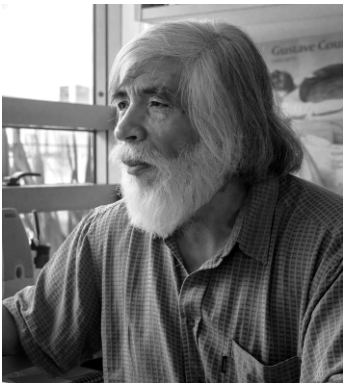
# Una mirada profunda

DOI: 10.29236/sistemas.n161a2



*El poder de los algoritmos, o al menos lo que en el pasado se ha creído, ha debido revaluarse continuamente. Ni la época actual ni las futuras tienen por qué ser excepciones en este sentido, a pesar de la aparente omnipotencia de los previsibles desarrollos.*

Rodrigo Cardoso Rodríguez

En la última mitad del siglo XIX, los matemáticos se esforzaron en definir axiomatizaciones para diferentes teorías, lo que permitiría demostrar con rigurosidad verdades de esas teorías. En el primer cuarto del siglo XX, K. Gödel mostró que esto era parcialmente posible para teorías tan sencillas y útiles como la lógica de primer orden, pero imposible para otras tan prácticas como la aritmética.

En 1938 A. Turing definió una noción de algoritmo que, partiendo de lo probado por Gödel, evidenció propiedades indecidibles, i.e.; no era concebible un método, un algoritmo que las demostrara. Concretamente, Turing mostró la indecidibilidad del problema de la parada; es decir, en un programa de computador es imposible diseñar un algoritmo que permita inferir si al ejecutarse va a detenerse o no, esto último está planteado aquí en términos actuales, considerando que, para la época de Turing, ni siquiera estaba claro qué sería un computador. La idea de computador y de algoritmo planteada por la famosa máquina de Turing es una versión

abstracta de lo que es un computador moderno. Hoy, ante otras plataformas de cálculo más poderosas, el modelo de Turing todavía se puede considerar pertinente y vigente.

¿Qué es un algoritmo? Para un programador es un conjunto de instrucciones que se dan a una máquina para cumplir una tarea. Algunas propiedades: (1) cada instrucción debe ser bien definida y efectivamente realizable; (2) puede tener o no datos de entrada, pero sí datos de salida; (3) debe terminar después de un número finito de pasos. La condición (3) es la que Turing mostró indecidible, en general. Aunque puede flexibilizarse cuando se consideran programas que no deberían terminar, como un sistema operacional, siempre es posible concebir que, aún en estos casos, hay comandos que permiten terminar la ejecución del programa de forma normal y controlada.

Al desarrollarse los primeros computadores se insistió en que cualquier problema decidible, para el que existiera un algoritmo solución, debería poderse implementar en la práctica. Una vez más se descubrió que se estaba pensando con el deseo. Incluso, se mostró que había funciones definibles, pero no calculables. El estado del arte en los años sesenta llegó a ser bastante incierto, porque se cuestionó si las soluciones sí resolvían los problemas correspondientes. Más aun, en el supuesto de que una solución hiciera lo que debería hacer, había que hilar más fino, porque la dificultad en solucionar los problemas depende de aspectos como la cantidad y la calidad de los datos, así como de la plataforma de cómputo en que se implemente la solución.

Para salir del atolladero de saber si un programa hacía lo que se suponía que hiciera, se estableció con claridad el concepto de corrección de un algoritmo. Típicamente se establece un lenguaje lógico que permite especificar el problema, planteando cómo es el estado de las cosas antes de correr un programa y cómo se espera que sea después. Así, una especificación es esencialmente un par de aserciones sobre el mundo del algoritmo: la pre y poscondición. No es la única forma de especificar, pero cualquier otra debería poderse entender así. El problema se resolvió formalmente con teorías en las que trabajaron, especialmente T. Hoare, E. W. Dijkstra y D. Gries. Sin embargo, hoy en día los programadores prácticos difícilmente demuestran la corrección de sus productos, aunque hay que reconocer que hacerlo podría ser más complicado que programar en sí. De todas maneras, cualquier programador serio sabe que debe poder argumentar coherentemente sobre la corrección de sus soluciones, así lo haga de manera informal. Pero se puede decir que este 'hueco' en la praxis de la programación es clave para que existan personajes que programan mucho mejor que otros y,

en consecuencia, deben tener más éxito en su trabajo.

En cuanto al lío de saber si una solución correcta es buena -léase práctica-, se debió establecer una noción de complejidad para los algoritmos. Era necesaria una teoría que fuera, en lo posible, independiente de detalles de implementación técnica que, con los desarrollos del *hardware*, pudieran ser irrelevantes. Entonces se llegó a plantear la complejidad de una solución cuyo uso de recursos, como tiempo y espacio, es dependiente del tamaño de los datos de entrada. Y así se llegó al consenso de que una dependencia polinomial en lo temporal es eficiente, pero una más cara que lo polinomial, por ejemplo, exponencial, es mala. La notación que se usa para esto se tomó prestada del cálculo de funciones de variable real. *Grosso modo*, se habla de una complejidad $O(R(n))$ cuando el uso del recurso se comporta, para $n$ grande como la función $R(n)$.

En su tesis de doctorado en 1971, S. Cook describió formalmente esta teoría de complejidad, pero su contribución más importante fue la definición de la clase de problemas **P**, **NP** y **NP**, completos. Una curiosidad de sus definiciones es que los problemas de decisión -solo hay que contestar "sí" o "no" para solucionarlos- representan el corazón de la dificultad de las soluciones. Y entonces, **P** es la clase de los problemas de decisión solucionables en tiempo polinomial y **NP** es la clase de esos problemas verificables en tiempo polinomial. Los problemas **NP**-completos son una subclase muy especial; si hay una solución polinomial para un problema **NP**-completo, también habrá una para cualquier problema en **NP**. Cook mostró que su teoría valía la pena exhibiendo varios problemas **NP**-completos, comenzando por el problema SAT. Éste consiste en plantear una fórmula de lógica proposicional (solo variables booleanas, sin cuantificadores) y responder si hay una manera de definir sus variables de modo que la fórmula sea verdadera.

Después de Cook, la algorítmica se desarrolló mostrando algoritmos polinomiales mejores -en lo posible, óptimos- y descubriendo problemas importantes que fueran **NP**-completos. Mejorando algoritmos polinomiales hasta encontrar cotas inferiores se establecen límites a la búsqueda de soluciones. Por ejemplo, en el caso de los algoritmos de ordenamiento, usando comparaciones se sabe que, si deben ordenarse $n$ elementos, una solución $O(n \log n)$ es óptima. Por otra parte, tener presente una lista de problemas **NP**-completos en el dominio del discurso de un programador, previene que éste se involucre en buscar soluciones que los mejores no han podido desarrollar; además, es posible dudar que algún día alguien lo haga.

Esto de conocer problemas **NP**-completos para no resbalarse en

tratar de solucionar uno de ellos, cae en el terreno de saber si es cierto o no que **P**=**NP**. Para mostrar la igualdad, bastaría que un problema **NP**-completo tuviera una solución polinomial, lo que conllevaría a que todo problema también la tuviera. Pero, desde que se planteó la duda, nadie ha podido demostrarlo; es quizás el problema abierto más importante de la informática hoy en día, y bien podría ser indecidible, de manera que nunca se pudiera demostrar. La mayoría de los informáticos creen, sin demostrarlo y sin esperar que nadie lo demuestre, que **PNP**. Hay, incluso, enunciados "plausibles" que serían falsos si fuera cierto que **P**=**NP**. Pero, a la fecha, no hay demostraciones de la desigualdad ni de la igualdad. Sí es verdad que muchos de los algoritmos que resultan prácticos tienen complejidades polinomiales, pero con exponentes más bien pequeños. Para ilustrar esta idea, los mejores algoritmos para decidir si un número entero $n$ es primo demoran $O(n^7)$; para $n$ grande, que es cuando se quisiera usar un computador, ese tiempo es todavía demasiado oneroso para ser práctico.

Uno de los papas de la informática, D. Knuth, planteó en 2014 su creencia personal de que SAT debe tener una solución polinomial. Esto, que a muchos parece un disparate, cuestiona en realidad el modelo de complejidad que se ha venido usando. Entre otras cosas, Knuth argumenta que, si hubiera una solución polinomial para SAT, podría ser, $O(n^{356})$: una complejidad supuestamente eficiente, pero decididamente impráctica.

Cuando se cuenta con una máquina rápida y con buena capacidad de memoria, es concebible que la corrección de un algoritmo se pueda establecer ejecutando, literalmente, todos los casos posibles que puedan darse. Esta es la idea básica de la técnica que se conoce como "chequeo de modelos", con técnicas planteadas en los 80s por E. Clarke y A. Emerson y llevadas a la praxis con éxito, en casos en que es posible aprovechar tanto los progresos del *hardware* como el análisis sistemático de los posibles casos de entrada. Una posibilidad menos explorada, pero prometedora es la verificación estadística, en la que se ensayan casos de prueba representativos para certificar la corrección con un grado de certeza deseado.

Los algoritmos de aproximación se contentan con soluciones no necesariamente exactas. Por ejemplo, en lugar de un óptimo, calcular un valor cercano. También podrían considerarse dentro de esta clase los algoritmos probabilísticos, que dan soluciones correctas con una probabilidad de acierto. Por ejemplo, generar números primos grandes, útil en ciertos algoritmos criptográficos, puede hacerse muy eficientemente y con una altísima probabilidad de corrección, sobre todo, si se compara con métodos exactos como los arriba comenta-

dos. Y, siguiendo por este camino, las técnicas de desarrollo de redes neuronales responden a aplicaciones de la estadística que, sin tener que pensar por qué el programa es correcto, se enfocan en enseñar a la red a comportarse de la manera en que se espera que lo haga, en cuanto a la relación de entrada-salida.

Por otra parte, la globalización de la información y la posibilidad de resolver problemas usando sistemas distribuidos, paralelismo y redes han dado lugar a avances muy significativos en el desempeño de soluciones de problemas importantes. Modelos de cómputo como *MapReduce* y sus sucesores dan lugar a algoritmos para el análisis de datos que permiten la búsqueda y el filtrado de manera eficiente, como los empleados en buscadores usuales en internet.

De hecho, la combinación de enfoques tecnológicos puede dar lugar a algoritmos potentes y eficientes para resolver problemas que, incluso hasta hace poco tiempo, se consideraban inabordables. Por ejemplo, hace 50 años se pensaba que una máquina nunca podría jugar ajedrez con la solvencia en que los buenos jugadores lo hacían. Hace ya 22 años que una máquina logró vencer al campeón mundial de entonces, G. Kasparov. Actualmente, AlphaZero, un programa basado en redes neuronales que juega ajedrez y go, es mejor que cualquier jugador humano. Como curiosidad, aprende a jugar desde cero, solo conociendo las reglas, y jugando contra él mismo, en un período de aprendizaje de apenas un par de días. Semejantes resultados hacen que el ajedrez de humanos contra máquinas no tenga sentido, aunque todavía puedan ser interesantes los juegos entre humanos, para los que esta clase de programas resulta de gran valor al analizar partidas. Por supuesto, también son interesantes los enfrentamientos entre máquinas.

En este punto del desarrollo de la informática, se están aplicando con éxito diferentes técnicas que hace unos años se consideraban como cuestiones académicas dentro de la llamada inteligencia artificial. En los años en que emergieron estas técnicas su aplicación era experimental y costosa; pero, al reconocer su aplicabilidad en dominios que reportan grandes beneficios - comercio, publicidad, elecciones populares…- se ha llegado a la formación de grandes multinacionales de la informática (Google, Microsoft, Facebook, …), gigantes tecnológicos que no conocen fronteras ni de países ni de aplicación de sus algoritmos.

Cada día se encuentran noticias sobre desarrollos en computación cuántica. Esta clase de tecnología, todavía muy experimental, podría en el futuro llegar a ser tan corriente como hoy lo son las técnicas de inteligencia artificial de hace treinta años. Si esto sucediera, las nocio-

nes de complejidad algorítmica mencionadas deberían revaluarse. Y el mundo cambiaría en forma sustancial. Basta mencionar un ejemplo, si se pueden factorizar eficientemente números grandes -cosa que se esperaría poder hacer con computadores cuánticos-, los sistemas de seguridad de todo el planeta se verían seriamente afectados, porque la criptografía actual sería fácilmente vulnerable.

¿Cómo puede seguir todo esto? Se puede ser optimista en el uso exitoso de algoritmos en muchas áreas de conocimiento. Pero claro, también resulta de doble filo pensar en los usos malos que ya se descubren, en especial, en lo que respecta a la tecnología aplicada a las redes sociales y sus consecuencias. Se puede pensar en regular estos usos, pero la dificultad evidente es que se estaría hablando sobre cómo regular el poder de quien posee la tecnología o simplemente en un avance que otros no tienen. Lo que sí parece una constante en el tiempo es que no ha funcionado pensar en una algorítmica omnipotente para la tecnología del momento; siempre ha aparecido algo que no se puede hacer. Pero también es cierto que la ciencia y la tecnología se superan continuamente y los límites concebibles se corren hacia adelante, a veces de forma insospechada.

***Rodrigo Cardoso R.*** *Ingeniero de Sistemas y Computación y Matemático de la Universidad de los Andes (1977), Dipl. Informatiker de la Technische Universität München (1983). Trabajó como ingeniero en el sector privado y en el sector público, pero se desempeñó principalmente como Profesor Asociado del Departamento de Ingeniería de Sistemas y Computación de la Universidad de los Andes, en el área de Fundamentos de Informática. Autor de "Verificación y Desarrollo de Programas" (Ecoé-Uniandes, 1993) y diseñador de cursos y textos de matemáticas discretas y análisis de algoritmos para programas de pre y posgrado. Fue representante por Colombia y Presidente del Centro Latinoamericano de Estudios para Informática (CLEI). En la actualidad es Director Regional para Suramérica-Norte de ICPC, la organización de maratones de programación más relevante a nivel global.*