el factor resultante se inclina ligeramente a favor de MERGESORT. Si tenemos en cuenta que el movimiento de elementos de la lista no ha sido contado (sólo las comparaciones), en la práctica resulta que QUICKSORT es más rápido.

Con este estudio se ha pretendido mostrar que el análisis de algoritmos puede ayudar de forma importante en el diseño de un algoritmo para la resolución de un problema concreto. Sin embargo, como veremos más adelante, hay problemas que son intrínsecamente difíciles de tratar y para los cuales no se conocen algoritmos eficientes. Ser capaz de identificarlos y conocer sus limitaciones también será de utilidad.

1.6 Clasificación de algoritmos

Para poder discutir de una forma abstracta y general la eficiencia de diferentes algoritmos en la solución de problemas y así poder hacer una clasificación, es preciso en primer lugar hacer una consideración sobre los tipos de problemas que se nos pueden plantear:

- Problemas de búsqueda: Encontrar una cierta X en los datos de entrada que satisfaga la propiedad P.
- Problemas de transformación: Transformar los datos de entrada para que satisfagan la propiedad P.
- Problemas de construcción: Construir un conjunto C que satisfaga la propiedad P.
- Problemas de optimización: Encontrar el mejor X que satisface la propiedad P.
- Problemas de decisión: Decidir si los datos de entrada satisfacen la propiedad P.

Para una discusión abstracta sobre la posibilidad de resolver problemas mediante algoritmos eficientes, resulta conveniente reformularlos en términos de problemas de decisión, en el sentido de buscar una respuesta del tipo si/no. Esto nos permitirá poder comparar problemas muy diferentes. Por ejemplo, el problema de multiplicar dos enteros (dados x e y, ¿cuál es el valor de su producto?) puede presentarse también como: Dados los enteros x, y y z, ¿es cierto que $xy = z$?

Un problema de decisión se dice que es de tipo **P** si para su resolución son precisos algoritmos que realicen un número de operaciones básicas $O(p)$, donde p es un polinomio en

el tamaño del problema. Hay un tipo de problemas especialmente difíciles de tratar. Son los llamados **NP**, que quiere decir *no determinístico polinómico*. Estos problemas pueden ser resueltos en tiempo polinómico con una máquina de Turing no determinista o MTND, que puede ser definida como un conjunto de MTD procesando la cinta en paralelo. En este caso, la complejidad temporal en función de la longitud n de la entrada viene dada por el número de pasos máximo que puede hacer, considerando todas las posibles entradas de longitud n y tomando como tiempo correspondiente a una entrada el de la máquina de Turing determinista que ha tardado menos que todas las que computaban en paralelo. Una solución de un problema NP puede ser comprobada en tiempo polinómico.

Para muchos problemas NP se conocen cotas lineales eficientes, sin embargo las cotas conocidas para el caso peor son exponenciales.

Los problemas tipo P se incluyen claramente en el conjunto de problemas tipo NP. Una controversia conocida en el mundo de la algorítmica es si P es igual a NP. Se piensa que $P \neq NP$. El hecho que esta cuestión sea difícil de responder reside en la dificultad de probar que un problema *no* se puede resolver en tiempo polinómico. Para esto sería preciso considerar *todos* los posibles algoritmos de resolución y demostrar que todos ellos son ineficientes.

Se dice que un problema de decisión X se puede *transformar polinómicamente* en otro problema de decisión Y si, dadas entradas A y B, que pueden construirse una de la otra en tiempo polinomial respecto del original, A hace que X tenga respuesta afirmativa si y solamente si B hace que Y tenga respuesta afirmativa. Con esta noción se puede introducir la definición de problema **NP-C** (*NP-completo*). Un cierto problema X es del tipo NP-C si, además de ser NP, todos los otros problemas NP se pueden transformar en X polinómicamente. Se demuestra [4] [7] que, si se sabe resolver de forma eficiente uno de los problemas NP-C, quedan resueltos todos. De la misma manera, si se demuestra que uno cualquiera de los problemas de esta categoría no es tratable, todos los otros tampoco lo son.

Finalmente conviene comentar que, a pesar de la dificultad de dar algoritmos eficientes para la solución general de problemas del tipo NP-C, se conocen buenos métodos heurísticos que llevan a soluciones cuasi-óptimas. Así, en el capítulo 7 se presentarán algunos de ellos.

Notas históricas y bibliográficas

Aunque las primeras referencias escritas de algoritmos son de los griegos (por ejemplo, el algoritmo de Euclides, *Elementos*, libro 7, prop. 1-2), el nombre tiene origen en el matemático persa Abu Kha'far Muhammad ibn Musá Abdallah *al-Hwarizmi* al-Madjusi al-Qutrubulli, nacido en Hwarizm (Urgenč, Uzbekistán) hacia el año 780 y que trabajó gran parte de su vida en Bagdad. Así, la palabra algoritmo en realidad viene del nombre del pueblo donde nació este matemático, más conocido por su obra *Kitab al-jabr wa'l-muqabala*, que precisamente

da nombre a otra parte importante de las matemáticas: el álgebra. A pesar de los orígenes históricos realmente antiguos de ejemplos concretos de algoritmos, la formulación precisa de la noción de algoritmo es de este siglo. La aportación más importante es sin duda la de Alan Turing, matemático inglés que en el año 1932 introdujo el concepto de *máquina de Turing*, que ha contribuido enormemente al desarrollo de la teoría de la computabilidad. Otro hito a indicar fueron los resultados de S. A. Cook (1971) y L. Levin (1973) sobre los problemas NP-C. La algorítmica es una de las áreas científicas más activas. Entre los últimos avances interesantes es preciso destacar los trabajos (1988 y 1990) sobre complejidad estructural de José L. Balcázar, Josep Díaz y Joaquim Gabarró, profesores de la UPC, y el resultado de 1990 de Carsten Lund, Lance Fortnow y Howard Karloff de la Universidad de Chicago, Noam Nisan del MIT y Adi Shamir del Weizmann Institute, que de forma resumida dice que prácticamente cualquier problema, incluyendo los NP, tiene una verificación probabilista sencilla.

Bibliografía

[1] M. Abellanas, D. Lodares. *Análisis de Algoritmos y Teoría de Grafos.* RA-MA Editorial, 1990.

[2] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] J. L. Balcázar, J. Diaz, J. Gabarró. *Structural Complexity I (II)*, Springer–Verlag, 1988 (1990).

[4] S. A. Cook. “The complexity of theorem proving procedures”, *Proceeding of the 3rd. Annual ACM Symposium on the Theory of Computation*, pp. 151–158, 1971.

[5] D. Harel. *Algorithmics. The Spirit of Computing*. Addison-Wesley, 1987.

[6] L. Kučera. *Combinatorial Algorithms.* Adam Hilger, 1990.

[7] L. Levin. “Universal search problems”, *Problems of Information Transmission*, **9**, pp. 265–266, 1973.

[8] S. B. Maurer, A. Ralston. *Discrete Algorithmic Mathematics.* Addison-Wesley, 1991.

[9] G. J. E. Rawlins. *Compared to What? An Introduction to the Analysis of Algorithms.* Freeman, 1992.

Problemas

1. Se considera una máquina de Turing donde la cinta usa un alfabeto con los símbolos x, 1 y 2 (además de la posibilidad de tener la casilla vacía, b). Los estados son cinco, de los cuales dos son estados especiales de parada que llamamos A_{SI} y A_{NO}. Dad un programa que compruebe si un entero m, $m > 1$ divide exactamente a otro entero n. Para ello usar la representación unaria para m y n y entrar los datos en la cinta inicialmente como

$$[\dots, x, 1, 1, \dots, 1, b, 1, 1, \dots, 1, 1, x, \dots]$$

con m unos en la parte a la izquierda del cabezal, que se encontrará situado sobre la casilla vacía, y n unos a la derecha.

2. Sea P_n el peor caso, en cuanto a comparaciones, en que se puede encontrar el algoritmo QUICKSORT para ordenar una lista de n elementos.

   (a) Justificar las ecuaciones:

$$P_n = n - 1 + \max_{0 \leqslant i < n} (P_i + P_{n-i-1}); \quad P_0 = 0$$

   (b) Demostrar que la solución (única) de las ecuaciones anteriores es $P_n = n(n-1)/2$

3. (a) ¿Qué tipo de lista inicial lleva al peor comportamiento para el algoritmo de burbujas, en cuanto a comparaciones?

   (b) Realizar el análisis del peor caso para este algoritmo.