Algoritmia

Ricardo Baeza Yates,
Dpto. de Cs. de la Computación, Univ. de Chile
rbaeza@dcc.uchile.cl , **www.baeza.cl**

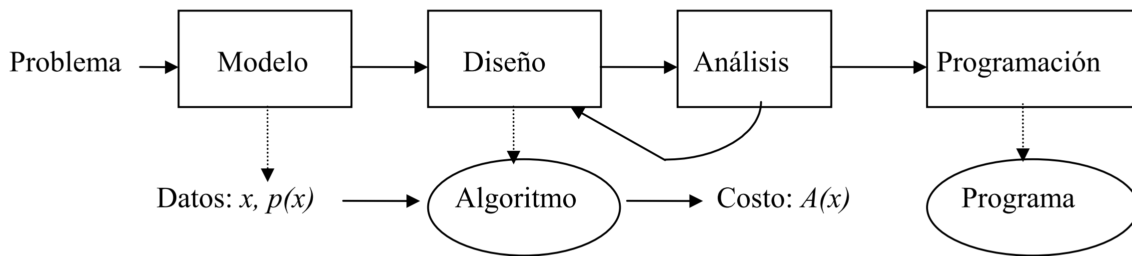
Introducción

Algoritmo, según la Real Academia, es un conjunto ordenado y finito de operaciones que permite encontrar la solución a un problema cualquiera. Ejemplos sencillos de algoritmos son una receta de cocina o las instrucciones para armar una bicicleta. Los primeros algoritmos registrados datan de Babilonia, originados en las matemáticas como un método para resolver un problema usando una secuencia de cálculos más simples. Esta palabra tiene su origen en el nombre de un famoso matemático y erudito árabe del siglo IX, Al-Khorezmi, a quien también le debemos las palabras guarismo y álgebra (ver anexo). Actualmente algoritmo se usa para denominar a la secuencia de pasos a seguir para resolver un problema usando un computador (ordenador). Por esta razón, la algoritmia o ciencia de los algoritmos, es uno de los pilares de la informática (ciencia de la computación en inglés).

En este artículo veremos distintos tipos de algoritmos y distintas técnicas para resolver problemas a través de varios ejemplos, muchos de ellos no computacionales. Todos los ejemplos resuelven variantes de un problema genérico: la búsqueda de información, un dilema que tenemos a diario. El objetivo final será encontrar el algoritmo que utilice menos operaciones o gaste menos recursos, dependiendo del caso.

Diseño y Análisis de Algoritmos

El desarrollo de un algoritmo tiene varias etapas (ver figura). Primero se modela el problema que se necesita resolver, a continuación se diseña la solución, luego ésta se analiza para determinar su grado de corrección y eficiencia, y finalmente se traduce a instrucciones de un lenguaje de programación que un computador entenderá. El *modelo* especifica todos los supuestos acerca de los datos de entrada y de la capacidad computacional del algoritmo. El *diseño* se basa en distintos métodos de resolución de problemas, muchos de los cuales serán presentados más adelante. Para el *análisis* de un algoritmo debemos estudiar cuántas operaciones se realizan para resolver un problema. Si tenemos un problema x diremos que el algoritmo realiza $A(x)$ operaciones (costo del algoritmo). Al valor máximo de $A(x)$ se le denomina el peor caso y al mínimo el mejor caso. En la práctica, interesa el peor caso, pues representa una cota superior al costo del algoritmo. Sin embargo, en muchos problemas esto ocurre con poca frecuencia o sólo existe en teoría. Entonces se estudia el promedio de $A(x)$, para lo cual es necesario definir la probabilidad de que ocurra cada x, $p(x)$, y calcular la suma ponderada de $p(x)$ por $A(x)$. Aunque esta medida es mucho más realista, muchas veces es difícil de calcular y otras ni siquiera podemos definir $p(x)$ porque no conocemos bien la realidad o es muy difícil de modelar. Si podemos demostrar que no existe un algoritmo que realice menos operaciones para resolver un problema, se dice que el algoritmo es *óptimo*, ya sea en el peor caso o en el caso promedio, dependiendo del modelo. Por esta razón, el análisis realimenta al diseño, para mejorar el algoritmo.



Búsqueda Secuencial

Veamos un primer ejemplo. Supongamos que hemos comprado el número x de una rifa en que se sortean n premios. El día del sorteo estamos presentes mientras se escogen los números ganadores. ¿Cuánto tiempo esperaremos para saber si hemos ganado algún premio? Analicemos el modelo. Podemos decir que el tiempo es proporcional a cada número premiado, así que nuestra operación básica será comparar x con cada número premiado. Enumeraremos la secuencia de números premiados de 1 a n y usaremos la notación *número(j)* para el j-ésimo número premiado. El diseño de la solución es inmediato: comparamos x con cada número premiado hasta que ganamos o no quedan más números. Veamos el análisis del peor caso: compararemos n veces, ya sea cuando ganamos con el último número o si el sorteo termina y no obtenemos ningún premio. Esto no es muy realista, pues si x es un número premiado, terminaremos antes. Para el caso promedio, supongamos que ganaremos un premio (esto no siempre es cierto pero simplifica los cálculos) y que la probabilidad de que x sea premiado es la misma para cada número, es decir $p(x) = 1/n$ es la probabilidad de que x sea igual a *número(j)* para cualquier j. Por lo tanto, con probabilidad $1/n$ compararemos j números, para cualquier j. Luego tenemos la suma de j/n desde j igual 1 hasta n, que es igual a $(n+1)/2$, aproximadamente la mitad de los números, lo que debiera coincidir con nuestra intuición. Este algoritmo se dice que es *en línea* (on-line) porque los datos (números) no se conocen de antemano, lo que es frecuente en la realidad. En el modelo usado este algoritmo es óptimo en el peor caso, pues siempre podemos no obtener un premio y realizar n comparaciones. Este algoritmo se llama *búsqueda secuencial* y lo presentamos en pseudo-código de programación a continuación:

Algoritmo: Búsqueda secuencial de x

```
Desde j ← 1 hasta n repetir {
    Si x = número(j) entonces { Ganamos }
}
Perdemos
```

Buscando un número

El ejemplo anterior es una simplificación del problema de buscar un número x en un conjunto de n números distintos. ¿Se puede hacer mejor? Sí. La idea es organizar los n números de manera que se agilice la búsqueda. La solución es ordenar los números, tal como en una guía o directorio telefónico, donde los nombres están ordenados alfabéticamente. Por supuesto ordenar estos números tiene un costo, pero la idea es que este costo se amortiza mediante muchas búsquedas. Si todos los datos son conocidos al comienzo, como en este caso, se dice que el algoritmo es *fuera de línea* (off-line). De lo contrario no podríamos ordenarlos ni tampoco acceder directamente a un número cualquiera. En este caso usaremos *número(j)* para denotar el j-ésimo número en el conjunto

ordenado. Al estar el conjunto ordenado, tenemos que *número(j) < número(j+1)* (recordar que no hay números repetidos). Por lo tanto, si comparamos x con *número(j)* sabemos que si es mayor sólo puede estar en una posición $k > j$. Por lo tanto descartamos una gran cantidad de números. Algo similar ocurre si x resulta ser menor. Para asegurar que siempre descartamos la misma cantidad de números escogemos $j = n/2$ [1]. Así, siempre eliminamos la mitad de los elementos en cada paso. Luego seguimos la búsqueda en la mitad correspondiente, hasta que quede sólo un número, el cual es comparado con x. Este es el método de *dividir para reinar* o *dividir para conquistar*, y a este algoritmo se le llama *búsqueda binaria* y se presenta a continuación:

Algoritmo: Búsqueda binaria de x ($n > 1$)

```
min ← 1;  max ← n
Mientras min < max repetir {
    j ← (max+min)/2                 [ mitad del segmento por revisar
    Si x > número(j) entonces {
        min ← j+1                   [ es mayor
    }
    sino {
        max ← j                     [ es menor
    }
}
Si x = número(min) entonces { Está }
sino { No está }
```

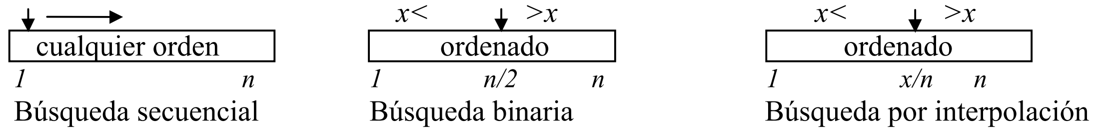
El análisis de búsqueda binaria no es tan sencillo. En cada paso dividimos el conjunto por dos hasta que sólo queda un número (*min*=*max*). Luego, si dividimos el conjunto k veces, dividiremos el problema por 2^k. Como queremos que quede sólo un elemento, debemos encontrar k tal que $n/2^k = 1$. Es decir $n=2^k$, lo que implica que $k=log_2(n)$[2]. Entonces comparamos k veces x y luego vemos si x es igual al número restante. Comparado con la búsqueda secuencial, la mejora es espectacular. Por ejemplo, si $n=2^{10}=1024$, tenemos que $log_2(n) + 1 = 11$ comparaciones en el peor caso en vez de 1024 que necesitaría la búsqueda secuencial. Este algoritmo tarda lo mismo en promedio, pues siempre hace el mismo número de comparaciones. Un algorítmico preguntaría: ¿Se puede hacer mejor?. En el peor caso no. Sin embargo en promedio sí se puede mejorar y es lo que hacemos todos los días cuando buscamos un número telefónico en la guía. Si estamos buscando Baeza, no abrimos la guía en la mitad sino que más cerca del comienzo, porque sabemos que las letras van de la A a la Z. Por lo tanto, si agregamos a nuestro modelo que conocemos el rango de valores de los números y que ellos están distribuidos de manera uniforme, podemos usar una *interpolación lineal*. De manera que si estamos buscando x entre los valores 1 a n y empezamos en la posición más cercana a x/n en vez de en la mitad. Este algoritmo se llama *búsqueda por interpolación* y su análisis es muy difícil. Si la distribución es uniforme, el número de comparaciones crece de forma proporcional a $2\ log_2(log_2(n))$ (es decir, dos veces el logaritmo

---

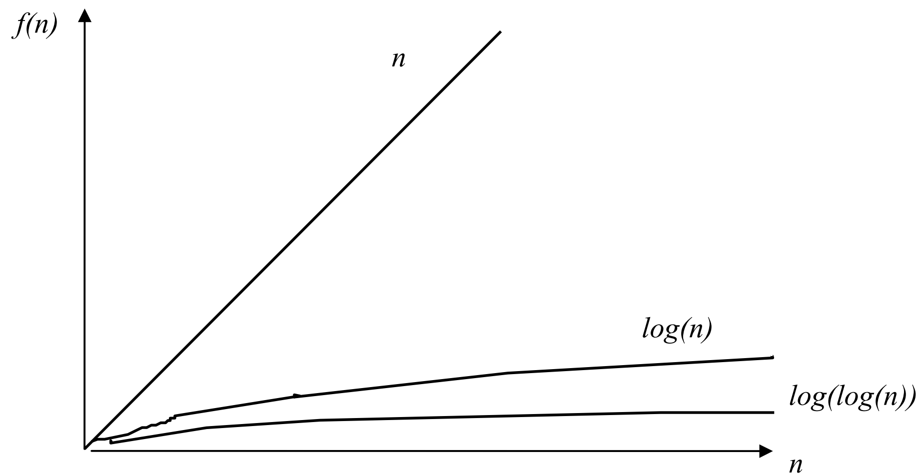
[1] Para simplificar supondremos que n es una potencia de 2, es decir, existe un $k > 0$ tal que $n=2^k$.

[2] El logaritmo en base b de un número x, $log_b(x)$, es tal que b elevado al logaritmo es igual a x.

del logaritmo). En nuestro ejemplo serían alrededor de 6 comparaciones. La desventaja es que en el peor caso, podríamos tener que hacer *n* comparaciones, aunque es muy poco probable. Por estas razones y el hecho de que su programación es también más complicada, generalmente se usa la búsqueda binaria. La siguiente figura esquematiza las diferencias entre los tres tipos de búsqueda descritos indicando la primera comparación:



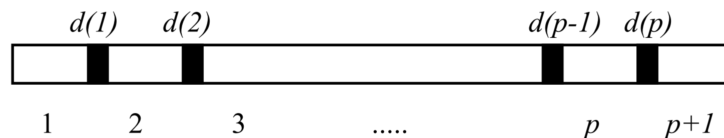
Para comparar dos algoritmos se usa la noción de orden, que indica cual es la relación entre el número de instrucciones ejecutadas (proporcional al tiempo de ejecución) y el tamaño del problema. Decimos que un algoritmo es de orden $f(n)$ (denotado por $O(f(n))$) si el tiempo de ejecución es menor o igual a una constante por $f(n)$, donde n representa el tamaño de los datos. En nuestro ejemplo de búsqueda, n es el número de elementos a buscar y podemos entonces decir que la búsqueda secuencial es $O(n)$ (lineal) , la búsqueda binaria es $O(log(n))$ (logarítmica) y la búsqueda por interpolación es $O(log(log(n)))$. El siguiente gráfico muestra la diferencia entre estos tres algoritmos.



Búsqueda en paralelo

Actualmente los computadores tienen más de un procesador. En otras palabras, pueden realizar varias operaciones al mismo tiempo. Los algoritmos que aprovechan esta capacidad de multiprocesamiento se llaman *paralelos*. Sin embargo, no todos los problemas pueden paralelizarse, pues en muchos casos hay operaciones que dependen de otras y por lo tanto no pueden realizarse al mismo tiempo. Podemos citar la búsqueda secuencial, donde los elementos arriban uno a uno y entonces no podemos procesarlos más rápido. Si tenemos todos los elementos ordenados en memoria, sí podremos agilizar la búsqueda. Supongamos que tenemos p procesadores y cada uno conoce el elemento buscado x (consideraremos que la memoria se puede leer en forma concurrente). Si $p \geq n$ podremos tener un procesador comparando cada elemento y el que encuentra el elemento buscado escribe su posición en una variable determinada. Por lo tanto tardaremos un tiempo

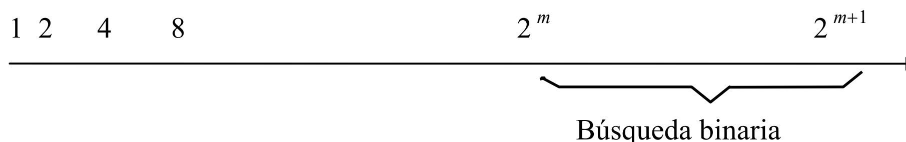
constante. Sin embargo, en la práctica, n es mucho más grande que el número de procesadores, y entonces necesitaremos otro algoritmo. La idea es dividir los elementos en $p+1$ partes, dejando fuera p elementos que dividen las partes y que llamaremos $d(i)$ (ver figura). El procesador j compara el elemento buscado x con los elementos $d(j-1)$ y $d(j)$ (el primer y último procesador se preocupan del caso en que el elemento es menor que $d(1)$ o mayor que $d(p)$, respectivamente). Si uno de ellos encuentra a x, terminamos. De otro modo, sólo uno de ellos encontrará que x debe estar en uno de los segmentos definidos. Luego aplicamos el mismo algoritmo en este segmento, hasta que quedan a lo más p elementos. En ese caso, podemos comparar cada elemento restante con x. Veamos el análisis. En cada etapa dividimos el problema en $p+1$ partes, así que en forma análoga a la búsqueda binaria el tiempo es de $O(\log_{p+1}(n))$. Debe tenerse en cuenta que si sólo hay un procesador tenemos logaritmo en base 2, obteniendo, como debía ser, la búsqueda binaria.



El uso del mismo algoritmo como parte de sí mismo se llama *recursividad*. Esta técnica permite simplificar el diseño y la exposición de muchos algoritmos, como veremos en otros ejemplos.

Un juego de números

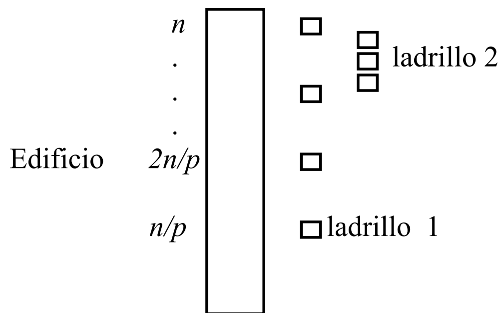
Considere el siguiente juego entre dos personas: una de ellas piensa un número entero positivo cualquiera y lo escribe en un papel; y la otra debe adivinar este número preguntando si es mayor, igual o menor a otro número. El juego termina cuando se adivina el número y el objetivo es realizar el menor número posible de preguntas. Este juego es similar al de buscar un número (ver secciones anteriores), pero en un rango no acotado, pues el valor del número es arbitrario. Si conociéramos un número mayor al que queremos adivinar, podríamos luego usar búsqueda binaria para encontrarlo. Para hallar un número mayor al que queremos adivinar, podemos ir preguntando por un número cada vez más grande. Hay muchas formas de hacer esto, pero si avanzamos lentamente tardaremos mucho y si avanzamos muy rápido, sobre estimaremos demasiado su valor. La mejor manera es ir doblando el número cada vez. Supongamos que preguntamos si es menor que 1, 2, 4, 8, etc. Existirá un valor m tal que el número es mayor que 2^m y menor que 2^{m+1} (si es que no es igual a uno de ellos), el cual, encontraremos después de hacer $m+1$ preguntas. Luego podemos usar búsqueda binaria entre esos dos valores para adivinar el número (ver figura). Como hay 2^m números entre 2^m y 2^{m+1}, necesitamos $\log_2(2^m) = m$ preguntas. En total son $2m+1$ preguntas. Notar que si n es el número adivinado, m es aproximadamente $\log_2(n)$ y entonces necesitamos $2\log_2(n)$ preguntas, lo que es sólo el doble de búsqueda binaria, pese a que el rango de búsqueda es infinito.



El número de comparaciones se puede mejorar a $log_2(n)$ preguntas más términos de menor orden usando la misma idea para buscar m aplicada en forma recursiva (es decir, sobre sí misma). Esta mejora queda como ejercicio para el lector.

Búsqueda con recursos acotados

En todos los ejemplos anteriores hemos supuesto que podemos preguntar un número ilimitado de veces. Sin embargo, no siempre esto es cierto, pues cada pregunta puede usar un recurso que es finito. Consideremos el siguiente problema ficticio. Una empresa necesita comprobar la resistencia de un nuevo tipo de ladrillos. Uno de los experimentos consiste en encontrar la altura máxima desde la cual se puede dejar caer un ladrillo sin que éste se rompa, realizando una serie de pruebas. Para ello se utiliza un edificio de n pisos. Sin embargo, sólo se tienen k de los ladrillos nuevos y queremos minimizar el número de pruebas. Esto significa que no podemos quebrar todos los ladrillos antes de encontrar la altura máxima. La primera tentación sería usar búsqueda binaria. De esta manera, tiramos un ladrillo desde la mitad del edificio, si se quiebra repetimos lo mismo en la mitad inferior, y sino seguimos en la mitad superior. En el peor caso, el ladrillo se quiebra y sólo nos quedarán $k-1$ ladrillos. Si k es menor que $log_2(n)$ (el peor caso de búsqueda binaria), quebraremos todos los ladrillos antes de encontrar la altura máxima. Por lo tanto este algoritmo sólo es útil cuando k es mayor o igual a $log_2(n)$. Si tan sólo hubiera un ladrillo, la única alternativa es usar búsqueda secuencial. Esto implica que debemos probar el piso 1, 2, 3, ... hasta encontrar el primer piso del cual se quiebra el ladrillo. Si tenemos 2 ladrillos, debemos tratar de balancear el trabajo de ambos ladrillos. Para esto, podemos dividir el edificio en p partes, cada parte con n/p pisos. Comenzamos tirando el primer ladrillo desde el piso n/p, luego $2n/p$, $3n/p$, hasta que se quiebra el ladrillo. En ese momento, usamos el segundo ladrillo en los n/p pisos que quedan entre el piso en que se quebró el primer ladrillo y el anterior en que no se quebró (ver figura).



En el peor caso, el primer ladrillo se rompe en el último piso, luego de ser tirado p veces. De igual modo, en el peor caso, el segundo se romperá en el último de los n/p pisos que se están verificando. En total, $n/p + p$ pruebas. El valor mínimo de esta expresión se obtiene cuando $n/p = p$, lo que implica $p = \sqrt{n}$. Luego el número total de pruebas es $2\sqrt{n}$. Si tenemos más ladrillos, podemos usar esta misma idea en forma recursiva. Es decir, cuando nos quedan n/p pisos, dividimos en p partes y usamos la misma idea, hasta que sólo nos queda un ladrillo, el cual deberá ser usado en forma secuencial. Si tenemos k ladrillos, en el peor caso cada uno de los $k-1$ ladrillos será usado p veces. El último será usado en un segmento con n/p^{k-1} pisos (porque dividimos $k-1$ veces en p partes). El total es $(k-1)p + n/p^{k-1}$. Nuevamente el mínimo se obtiene cuando cada ladrillo realiza el mismo número de pruebas. Esto implica que $p = n/p^{k-1}$, o $p = \sqrt[k]{n}$ (raíz k-ésima de n). En total

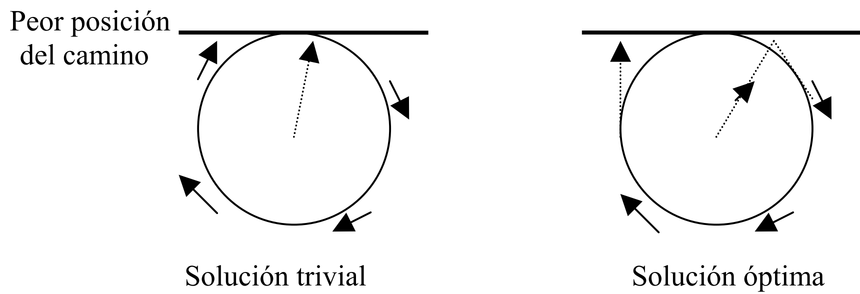
$k\sqrt[k]{n}$ pruebas. El lector puede comprobar que cuando k es logarítmico, esta expresión coincide con búsqueda binaria. Esta técnica se conoce como *división balanceada del trabajo.*

Búsqueda sin información

En la realidad no sólo tenemos recursos finitos, sino que también desconocemos parcial o totalmente el contexto del problema. Por ejemplo, supongamos que estamos perdidos en el desierto y de pronto encontramos una carretera completamente recta. ¿Cómo encontramos el lugar habitado más cercano? El objetivo será minimizar la distancia que tenemos que caminar y lo que no conocemos es la dirección del lugar habitado. Para abstraer este problema, podemos pensar que queremos encontrar un número entero desconocido que puede ser negativo o positivo comenzando en el origen (el cero) y usando como función de costo la distancia que tenemos que recorrer. Para encontrar ese número, debemos progresar tanto en los números positivos como en los negativos, debiendo volver cada vez sobre nuestros pasos para explorar la dirección contraria. Este es otro ejemplo de algoritmo en línea, donde a medida que vamos avanzando, obtenemos información. Para comparar algoritmos de este tipo se usa una medida llamada *competitividad* que se define como el costo del algoritmo sobre el costo óptimo que considera que toda la información del problema es conocida desde el principio. En nuestro ejemplo, la competitividad sería la distancia recorrida con respecto a la distancia directa desde el origen en donde se encontraba el número. Aunque la secuencia óptima de pasos tiene una forma muy simple, demostrar que minimiza la competitividad no es trivial. Primero avanzamos 1 a la derecha y volvemos al punto de partida, luego 2 a la izquierda y luego al punto de partida, después 4 a la derecha, etc. En otras palabras, potencias de 2 a partir del origen, alternando la dirección. El peor caso se obtiene cuando llegamos a un número antes del buscado, cambiamos de dirección alejándonos casi dos veces más lejos hacia el otro lado, para luego encontrar el número a la vuelta (ver figura). Un simple cálculo muestra que la competitividad es 9. Esto significa que debemos caminar 9 veces más que la distancia real por no saber la dirección.

Este problema se puede extender a dos o más dimensiones. Como en el caso de un plano en que podríamos querer encontrar una recta (un camino en un desierto). Si la distancia es conocida, el problema es más simple, pero no trivial. En este caso la solución consiste en encontrar una tangente cualquiera a un círculo que tiene por radio la distancia dada. Así habrá de hacerse si queremos encontrar un camino que sabemos está a un kilómetro en un bosque muy denso o en la selva. Es posible demostrar que la competitividad es aproximadamente 6.39. La solución inmediata es caminar un kilómetro en cualquier dirección y luego seguir un círculo imaginario de ese radio hasta encontrar el camino, que en el peor de los casos se encontrará casi al final del círculo. La distancia recorrida será un kilómetro más el perímetro del círculo dando un total aproximado de 7.28. Para lograr este resultado debemos caminar fuera del círculo, aunque parezca poco intuitivo, pues así cubrimos ambos lados del círculo imaginario. Luego volvemos al círculo usando una tangente al mismo cuando el ángulo entre ésta y la recta que hemos escogido es 60°. Continuamos por el perímetro del círculo hasta tomar una tangente perpendicular a la peor ubicación posible del camino, pues el camino más corto hasta una recta es una perpendicular a ella (ver figura al comienzo de la próxima página).

Cuando no conocemos la distancia, podemos usar una espiral logarítmica para encontrar la recta, logrando una competitividad de 13.81. Esta es una solución que se conjetura como óptima, pero que no ha podido ser demostrada.



Epílogo

En la práctica los problemas reales son más complejos que los que hemos descrito aquí. Sin embargo, tienen un poco de cada caso: recursos acotados, información incompleta, etc. Cabe mencionar que otra técnica útil para resolver un problema es usar el azar. Así, muchos algoritmos funcionan mejor si los datos de entrada tiene una cierta distribución de probabilidades. Luego, el primer paso podría ser aleatorizar los datos de modo que tengan la distribución deseada. Otras veces, podemos tomar decisiones aleatorias. Apliquemos esta idea a búsqueda secuencial. Ya sabemos que si un número esta en un conjunto de números, en el peor caso se deben hacer n comparaciones si hay n números. Sin embargo, si al comienzo de una búsqueda secuencial tiramos una moneda y de acuerdo a su valor buscamos de izquierda a derecha o de derecha a izquierda en los números, el peor caso será distinto, pues ahora tenemos dos algoritmos. Supongamos que el elemento buscado está en la posición j. Con probabilidad ½ lo encontraremos usando j comparaciones. Si no, si buscamos de derecha a izquierda usaremos $n-j+1$ comparaciones. En total: $j/2 + (n-j+1)/2 = (n+1)/2$. Lo que implica que el peor caso promedio entre los dos algoritmos es $(n+1)/2$. En otras palabras, la peor situación se produce cuando el elemento está en el medio y ambos algoritmos necesitan el mismo trabajo.

Existen otras técnicas más avanzadas para resolver problemas, pero hemos presentado las más importantes. Si alguien desea profundizar en el tema, existen muchos libros de texto acerca de algoritmos y estructuras de datos que permiten almacenar y recuperar información. Hemos clasificado los algoritmos sobre la base del método que usan: secuenciales, paralelos, aleatorios, etc. Otros algoritmos no dan siempre respuestas exactas o se equivocan con probabilidad muy baja. Otra forma de clasificar algoritmos es basándose en los problemas que resuelven, como los algoritmos para buscar palabras, numéricos, geométricos, asociados a la biología (por ejemplo encontrar dos secuencias de ADN similares), etc. En el futuro se inventarán nuevos tipos de algoritmos y aparecerán nuevas aplicaciones. También se pueden clasificar los problemas. Todos los problemas mencionados son resueltos en tiempo polinomial. Hay una clase de problemas más difíciles para los cuales no se conocen soluciones polinomiales. Un ejemplo clásico es el problema del vendedor viajero, que consiste en encontrar un camino que recorre todas las ciudades que debe visitar, volviendo a la inicial, sin pasar más de una vez por la misma ciudad. Finalmente, hay problemas que no tienen solución.

Nuestra intención principal ha sido entregar al lector la riqueza y belleza de las técnicas de resolución de problemas que existen. Muchas veces es más elegante y barato mejorar un algoritmo que comprar un computador más rápido. Los dilemas algorítmicos están presentes en todas las

actividades humanas y resolverlos en forma rápida y eficiente puede ser un desafío entretenido si gustamos de la lógica y las matemáticas.

BIBLIOGRAFIA COMPLEMENTARIA

SEARCHING: AN ALGORITHMIC TOUR. R. Baeza-Yates, en *Encyclopedia of Computer Science and Technology 37*, dirigido por Allen Kent y James G. Williams. Marcel Dekker, Inc., 1997.
ALGORITHMICS: THEORY AND PRACTICE. G. Brassard y P. Bratley. Prentice-Hall, 1988, (segunda edición), 1995.
INTRODUCTION TO ALGORITHMS. T. Cormen, C. Leiserson y R. Rivest, MIT Press, 1991.
HANDBOOK OF ALGORITHMS AND DATA STRUCTURES. G. Gonnet y R. Baeza-Yates. Addison-Wesley, (segunda edición), 1991.
THEORY AND PRACTICE. D. Knuth, en *Theoretical Computer Science*, vol. 90, 1991.
INTRODUCTION TO ALGORITHMS: A CREATIVE APPROACH. U. Manber, Addison-Wesley, 1989.
COMPARED TO WHAT? AN INTRODUCTION TO ANALYSIS OF ALGORITHMS. G. Rawlins, Computer Science Press, 1991.

Anexo: Al-Khorezmi

Muhammad ibn Musa abu Djafar Al-Khorezmi nació alrededor del 780 DC en *Khorezm*, al sur del Mar de Aral (hoy Khiva, Uzbekistán), que había sido conquistado 70 años antes por los árabes. Su nombre ya dice mucho, pues significa "Mohamed, hijo de Moisés, padre de Jafar, el de Khorezm". Su fama debió ser muy grande para que todo el mundo lo conociera por su lugar de origen (en el caso de los griegos se anteponía un nombre, por ejemplo, Tales de Mileto). Hacia el 820, Al-Khorezmi fue llamado a Bagdad por el califa abasida *Al-Mamun*, segundo hijo de Harun ar-Rashid, por todos conocido gracias a las "Mil y una Noches". Al-Mamun continuó el enriquecimiento de la ciencia árabe y de la Academia de Ciencias (llamada la Casa de la Sabiduría) creada por su padre, la cual traería importantes consecuencias en el desarrollo de la ciencia en Europa, principalmente a través de España. Poco se sabe de su vida, pero realizó viajes a Afganistán, el sur de Rusia y Bizancio (hoy Turquía). Falleció en Bagdad hacia el 850 DC.

La mayoría de sus diez obras son conocidas en forma indirecta o por traducciones hechas más tarde al latín. De un par de ellas sólo se conoce el título. Al-Khorezmi fue un recopilador del conocimiento de los griegos y de la India, principalmente matemáticas, pero también astronomía, astrología, geografía e historia. Su trabajo más conocido y usado fueron sus *Tablas Astronómicas,* basadas en la astronomía india. Incluyen algoritmos para calcular fechas y las primeras tablas conocidas de las funciones trigonométricas seno y cotangente. Lo más increíble es que no usó los números negativos (que aún no se conocían), ni el sistema decimal ni fracciones, aunque sí el concepto del cero. Su *Aritmética*, traducida al latín como "Algorithmi de numero Indorum" introduce el sistema numérico de la India (sólo conocido por los árabes unos 50 años antes) y los algoritmos para calcular con él. Finalmente tenemos el *Algebra*, una introducción compacta al cálculo, usando reglas para completar y reducir ecuaciones. Además de sistematizar la resolución de ecuaciones cuadráticas, también trata de geometría, cálculos comerciales y de herencias. Quizás éste es el libro árabe más antiguo conocido y parte de su título "Kitab al-jabr wa'l-muqabala" da origen a la palabra álgebra. Aunque los historiadores no se han puesto de acuerdo en la mejor traducción del título, éste significa "El libro de restaurar e igualar" o "El arte de resolver ecuaciones".

El trabajo de Al-Khorezmi permitió preservar y difundir el conocimiento de indios y griegos (con la notable excepción del trabajo de Diofanto), pilares de nuestra civilización. Rescató de los griegos la rigurosidad y de los indios la simplicidad (sustituye una larga demostración, por un diagrama junto a la palabra "Mira!"). Sus libros son intuitivos y prácticos y su principal contribución fue simplificar las matemáticas a un nivel entendible por no expertos. En particular muestran las ventajas de usar el sistema decimal indio, un atrevimiento para su época, dado lo tradicional de la cultura árabe. La exposición clara de cómo calcular de una manera sistemática a través de algoritmos diseñados para ser usados con algún tipo de dispositivo mecánico similar a un ábaco, más que con lápiz y papel, muestra la intuición y el poder de abstracción de Al-Khorezmi. Hasta se preocupaba de reducir el número de operaciones necesarias en cada cálculo. Por esta razón, aunque no haya sido él el inventor del primer algoritmo, merece que este concepto se encuentre asociado a su nombre. Al-Khorezmi fue sin duda el primer pensador algorítmico.