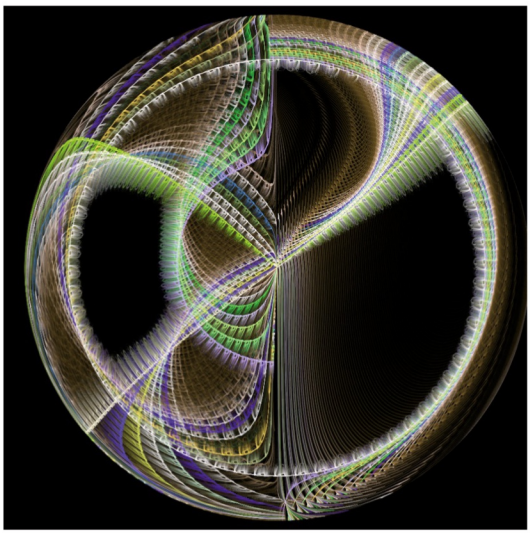
Teoría de la complejidad computacional



Lucien Sina

Teoría de la complejidad computacional

Lucien Sina

1.9.2025

Índice general

Prefacio

La teoría de la complejidad es uno de los ámbitos más fascinantes y, a la vez, más desafiantes de la informática teórica. Se ocupa de una pregunta aparentemente sencilla, pero de gran profundidad y relevancia: ¿qué esfuerzo (en términos de recursos) es necesario para resolver un problema? Mientras que la teoría de la computabilidad investiga qué problemas son algorítmicamente resolubles, la teoría de la complejidad se pregunta por la eficiencia de esas soluciones. De este modo constituye el puente entre los principios fundamentales del cómputo y la viabilidad práctica de los procedimientos algorítmicos.

Este libro nace del deseo de presentar los conceptos, métodos e ideas de la teoría de la complejidad de forma sistemática y a la vez comprensible. Conduce desde los conceptos básicos de clases de complejidad y reducciones hasta temas modernos como la teoría PCP, los sistemas de pruebas interactivas, la complejidad en espacio, la complejidad de circuitos y la complejidad de comunicación. En todo momento se procura no sólo confrontar al lector con definiciones y teoremas, sino

también revelar las intuiciones y las interconexiones subyacentes.

Una preocupación central de este libro es hacer visible la estructura interna de la teoría de la complejidad: ¿Cómo se fundamentan entre sí los distintos conceptos? ¿Cómo se derivan los temas de investigación actuales a partir de preguntas clásicas? ¿Y cómo pueden los límites teóricos proporcionar ideas valiosas para el diseño práctico de algoritmos?

Los capítulos están concebidos de modo que puedan leerse tanto de forma aislada como en conjunto. Cada uno trata un área temática cerrada, desde los fundamentos y la NP-completitud hasta campos de investigación más recientes. A través de ejemplos, esbozos de demostraciones y ejercicios complementarios, el lector se ve animado a trabajar activamente con los conceptos y a comprender su alcance.

Este libro no pretende ser sólo una introducción, sino una obra que armoniza con otros temas fundamentales de la informática. Mis libros complementarios — *Algoritmos y estructuras de datos*, *Teoría de la computabilidad*, *Lógica: Fundamentos, el problema P vs. NP y perspectivas desde la teoría de la información* y *Programación orientada a objetos en Java* — forman conjuntamente con la presente obra una unidad coherente de contenidos. Juntos trazan el arco que va desde la fundamentación teórica y el pensamiento algorítmico hasta la implementación práctica en programas.

La teoría de la complejidad sigue siendo un campo de investigación abierto y vivo. Muchas de sus preguntas centrales —encabezadas por el problema P vs. NP— permanecen sin resolver a día de hoy. Sin embargo, en las últimas décadas ha

quedado patente que los métodos y las ideas de la teoría de la complejidad trascienden la teoría pura: nos ayudan a entender los límites de lo factible, a desarrollar procedimientos eficientes y, aun cuando soluciones completas parezcan inaccesibles, a encontrar compromisos óptimos.

Espero que este libro ofrezca a lectoras y lectores una imagen clara, coherente y estimulante de este campo, que los anime a seguir pensando y que establezca el fundamento teórico sobre el que descansan muchas de las realizaciones prácticas de la informática.

Lucien Sina

Capítulo 1

Introducción

1.1 ¿Qué es la teoría de la complejidad?

La teoría de la complejidad estudia qué recursos mínimos son necesarios para resolver problemas algorítmicos. A diferencia del diseño de algoritmos concretos, que proporciona cotas superiores para el consumo de recursos, la teoría de la complejidad intenta probar cotas inferiores —es decir, afirmaciones sobre cuánto tiempo, memoria u otros recursos escasos *debe* requerir *cualquier* algoritmo que resuelva correctamente un problema dado. Tales afirmaciones permiten distinguir entre lo que es practicable y lo que es, en principio, imposible, evitando así búsquedas inútiles de soluciones inalcanzables.

Un problema algorítmico se formula típicamente de forma general, precisando entradas, salidas deseadas y criterios de calidad. Si la variante formulada en general no admite algoritmos eficientes, se siguen dos estrategias clásicas: o bien

se restringe la clase de entradas o los requisitos (especialización), o bien se aceptan soluciones aproximadas/heurísticas (relajación de requisitos). Ambos caminos pueden conducir a resultados prácticos y a menudo muy útiles, porque ponen al descubierto el núcleo de la dificultad de un problema.

A diferencia del puro diseño de algoritmos, que entrega procedimientos concretos, la teoría de la complejidad ofrece un instrumental formal para clasificar problemas: clases como P (tiempo polinómico), NP (verificabilidad polinómica no determinista) o las clases de espacio agrupan problemas según su consumo asintótico de recursos. Las reducciones entre problemas permiten comparar su dificultad relativa: si el problema A puede reducirse eficientemente al problema B, entonces A no es más difícil que B respecto a los recursos considerados.

El problema más célebre y aún abierto de la disciplina es el problema P vs. NP: ¿es decidible en tiempo polinómico toda propiedad cuya validez puede verificarse en tiempo polinómico? Muchos problemas combinatorios importantes (p. ej., SAT, clique, ciclo hamiltoniano) son NP-completos, es decir, pertenecen a la capa más difícil de NP. Si se encontrara para alguno de estos problemas un algoritmo en tiempo polinómico, se obtendría una cota polinómica para todos los problemas de NP; una prueba de imposibilidad, por el contrario, descartaría toda una clase de algoritmos eficientes.

Además del tiempo de ejecución y la memoria, en consideraciones prácticas y teóricas aparecen otras fuentes de recursos: capacidad de entrada/salida, consumo de energía, ancho de banda de comunicación, tamaño y profundidad de

circuitos, accesos cuánticos, etc. Según la aplicación y el modelo de cálculo, cambian las clases relevantes y las técnicas para su análisis. De ahí la diversidad de métodos de demostración: diagonalizaciones, simulaciones entre modelos, teoría de la información, pruebas por reducción, métodos adversarios y —en desarrollos más recientes— métodos procedentes del álgebra y de la estadística.

La teoría de la complejidad y el diseño de algoritmos son complementarios: los algoritmos muestran lo que es alcanzable; las cotas inferiores muestran lo que no lo es. Ambas perspectivas juntas conducen a una comprensión más profunda de los problemas algorítmicos —ya sea mediante el hallazgo de procedimientos eficientes, la detección de obstáculos o el reconocimiento de causas estructurales de dureza.

1.2 Problemas algorítmicos

Definición 1. *Un **problema algorítmico** es un conjunto de entradas admisibles (instancias), que se representan como palabras finitas sobre un alfabeto finito Σ, junto con una asignación que a cada entrada admisible $x \in \Sigma^*$ asocia un conjunto no vacío de salidas correctas $S(x)$ (respuestas, certificados, valores). Las salidas también son palabras finitas sobre un alfabeto finito (posiblemente sobre un alfabeto distinto). A una entrada concreta la llamamos también una* instancia *del problema; el problema en sí es el conjunto de todas las instancias junto con la especificación de las salidas admisibles.*

Mediante esta formalización adaptamos los problemas a

las capacidades de procesamiento de los ordenadores: entradas y salidas se codifican como palabras finitas en bits o caracteres, de modo que la cuestión de la calculabilidad y del consumo de recursos puede formularse con precisión.

Se distinguen varios tipos de problemas según la naturaleza de la salida requerida:

- **Problemas de decisión:** La salida es una de dos respuestas (p. ej., `Sí`/`No`). Los problemas de decisión suelen formularse como lenguajes $L \subseteq \Sigma^*$: una instancia x pertenece a L si y solo si la respuesta correcta es `Sí`.
- **Problemas de búsqueda:** Basta encontrar, para una instancia x, algún $y \in S(x)$ (p. ej., una prueba o un certificado).
- **Problemas de optimización:** A cada instancia le corresponde un conjunto de soluciones admisibles con una función de valoración; se busca una solución de calidad máxima (o mínima) (p. ej., el ciclo hamiltoniano de longitud mínima).
- **Problemas de evaluación:** En lugar de una solución concreta interesa solo el valor asociado (p. ej., la longitud del camino más corto entre dos vértices).

Con frecuencia es posible transformar problemas entre sí: a partir de un problema de optimización se puede obtener un problema de decisión parametrizando un umbral (p. ej., «¿Existe un ciclo de longitud $\leq k$?»). A la inversa, un problema de decisión puede verse como un caso particular de un problema de búsqueda o de evaluación.

Para problemas resolubles exactamente, todas las salidas correctas se consideran equivalentes; en problemas más difíciles se introduce con frecuencia una métrica de calidad y se permite la aproximación. Una formulación habitual de la calidad de aproximación es un factor $\alpha \geq 1$ (para problemas de minimización, una solución es α-aproximada si su valor es como máximo α veces el valor óptimo). Estas relajaciones de la exigencia de optimalidad son importantes tanto en la práctica como en la teoría.

Un problema algorítmico se considera *resuelto* si existe un mecanismo efectivo y bien definido (algoritmo, máquina de Turing, etc.) que para cada instancia admisible produce una salida correcta de $S(x)$. En teoría de la complejidad además nos interesa con qué coste —por ejemplo en tiempo, espacio, pasos de comunicación, energía o profundidad de circuito— se logra esto.

Finalmente, la representación de las entradas es importante: diferentes codificaciones de la misma entrada matemática pueden dar lugar a problemas algorítmicos distintos. En la práctica se consideran codificaciones equivalentes si pueden transformarse eficientemente entre sí (p. ej., en tiempo polinómico). Esta idea de codificación eficiente permite comparar la dificultad de problemas sin depender de detalles arbitrarios de representación.

Ejemplos: Ejemplos típicos que ilustran las distinciones anteriores:

- **SAT** (decisión): ¿Existe una asignación de verdad que satisfaga una fórmula booleana dada?

- **Camino más corto** (evaluativo/optimizador): Determinar la longitud o el recorrido de mínima distancia entre dos nodos en un grafo ponderado.
- **TSP** (optimizador / relevante para aproximación): Encontrar el ciclo hamiltoniano de longitud mínima en un grafo completo ponderado (problema del viajante).

La formulación precisa de los problemas algorítmicos y sus codificaciones constituye la base para todas las definiciones posteriores de medidas y clases de complejidad, que se tratarán en el capítulo siguiente.

1.3 Algoritmo

¿Qué es un algoritmo?

Definición 2. *Un **algoritmo** es una regla bien definida y unívoca que describe, para cada entrada admisible del problema algorítmico considerado, una sucesión finita y bien ordenada de pasos de cálculo que —en caso de terminar— produce una salida correcta.*

Para la teoría de la complejidad son relevantes, además del concepto de "algoritmo", distintos modelos de cálculo y modos de ejecución. A continuación damos definiciones concisas de los tipos habituales:

Definición 3. *Un algoritmo se llama **determinista** si, en cada configuración (compuesta por la entrada, el contenido de trabajo y el estado), el siguiente paso de cálculo queda*

determinado de forma única. Los algoritmos deterministas corresponden al flujo de ejecución intuitivo en el que cada decisión viene determinada por la información disponible hasta ese momento.

Definición 4. *Un algoritmo se llama* ***no determinista*** *(en el sentido teórico) si en ciertos puntos permite varios pasos sucesivos posibles y una entrada se considera* aceptada *tan pronto como* al menos una *de las ramas de ejecución conduce a una configuración final aceptante. Formalmente, los algoritmos no deterministas se modelan, por ejemplo, mediante máquinas de Turing no deterministas que «adivinan» simultáneamente todas las posibilidades (o, de modo equivalente, mediante la existencia de un certificado adecuado).*

Definición 5. *Un algoritmo es* ***aleatorizado*** *(o* ***probabilístico****) si su ejecución dispone además de bits aleatorios y decisiones individuales durante la ejecución dependen del resultado de esos bits aleatorios. Los algoritmos aleatorizados se clasifican según su comportamiento de error (p. ej., Monte-Carlo con tasa de error máxima* $< \frac{1}{2}$ *o Las-Vegas, que siempre devuelven un resultado correcto pero con tiempo de ejecución aleatorio).*

Un punto importante: no determinismo y aleatorización son conceptos distintos. El no determinismo es un modelo teórico que exige la existencia de una rama de elección "correcta" (existencia de solución), mientras que la aleatorización permite decisiones verdaderamente aleatorias con errores cuantificables o valores esperados para tiempo de ejecución o salida. No es correcto considerar el no determinismo de

forma general como un caso particular de la aleatorización: la relación exacta entre las clases de complejidad correspondientes (por ejemplo P, NP, RP, BPP, ZPP) constituye una de las cuestiones centrales de la teoría y sigue siendo en parte abierta.

Ejemplos:

- Un algoritmo **determinista**: Mergesort — para cada entrada el flujo es único y devuelve una secuencia ordenada en tiempo $O(n \log n)$.

- Un procedimiento de decisión **no determinista** para SAT: «Adivina» una asignación de las variables y verifica de forma determinista que la fórmula queda satisfecha. Formalmente, esto muestra que SAT pertenece a NP.

- Un algoritmo **aleatorizado**: Quicksort aleatorizado (tiempo esperado $O(n \log n)$); Miller–Rabin como test Monte-Carlo para la compositicidad (baja tasa de error, ejecución muy rápida).

En teoría de la complejidad estos conceptos formalizan diferentes modelos de cómputo y conducen a distintas clases, que en lo sucesivo se introducirán y compararán de forma sistemática.

1.4 Tiempo de ejecución de un algoritmo

El tiempo de ejecución de un algoritmo es una medida central de su eficiencia. Una definición ingenua —el tiempo absoluto de cálculo medido en segundos en un ordenador concreto— no es adecuada para comparaciones teóricas, porque depende de muchos factores externos (hardware, lenguaje de programación, detalles de implementación, sistema operativo, optimizaciones del compilador, etc.). El objetivo de una consideración teóricamente limpia es elegir una medida de tiempo que dependa únicamente del propio algoritmo y de su entrada, y que permita comparar de forma razonable distintos modelos de cálculo pertinentes.

Para ello se siguen dos pasos fundamentales:

1. Elegir un modelo de cálculo abstracto que fije las operaciones elementales y sus costes.

2. Formular una medida asintótica del tiempo de ejecución que describa la dependencia de los recursos requeridos respecto a la longitud de la entrada.

1.4.1 Operaciones elementales y medidas de coste

Usualmente medimos el tiempo contando el número de operaciones elementales, donde por operación elemental se entiende aquella que, en un modelo de cálculo realista, se considera de coste «constante»: operaciones aritméticas básicas en palabras de longitud fija, leer/escribir una celda de memoria, transiciones de estado, desplazamiento del cabezal de lectura, etc. Dos medidas de coste comunes son:

Modelo de coste unitario (unit cost): Cada operación elemental cuesta 1. Esto es sencillo de manejar, pero puede dar estimaciones irreales para operaciones sobre números muy grandes (p. ej., la suma de enteros arbitrariamente grandes se cuenta como coste constante).

Medida de coste logarítmica (bit cost): El coste de una operación depende proporcionalmente de la longitud en bits de los operandos; el modelo contabiliza, por así decirlo, operaciones sobre bits. Esta medida refleja mejor la dificultad real de operar con números grandes.

Para muchas afirmaciones teóricas (en particular en el ámbito de clases de complejidad como P y NP) es conveniente elegir una medida de coste robusta frente a la elección de un modelo concreto «razonable». Aquí, robustez significa que dos modelos sensatos difieren en sus tiempos de ejecución a lo sumo por un factor polinómico: la equivalencia polinómica es el criterio de equivalencia habitual en teoría de la complejidad.

1.4.2 Máquina de Turing: modelo de cálculo formal

El modelo formal que más se aproxima al concepto intuitivo de algoritmo y que se ha impuesto como estándar en teoría de la complejidad es la máquina de Turing. A continuación damos la definición usual (determinista) en forma comprimida.

Definición 6 (Máquina de Turing determinista)**.** *Una máquina*

de Turing (determinista) es un tupla

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej}),$$

donde Q es un conjunto finito de estados, Σ es el alfabeto de entrada (sin el símbolo en blanco $\sqcup$), Γ es el alfabeto de trabajo con $\Sigma \subseteq \Gamma$, $q_0 \in Q$ es el estado inicial y $q_{acc}, q_{rej} \in Q$ son los estados finales de aceptación y de rechazo, respectivamente. La función de transición

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

determina en cada paso el siguiente estado, el símbolo a escribir y el movimiento del cabezal (izquierda, derecha, quieto).

Una configuración de una máquina de Turing describe la máquina actual, el contenido de la cinta y la posición del cabezal. Un paso corresponde a la aplicación de δ. El *tiempo de ejecución* de M en la entrada x es el número de pasos hasta la primera llegada a un estado final (si M termina). El tiempo de ejecución en el peor caso de una máquina para entradas de longitud n se denota por $T_M(n) = \max\{\text{ pasos}(M, x) \mid |x| = n\}$.

En la práctica se suelen usar máquinas de Turing de múltiples cintas (varias cintas de trabajo), ya que modelan muchos algoritmos de forma más natural; no obstante, las distintas variantes son polinómicamente equivalentes (las máquinas de múltiples cintas simulan a las de una sola cinta con una sobrecarga cuadrática o, más generalmente, polinómica, y viceversa).

1.4.3 Robustez polinómica y la tesis de Church ampliada

La (ampliada) tesis de Church–Turing en su forma polinómica (frecuentemente denominada *polynomial Church–Turing thesis* o *extended Church–Turing thesis*) afirma esencialmente lo siguiente:

Afirmación (simulación polinómica). *Todo modelo de cálculo "razonable" R (p. ej., máquinas de acceso aleatorio —RAM—, máquinas de registros, modelos de hardware realistas que no admitan operaciones mágicas y no acotadas) puede ser simulado por una máquina de Turing de modo que una ejecución en R que requiera t pasos elementales (bajo una medida de coste adecuada, p. ej. la medida logarítmica / por bits) se pueda realizar en a lo sumo $p(t)$ pasos en una (multi-cinta) máquina de Turing, donde p es un polinomio que depende del método de simulación elegido.*

Esta robustez polinómica justifica formular clases de complejidad (p. ej. P, NP) y afirmaciones asintóticas de tiempo sobre el modelo de Turing: las afirmaciones son entonces independientes del modelo salvo por factores polinómicos, y por tanto estables para la mayoría de las cuestiones teóricas.

Quedamos constancia

Sea R un modelo de cálculo realista y supóngase que el tiempo de ejecución de un algoritmo en R bajo la medida logarítmica de costes viene dado por $t(n)$ para entradas de longitud n. Entonces existe una (multi-cinta) máquina de Turing M y

un polinomio p tales que

$$T_M(n) \leq p(t(n))$$

para n suficientemente grande. De este modo, la complejidad temporal es, en sentido polinómico, independiente del modelo: un pilar central del análisis teórico de la complejidad.

Observaciones

- La forma exacta del polinomio p depende de las operaciones permitidas en R; los modelos que permiten operaciones "injustas" (p. ej., multiplicación directa de enteros arbitrariamente grandes en tiempo constante, o potentes oráculos) no quedan cubiertos por esta afirmación.

- Para cuestiones prácticas finas (p. ej., costes de operaciones con enteros grandes) la medida logarítmica de costes resulta más realista que el modelo de coste unitario.

- La equivalencia polinómica fundamenta el estudio de clases de complejidad bajo la relación de equivalencia "igualdad salvo factores polinómicos", en particular cuando se trata de preguntas como P vs. NP.

1.5 Complejidad de problemas algorítmicos

Sea $t_A(x)$ el tiempo de cálculo de un algoritmo A para una entrada x en el modelo de coste unitario sobre un modelo de referencia (p. ej., una máquina de Turing). Para hacer comparables los tiempos de distintos algoritmos que resuelven el mismo problema algorítmico son necesarias ciertas precauciones; una regla ingenua de comparación punto por punto

$$A \text{ es al menos tan rápido como } B \quad \text{si} \quad t_A(x) \leq t_B(x)\ \forall x$$

conduce a varias dificultades prácticas y teóricas:

1. Los tiempos absolutos dependen del modelo de referencia elegido (máquina de Turing de una cinta frente a multi-cinta, modelo RAM, etc.).

2. Exigir considerar todas las entradas x suele ser demasiado estricto y poco manejable para afirmaciones asintóticas.

3. Para entradas pequeñas un algoritmo "más simple" puede ser más rápido, mientras que algoritmos especializados muestran sus ventajas asintóticas solo para entradas suficientemente grandes.

Para superar estos problemas se emplean en teoría de la complejidad los siguientes conceptos estándar.

1.5.1 Tiempo de ejecución en el peor caso

Para una máquina A se define la función de tiempo (en el peor caso)

$$t_A(n) := \sup\{\, t_A(x) \mid |x| \leq n \,\},$$

o, frecuentemente equivalente,

$$t_A(n) = \max\{\, t_A(x) \mid |x| = n \,\},$$

siempre que el máximo exista. Para la mayoría de los modelos razonables $t_A(n)$ es monótonamente creciente en n. La comparación de algoritmos se hace entonces en forma asintótica: A es *asintóticamente al menos tan rápido* como B si

$$t_A(n) = O\big(t_B(n)\big).$$

1.5.2 Caso promedio y distribuciones

Si existen «instancias atípicas» que dominan el tiempo en el peor caso, puede tener sentido considerar una medida de probabilidad ν_n sobre las entradas de longitud n. El tiempo medio de A respecto de ν viene dado por

$$t_A^{\nu}(n) := \sum_{|x|=n} \nu_n(x)\, t_A(x),$$

donde ν_n es para cada n una distribución de probabilidad sobre $\{x \mid |x| = n\}$. Los análisis en caso promedio son tan informativos como lo sean las suposiciones sobre la distribución de las entradas.

Medidas asintóticas de complejidad

Las notaciones asintóticas habituales $O(\cdot)$, $\Omega(\cdot)$ y $\Theta(\cdot)$ condensan el orden de crecimiento de las funciones de tiempo y permiten comparaciones robustas frente al modelo salvo factores polinómicos (véase la formulación polinómica de la tesis de Church–Turing).

Definición 7. *Un algoritmo A resuelve un problema en tiempo $O(f(n))$ si $t_A(n) = O(f(n))$. Se dice que la **complejidad (temporal) algorítmica** del problema es $f(n)$ si*

1. *existe un algoritmo A con $t_A(n) = O(f(n))$ (cota superior), y*
2. *toda máquina que resuelva el problema requiere al menos un tiempo $\Omega(f(n))$ (cota inferior).*

Si existe tanto una cota superior como una cota inferior con el mismo comportamiento asintótico $f(n)$, se escribe la complejidad como $\Theta(f(n))$ y con ello queda determinada la clase de crecimiento temporal del problema.

1.5.3 Independencia del modelo

La formulación polinómica de la tesis de Church–Turing afirma que para modelos de cálculo «razonables» R_1, R_2 existe un polinomio p tal que una ejecución en R_1 con t pasos elementales puede simularse en R_2 en a lo sumo $p(t)$ pasos. Por ello, las afirmaciones sobre complejidad temporal son, en sentido polinómico, independientes del modelo: un pilar de la adopción de las máquinas de Turing como modelo estándar en teoría de la complejidad.

Observaciones

- Para análisis más finos (p. ej., costes de operaciones con enteros grandes) la medida de coste elegida (coste unitario frente a coste por bits) es determinante.
- Además del tiempo, se consideran por supuesto otros recursos (espacio, comunicación, bits aleatorios, profundidad de circuitos), definiéndose para cada recurso funciones análogas en el peor caso y en el caso promedio.
- Las cotas inferiores son con frecuencia difíciles de demostrar y constituyen el núcleo de muchos resultados en teoría de la complejidad.

Capítulo 2

Clases de complejidad

2.1 La clase de complejidad P

Los tiempos de ejecución polinómicos ocupan un lugar especial en la teoría de la complejidad. Según la formulación (polinómica) de la tesis de Church–Turing, diversos modelos de cómputo «razonables» son polinómicamente equivalentes, de modo que las funciones de tiempo que difieren solo por factores polinómicos se consideran equivalentes para la mayoría de las cuestiones teóricas. Por ello, los tiempos polinómicos se utilizan como una formalización robusta e independiente del modelo del concepto de *computabilidad eficiente*.

Otra razón práctica de la importancia de los tiempos polinómicos es que muchos algoritmos deben leer toda la entrada; esto genera a menudo un límite inferior lineal $\Omega(n)$ para el tiempo de ejecución, y los tiempos polinómicos son, en relación con la longitud de la entrada, relativamente moderados.