LA ESTRUCTURA MATEMÁTICA DE LOS ALGORITMOS

Historia, recursividad, computabilidad y complejidad

Joel Ayala de la Vega
Hipólito Gómez Ayala
Irene Aguilar Juárez

Sello Editorial
REDEM
RED EDUCATIVA MUNDIAL

Joel Ayala de la Vega

Hipólito Gómez Ayala

Irene Aguilar Juárez

LA ESTRUCTURA MATEMÁTICA DE LOS ALGORITMOS

Historia, recursividad, computabilidad y complejidad

Editorial REDEM: Red Educativa Mundial

LA ESTRUCTURA MATEMÁTICA DE LOS ALGORITMOS
Historia, recursividad, computabilidad y complejidad

Para la presente edición:
Editado por Grupo MDM Corp S.A.C.
Para su sello editorial REDEM: Red Educativa Mundial ©
Av. Costanera 2438 Torre “C” Oficina 203 San Miguel, Lima, Perú.
www.redem.org

Primera edición digital, julio del 2026

ISBN: 978-612-99446-5-4

Depósito legal N° 2026-07670

Publicación E-book
Editado y distribuido por REDEM
Libro electrónico disponible en https://editorial.redem.org

Contenido

Prólogo

La informática suele presentarse como un conjunto de técnicas orientadas a construir programas, diseñar estructuras de datos y optimizar implementaciones. Sin embargo, mucho antes de que existieran los ordenadores, ya se había formulado la pregunta que haría posible todo ese desarrollo posterior: ¿qué significa realmente calcular?

Una computadora es, esencialmente, una máquina que opera bajo reglas formales estrictas. Estas directrices no emanan de los componentes físicos, sino de abstracciones lógico-matemáticas que el pensamiento humano ha estructurado y traducido en modelos computables. Por ello, la pregunta fundacional de esta disciplina no nace en la ingeniería, sino en la confluencia entre la filosofía y la matemática. Interroga la naturaleza misma del procedimiento, de la regla y del método, planteando, en el fondo, un problema filosófico sobre la esencia del cálculo y su formalización.

A comienzos del siglo XX, un grupo de lógicos y matemáticos comprendió que antes de construir dispositivos físicos era necesario esclarecer esta cuestión con rigor. Era imprescindible definir qué problemas podían resolverse mediante un procedimiento mecánico —entendido como una secuencia finita de pasos regidos por reglas precisas— y cuáles escapaban a esa posibilidad. De esta búsqueda no surgieron inicialmente máquinas, sino conceptos. Y fueron estos conceptos los que, más tarde, hicieron posibles las máquinas.

Este libro se sitúa en ese punto de origen. No parte de la programación como técnica, sino del fundamento formal que permite comprender qué es un algoritmo, cómo puede definirse con precisión y cuáles son sus límites intrínsecos. Las ideas que aquí se exponen no constituyen simplemente un episodio en la historia de la lógica matemática: representan el momento en que la noción intuitiva de cálculo se transformó en un objeto formal de estudio.

En el corazón de esta transformación hallamos la recursividad. La recursividad no es solo un mecanismo para definir funciones o estructurar programas; es un modo de pensamiento que se refleja a sí mismo, que se apoya en lo ya establecido para construir lo nuevo y que es capaz de cuestionar sus propios fundamentos desde dentro de su propia arquitectura. Puede interpretarse como una analogía formal de la autorreferencia presente en el pensamiento

humano: la capacidad de reflexionar sobre los propios procesos y operar sobre ellos simbólicamente. Cuando una función se define en términos de sí misma, cuando un proceso examina su propia traza o cuando un sistema formal interroga sus límites mediante la autorreferencia, asistimos a una representación formal de procesos autorreferenciales.

El lector encontrará aquí un recorrido que conecta esa raíz filosófica con la manera en que actualmente entendemos los algoritmos. Descubrirá que la noción de procedimiento efectivo, tan familiar hoy, fue durante décadas un problema abierto, y que su solución exigió el concurso de inteligencias extraordinarias que trabajaron no para construir máquinas más rápidas, sino para entender qué significa, en sentido profundo, «seguir una regla».

Al final del camino, le aguarda una paradoja: las mismas herramientas conceptuales que permitieron formalizar procesos lógicos en estructuras computacionales revelan, al mismo tiempo, los límites estructurales de los sistemas formales. La computación nació bajo el signo de una pregunta y maduró bajo el signo de una frontera. Este libro es la bitácora de ese viaje.

Introducción

Este libro no es una introducción a la programación. Tampoco es un manual de algoritmos. Es, ante todo, una **arqueología conceptual**: una excavación en las capas de la práctica técnica hasta dar con los estratos matemáticos y lógicos que la sostienen.

Quien se forma en informática suele recorrer un camino bien definido: aprende un lenguaje, implementa estructuras de datos, adquiere criterios de eficiencia. Este trayecto es operativamente eficaz, pero con frecuencia oculta el subsuelo sobre el que se asienta. El algoritmo se vuelve un instrumento cotidiano, casi invisible, cuyo origen formal permanece en la penumbra. Esta obra propone un movimiento inverso: en lugar de partir de la implementación para luego abstraer, parte de los fundamentos que hicieron posible la noción misma de algoritmo.

Para ello, el texto se organiza como un recorrido progresivo que va desde el origen conceptual del algoritmo hasta el estudio de los límites de la computación y la complejidad de los problemas.

En lugar de presentar directamente técnicas o herramientas, la obra parte de una pregunta fundamental: **¿qué significa realmente calcular?** A partir de ella, se desarrolla un itinerario que atraviesa la historia, la lógica y la matemática hasta consolidar la computación como una ciencia formal.

El Capítulo I reconstruye el proceso histórico mediante el cual la noción intuitiva de procedimiento efectivo se transforma en un objeto matemático. Desde las primeras ideas de sistematización del pensamiento hasta la crisis de fundamentos del siglo XX, se analizan las contribuciones de figuras como Llull, Leibniz, Dedekind y Peano, culminando en Gödel, Church y Turing. En este recorrido emerge una idea central: la computación no surge con las máquinas, sino con la necesidad de delimitar qué puede resolverse mediante reglas finitas y qué no.

El Capítulo II introduce la formalización rigurosa mediante la teoría de las funciones recursivas. Aquí se define con precisión qué significa que una función sea computable, recorriendo la transición desde funciones primitivas recursivas hasta funciones recursivas generales, donde aparece de forma esencial la posibilidad de no terminación.

El Capítulo III analiza el impacto de la computabilidad en la computación, mostrando cómo los resultados teóricos establecen límites fundamentales para los algoritmos y redefinen la naturaleza de los problemas computacionales.

El Capítulo IV presenta las herramientas derivadas de este marco teórico, evidenciando cómo los conceptos formales se traducen en estructuras y prácticas concretas dentro de la informática, estableciendo un puente entre teoría y aplicación.

El Capítulo V aborda los algoritmos como objetos de estudio, proponiendo su clasificación según su estructura y estrategias de diseño. Se sistematiza así el estudio de los procedimientos computacionales.

El Capítulo VI se centra en la complejidad de los algoritmos, introduciendo métodos para analizar su eficiencia y describir su comportamiento en función del tamaño de la entrada.

El Capítulo VII amplía este análisis hacia la complejidad de los problemas, estudiando clases como P y NP y explorando la distinción entre problemas resolubles eficientemente y aquellos que, aunque verificables, podrían no serlo.

El Capítulo VIII cierra el recorrido abordando la frontera entre lo eficiente y lo imposible. Aquí se examinan los límites de la complejidad algorítmica, consolidando una visión en la que no todo problema computacional admite solución práctica, e incluso algunos escapan por completo a cualquier procedimiento efectivo.

Este recorrido no es meramente expositivo. Su propósito es mostrar que el algoritmo no es una herramienta aislada, sino el resultado de un proceso intelectual profundo en el que convergen la lógica, la matemática y la filosofía. Comprender ese proceso es comprender que la computación no solo trata de lo que puede hacerse, sino también —y de manera esencial— de lo que no puede hacerse.

La computación no se define solo por lo que puede hacer. En modelos Turing-completos, la expresividad computacional general implica admitir procesos cuya terminación no puede garantizarse universalmente; comprenderla es comprender sus límites: allí donde el algoritmo encuentra sus límites, emergen preguntas que exceden el alcance de los procedimientos formales.

Este libro no enseña a programar. Enseña a leer la arquitectura invisible que hace posible todo programa.

> «La teoría nos dice que es posible; la práctica nos dice qué es viable.
> La sabiduría está en entender ambas.»

Capítulo1 La chispa de la computación: un viaje por la historia de la teoría de la recursividad

Este capítulo no es un recorrido histórico por mera cultura general. Es aquí donde surge, por primera vez, el marco matemático que permitiría definir con precisión lo que hoy llamamos *algoritmo*. Antes de que existieran los ordenadores y mucho antes de que se escribiera una sola línea de código, ya se estaba gestando la pregunta fundamental que daría origen a toda la computación moderna: qué puede resolverse mediante un procedimiento mecánico y qué escapa a esa posibilidad. La computación nace no cuando aparecen máquinas, sino cuando aparecen límites.

En el corazón de la computación subyace una cuestión tan antigua como la matemática misma: ¿qué significa realmente *calcular*? A comienzos del siglo XX, un grupo de matemáticos y lógicos emprendió una búsqueda intelectual sin precedentes para formalizar esta noción aparentemente sencilla. Su objetivo no era construir máquinas físicas, al menos no de inmediato, sino establecer con rigor cuáles problemas podían resolverse mediante una secuencia finita de pasos bien definidos. La computación nace no cuando aparecen máquinas, sino cuando aparecen límites. De esta empresa surgiría la Teoría de la Recursividad, disciplina que delimitaría las fronteras de lo computable y sentaría las bases lógicas sobre las que se edificaría la era digital.

Este capítulo describe cómo la idea informal de "computación efectiva" se transformó en una teoría rigurosa capaz de caracterizar la naturaleza del cálculo. Se examinan las aportaciones decisivas de figuras como Kurt Gödel, quien reveló los límites de los sistemas formales; Alonzo Church, cuyo cálculo lambda ofreció una definición elegante de función computable; Alan Turing, cuya máquina conceptual se convirtió en el arquetipo de todo ordenador; Rózsa Péter, quien sistematizó el estudio de las funciones recursivas primitivas; y Emil Post, cuyas formulaciones reforzaron la universalidad de estas ideas.

A través de este recorrido, se comprenderá que la teoría de la recursividad no es solo un episodio relevante en la historia de la lógica matemática, sino la verdadera chispa fundacional que hizo posible concebir el algoritmo como objeto formal. Para facilitar su comprensión, el capítulo analiza cuatro ejes de análisis: la etapa del proto-algoritmo, la búsqueda del

algoritmo universal y la crisis de los fundamentos; el nacimiento formal de la teoría de la recursividad y los modelos formales de computación.

1.1 El proto-algoritmo (XVII a. c., fines del siglo XIX)

La época del proto-algoritmo no se limita a un solo siglo, sino que constituye un arco evolutivo inmenso. Es el periodo vital donde el algoritmo existía como un método, receta o procedimiento deductivo puro, mucho antes de que la teoría de la computabilidad otorgara su definición formal y matemática. Durante estos milenios, el desarrollo de la computación estuvo impulsado casi en su totalidad por la abstracción lógica, demostrando que la teoría matemática y la secuenciación de pasos siempre antecedieron a las máquinas físicas diseñadas para ejecutarlos. Para su análisis se tratan las fases: la matemática en la antigüedad, la matemática en la edad media y la era pre-formal.

La antigüedad

El pensamiento algorítmico tiene su origen en las primeras civilizaciones, donde surgieron métodos sistemáticos para resolver problemas prácticos como el cálculo, la medición o la distribución de recursos. Aunque carecían de formalización lógica, estos métodos ya consistían en secuencias finitas de pasos ordenados y reproducibles, rasgo esencial de los algoritmos modernos. En esta fase sobresale personajes como Ahmes (Egipto, c. 1650 a. C.), Euclides de Alejandría (Grecia, c. 300 a. C.) y Eratóstenes de Cirene (Grecia, c. 276–194 a. C.).

Ahmes fue el **e**scriba responsable del *Papiro Matemático Rhind*, uno de los documentos más antiguos de la matemática. En él se registran problemas acompañados de procedimientos paso a paso, incluyen operaciones con fracciones, áreas y volúmenes. Estos métodos muestran una forma temprana de algoritmo: reglas generales aplicables de manera repetida y con finalidad práctica, además de una clara intención pedagógica.

Euclides de Alejandría fue autor de *Elementos*, donde establece la estructura axiomática de la matemática basada en definiciones, postulados y demostraciones. Su aporte clave al pensamiento algorítmico es el algoritmo para el máximo común divisor, un procedimiento general basado en divisiones sucesivas. Además, sus construcciones geométricas siguen

secuencias ordenadas, consolidando el uso de métodos sistemáticos dentro de un marco lógico riguroso.

Eratóstenes de Cirene desarrolló la criba de Eratóstenes, un procedimiento iterativo para identificar números primos mediante eliminación sucesiva de múltiplos. Este método constituye uno de los primeros algoritmos explícitos para un problema general. Asimismo, aplicó procedimientos matemáticos estructurados a la medición de la Tierra, mostrando la potencia del enfoque sistemático en fenómenos naturales.

En la Antigüedad se marca el tránsito desde técnicas empíricas hacia procedimientos generalizables basados en reglas, donde el cálculo deja de ser solo práctica y se convierte en proceso estructurado. Aunque aún no existe formalización, ya están presentes los elementos fundamentales del algoritmo: finitud, orden, repetibilidad y aplicabilidad general.

1.2 La Matemática en la Edad Media (Siglos IX - XV d.C.)

En la edad media el foco lógico se desplaza hacia el mundo islámico y la Europa medieval, donde se acuña el concepto que da nombre a la disciplina. Los matemáticos con mayores aportaciones son Al-Juarismi (Persia, c. 780–850 d. C.), Ramón Llull (1232–1316) , y François Viète (1540–1603).

Al-Juarismi fue el matemático clave de la Casa de la Sabiduría, su gran aporte fue sistematizar métodos para resolver ecuaciones lineales y cuadráticas mediante procedimientos paso a paso, estableciendo un modelo temprano de algoritmo. Impulsó la difusión del sistema numérico indo-arábigo (incluido el cero) y desarrolló técnicas de cálculo aplicadas en aritmética y astronomía. Su obra dio origen al término *algoritmo*, consolidando la idea de cálculo como conjunto de reglas precisas y reproducibles.

Ramón Llull fue un filósofo y teólogo mallorquín, es uno de los primeros pensadores en concebir el razonamiento como un proceso mecánico y sistematizable, lo que lo convierte en un precursor directo de la computación. En su obra *Ars Magna* propuso un sistema basado en la combinación formal de símbolos mediante reglas, orientado a generar conocimiento de manera automática.

Entre sus aportes destacan: la representación simbólica del conocimiento; planteó que las ideas podían codificarse en símbolos manipulables, anticipando la base de la computación simbólica. Planteo el razonamiento como proceso mecánico, su método generaba conclusiones mediante la aplicación de reglas fijas, funcionando como un algoritmo primitivo. Desarrolló la combinatoria sistemática usando permutaciones de conceptos como mecanismo de exploración de soluciones, anticipando técnicas de búsqueda en espacios de estados. También diseño y uso dispositivos lógicos como diagramas y ruedas combinatorias que pueden interpretarse como máquinas conceptuales del cómputo.

Llull introdujo una idea radical "pensar puede reducirse a calcular". Esta visión es fundamental en la historia del cómputo, ya que anticipa la automatización del razonamiento siglos antes de la lógica formal. Su influencia se extiende a Gottfried Wilhelm Leibniz, George Boole y Alan Turing, quienes formalizaron esa intuición. En relación con la inteligencia artificial, el método de Llull puede considerarse un antecedente directo de los sistemas basados en reglas, de la generación automática de conocimiento y de los enfoques de búsqueda combinatoria en la IA.

Aunque su sistema no era computacional en sentido moderno, su núcleo conceptual (la manipulación simbólica mediante reglas) constituye uno de los pilares fundamentales sobre los que se construye la inteligencia artificial contemporánea. François Viète **t**ransformó el álgebra en un lenguaje simbólico manipulable, permitiendo describir cálculos como secuencias de operaciones formales independientes de valores concretos. Al introducir letras para representar variables y sistematizar reglas para operar con ellas, convirtió la resolución de problemas en procesos generales y reproducibles, anticipando la noción de algoritmo. Su obra "Ars analytica" consolidó la idea de que el cálculo puede expresarse como transformaciones simbólicas regidas por reglas, base conceptual de la computación simbólica y de los lenguajes de programación.

La era Pre-formal (Siglos XVII - XIX d.C.)

La culminación del proto-algoritmo. Los matemáticos comienzan a visualizar que la ejecución lógica puede separarse del cerebro humano y delegarse a un sustrato diferente.

René Descartes (1596–1650)

René Descartes sentó bases fundamentales del pensamiento computacional al proponer que los problemas pueden representarse de forma abstracta, descomponerse y resolverse mediante procedimientos ordenados. Su integración del álgebra y la geometría permitió pasar de lo visual a lo simbólico, anticipando cómo la computación transforma problemas en datos manipulables.

Los principales aportes a la algorítmica y al cómputo se concretan en cuatro aportaciones esenciales: la geometría analítica que permitió expresar figuras y problemas geométricos como ecuaciones, es decir, convertir información visual en datos simbólicos procesables, principio clave en computación; el uso de las coordenadas cartesianas las cuales introdujo como una forma estructurada de representar posiciones como pares (x, y), equivalente a una estructura de datos que ahora son la base de algoritmos en gráficos, simulación y análisis numérico.

Descartes también generó un método para dividir complejos en partes simples y resolverlos paso a paso, lo que constituye el núcleo del diseño de algoritmos (ahora se identifica la estrategia como divide y vencerás en la programación estructurada). Finalmente desarrolló un lenguaje simbólico claro y generalizable, permitiendo describir procedimientos como secuencias precisas de operaciones**,** similar a las instrucciones en programación. Descartes no solo aportó herramientas matemáticas, sino una forma de pensar: representar, estructurar y resolver problemas mediante pasos definidos, fundamento directo del cómputo moderno.

Gottfried Wilhelm Leibniz (1646–1716)

Leibniz avanzó significativamente en la formalización del pensamiento lógico y algorítmico. Sus contribuciones incluyen: la máquina de calcular la cual demostró que los procesos aritméticos podían automatizarse mediante instrucciones repetitivas, además propuso un

lenguaje formal universal para expresar cualquier proposición, acompañado de un ”calculus *ratiocinator*” para derivar verdades mecánicamente. También retomó el método de Llull y lo amplió; buscaba una lógica universal que redujera el razonamiento a cálculo; finalmente desarrollo la visión de una ciencia unificada basada en principios matemáticos.

Leibniz sentó las bases conceptuales de la computabilidad y la representación simbólica del conocimiento. Sus ideas anticiparon la noción del algoritmo e influyeron en el desarrollo de la lógica formal y los lenguajes de programación.

George Boole (Inglaterra, 1815–1864)

George Boole fue un matemático y lógico británico cuyas investigaciones sentaron las bases de la lógica algebraica moderna. Su trabajo transformó el estudio del razonamiento lógico al mostrar que las proposiciones podían representarse y manipularse mediante estructuras matemáticas formales. Entre sus contribuciones más importantes destacan: el algebra de la lógica; en su obra *An Investigation of the Laws of Thought* (1854) desarrolló un sistema algebraico para representar proposiciones lógicas mediante variables y operadores.

Él introdujo el uso y formalización de las operaciones equivalentes a las actuales AND, OR y NOT, sus operadores permitieron expresar las relaciones lógicas mediante reglas algebraicas. Representó de forma binaria del razonamiento; mostró que las proposiciones podían representarse mediante valores que hoy se interpretan como 1 (verdadero) y 0 (falso), esto permitió modelar formalmente los procesos de decisión. Formalizó la matemática del pensamiento lógico y defendió que los procesos del razonamiento humano podían analizarse mediante leyes matemáticas, estableciendo así un marco formal para el estudio de la lógica.

El álgebra de Boole constituye uno de los fundamentos teóricos más importantes de la computación digital moderna. Los circuitos lógicos utilizados en computadoras, microprocesadores y sistemas electrónicos están basados directamente en operaciones booleanas. Además, su trabajo influyó profundamente en el desarrollo de la lógica matemática, la teoría de la computación y el diseño de lenguajes de programación. De este modo, la formalización del razonamiento lógico propuesta por Boole representa un paso

crucial en la transición desde el pensamiento matemático clásico hacia la computación moderna.

Richard Dedekind (1831–1916)

Matemático alemán clave en la transición del cálculo aritmético al estudio de las estructuras abstractas. Al formalizar la naturaleza lógica de los números, construyó el andamiaje conceptual sobre el cual se erigiría posteriormente el análisis estricto de los algoritmos y la teoría de la computabilidad.

Las dos grandes aportaciones que realizó a la computación fueron: la formulación de la recursividad y la abstracción de secuencias y datos.

Respecto a la recursividad, en su obra *¿Qué son los números y qué deberían significar?* (1888), formuló el principio de definición recursiva. Al definir matemáticamente el "caso base" ($f(0) = a$) y el "paso recursivo" (($f(n+1) = g(f(n))$), estableció el esqueleto lógico exacto de todos los algoritmos recursivos y de las estructuras de control iterativas (ciclos) de la programación moderna. Por otro lado, su concepción de un conjunto $\mathbb{N}$ equipado con un elemento inicial y una función sucesora inyectiva S representa el modelo teórico primigenio de las estructuras de datos secuenciales y dinámicas, como las listas enlazadas.

Dedekind convirtió la recursión en un mecanismo matemático riguroso. Esta formalización es el núcleo absoluto del diseño de algoritmos, ya que permite modelar axiomáticamente cómo una máquina puede ejecutar instrucciones secuenciales, calculando estados futuros a partir de estados previos.

Giuseppe Peano (1858–1932)

Fue un matemático y lógico italiano pionero en la reducción del razonamiento a un lenguaje estrictamente formal. Su esfuerzo por axiomatizar las matemáticas mediante símbolos y operaciones libres de ambigüedad anticipó de manera directa la sintaxis rígida que hoy exigen los lenguajes de programación. Su dedicación generó dos grandes aportaciones al cómputo. En primer lugar, la axiomatización para la máquina; con la formulación de los *Axiomas de*

Peano (1889), redujo la totalidad de los números naturales a un sistema mecánico de operaciones elementales: un estado inicial (0) y un operador estricto de incremento (la función sucesora S). Esta reducción lógica es conceptualmente idéntica a cómo las Unidades Aritmético Lógicas (ALU) manejan el cómputo a nivel de código máquina.

En segundo lugar, demostró la Correctitud Algorítmica mediante el axioma de inducción (si una propiedad vale para 0 y se preserva bajo sucesión, vale para todos) trascendió la matemática para convertirse en el método estándar en la verificación formal de software. Es la base de los "invariantes de bucle", utilizados para demostrar matemáticamente que un algoritmo cumple su propósito sin errores.

El sistema de Peano creó el entorno formal estricto sobre el cual se edificó la teoría de funciones recursivas computables. Proporcionó el marco lógico indispensable para definir teóricamente qué es un algoritmo y establecer los límites absolutos de lo que una computadora puede llegar a resolver.

1.3 La crisis de fundamentos y la búsqueda de un algoritmo universal

David Hilbert (1862–1943)

Hilbert fue un matemático alemán cuyo ambicioso *Programa Formalista* buscaba mecanizar el razonamiento matemático mediante reglas finitas, precisas y libres de contradicciones. Aunque este ideal de una matemática completamente automatizable fue refutado posteriormente, su búsqueda de rigor sentó las bases de la teoría de la computabilidad y obligó a definir formalmente el concepto de algoritmo**.**

Las principales aportaciones de Hibert son tres: primero el Entscheidungsproblem (Problema de decisión); planteo la necesidad de un procedimiento mecánico capaz de determinar la verdad o falsedad de cualquier enunciado formal. Este problema constituyó la primera formulación conceptual de un algoritmo universal y motivó el desarrollo de modelos teóricos de cómputo.

En segundo lugar, formalizó junto con Wilhelm Ackermann (1928) los sistemas basados en reglas recursivas. La función de Ackermann evidenció los límites de estos sistemas y hoy se utiliza como referencia teórica en el análisis de algoritmos y recursividad. También aporto significativamente a la computación al tratar el tema de la completitud y la consistencia. Buscó sistemas formales que fueran completos, es decir, que fueran capaces de decidir cualquier enunciado y consistentes (sin contradicciones). Estos principios anticipan conceptos clave en computación, como la correctitud de programas y la capacidad expresiva de los lenguajes.

El intento de Hilbert por formalizar completamente las matemáticas impulsó el desarrollo de la computación teórica. Paradójicamente, este esfuerzo permitió a Gödel, Turing y Church demostrar los límites de lo computable. En particular, el estudio del *Entscheidungsproblem* condujo al modelo de la Máquina de Turing, estableciendo qué problemas pueden resolverse mediante algoritmos y cuáles no.

El Círculo de Viena y el Positivismo Lógico (décadas de 1920–1930)

El círculo de Viena impulsó una postura radicalmente anti-metafísica, sosteniendo que solo tienen sentido los enunciados que pueden verificarse empíricamente o formalizarse lógicamente. Todo lo demás —especialmente la metafísica tradicional— fue considerado carente de significado cognitivo.

Aunque David Hilbert no perteneció al Círculo de Viena, su programa formalista fue una influencia decisiva en su pensamiento: la idea de que las matemáticas podían fundamentarse mediante sistemas formales, finitos y consistentes reforzó el ideal del Círculo de reducir el conocimiento a estructuras lógicas rigurosas. En conjunto, esta postura buscaba limpiar el conocimiento de ambigüedades y hacerlo preciso, formal y cercano a lo computable.

Resumiendo, las aportaciones del círculo de Viena son significativas ya que logró la **v**erificación y formalización; propusieron que el conocimiento válido debía poder expresarse mediante reglas lógicas claras, casi como procedimientos algorítmicos, anticipando la idea de que el razonamiento puede mecanizarse. En segundo lugar, intentaron eliminar la metafísica del discurso científico, promoviendo un lenguaje completamente formal. Esto sentó las bases para la manipulación simbólica, esencial en la computación. En tercer lugar, separaron estrictamente sintaxis (estructura) **y** semántica (significado)**,** lo que hoy es fundamental en lenguajes de programación y compiladores. Y, por último, desde este entorno, Gödel demostró que no todo puede formalizarse ni decidirse algorítmicamente, revelando límites fundamentales del enfoque anti metafísico.

El Círculo de Viena marcó el inicio de una visión del conocimiento como procesamiento formal de información, clave para el desarrollo de la computación. Sin embargo, su rechazo total de la metafísica resultó ser excesivo.

Esta limitación se hizo evidente con los resultados de Gödel: al demostrar que existen verdades que no pueden ser capturadas por ningún sistema formal, reveló que el ideal de eliminar completamente la metafísica era insostenible. La lógica formal podía estructurar el conocimiento, pero no agotarlo.

Hoy se reconoce que la metafísica no solo persiste, sino que resulta necesaria incluso en contextos computacionales. En inteligencia artificial, las preguntas sobre significado,

conciencia o representación no pueden reducirse completamente a reglas formales, mostrando una brecha entre manipulación simbólica y comprensión.

En ontologías y modelado de datos, es necesario definir qué "existe" y cómo se estructura la realidad, lo que implica decisiones ontológicas profundamente metafísicas. En los límites de los algoritmos, establecidos por Gödel, Church y Turing, se evidencia que no todo conocimiento es computable ni verificable mecánicamente, introduciendo una distinción entre verdad y demostrabilidad. Respecto a la Naturaleza de la información, ¿Qué es realmente la información? ¿Es una propiedad física, matemática o conceptual? Esta pregunta atraviesa la teoría de la información, la computación cuántica y la filosofía de la mente.

Identidad y abstracción en software: conceptos como objeto, estado, identidad o persistencia en sistemas computacionales implican asumir qué significa que "algo" sea el mismo a través del tiempo, un problema clásico de la metafísica de la identidad. Virtualidad y realidad digital: los entornos virtuales, simulaciones y sistemas digitales plantean preguntas sobre qué es "real" y qué significa existir en un sistema computacional, reactivando debates metafísicos sobre la naturaleza de la realidad.

En resumen, el Círculo de Viena fue crucial al intentar eliminar la metafísica para alcanzar precisión lógica, pero los propios resultados de Gödel mostraron que este proyecto tenía un límite estructural. Lejos de desaparecer, la metafísica reaparece como el marco necesario para comprender aquello que los sistemas formales no pueden capturar.

Nacimiento formal de la teoría de la recursividad (décadas de 1930–1940)

Kurt Gödel (1906–1978)

Kurt Gödel fue un lógico y matemático nacido en Brno (entonces parte del Imperio austrohúngaro). Es una de las figuras más influyentes en la lógica matemática y los fundamentos de la computación.

Aunque no fue miembro formal, mantuvo una relación estrecha con el Círculo de Viena, asistiendo regularmente a sus coloquios. En ese entorno dominado por el ideal de eliminar la metafísica mediante la lógica formal, Gödel desarrolló resultados que no solo transformaron la lógica, sino que mostraron que ese ideal tenía límites insuperables.

Formación académica

Tesis doctoral (1930): Gödel demostró el teorema de completitud de la lógica de primer orden, estableciendo que todo enunciado verdadero en este sistema puede demostrarse formalmente. Este resultado fortalecía el programa lógico del Círculo de Viena, al sugerir que la lógica podía capturar completamente la verdad en su dominio.

Trabajo posdoctoral (1931): Un año después, publicó sus teoremas de incompletitud, mostrando que sistemas más expresivos como la aritmética son necesariamente incompletos.

Teoremas de incompletitud: Los teoremas de Gödel establecen que:

1. **Incompletitud:** en cualquier sistema formal consistente que contenga aritmética, existen proposiciones verdaderas que no pueden demostrarse dentro del sistema.
2. **Indemostrabilidad de la consistencia:** el sistema no puede probar su propia consistencia usando solo sus propias reglas.

Para lograrlo, Gödel desarrolló la **aritmetización de la sintaxis (numeración de Gödel)**, convirtiendo proposiciones y demostraciones en números. Este paso es crucial, ya que transforma el razonamiento en un proceso manipulable, anticipando el núcleo del cómputo moderno.

Impacto en el Círculo de Viena

Los resultados de Gödel tuvieron un efecto decisivo:

- Destruyeron la idea de que todo conocimiento podía formalizarse completamente.
- Mostraron que el ideal anti-metafísico era intrínsecamente limitado.
- Introdujeron la noción de que existen verdades que trascienden cualquier sistema formal.

En otras palabras, el intento de eliminar la metafísica mediante la lógica reveló, paradójicamente, la necesidad de aquello que se buscaba excluir.

Límites de la computación

Los teoremas de Gödel son fundamentales para entender los límites del cómputo:

- No existe un algoritmo universal que pueda decidir todas las verdades matemáticas.
- Existen problemas indecidibles que ninguna máquina puede resolver en todos los casos.
- Inspiraron directamente el trabajo de Alan Turing, quien formalizó estos límites mediante la máquina de Turing.

Gödel no solo mostró los límites de la lógica formal, sino que reveló que toda formalización deja un residuo de verdad que la trasciende: allí donde el algoritmo se detiene, reaparece inevitablemente la dimensión metafísica del conocimiento.

Stephen Cole Kleene (1909–1994)

Kleene tomó las herramientas técnicas utilizadas por Gödel y las convirtió en objeto de estudio sistemático. Formalizó el concepto de **función recursiva** y, con el operador de minimización (μ), amplió el marco hacia funciones cuyo proceso de cálculo no garantiza terminación.

Mientras Gödel usó funciones recursivas para codificar la lógica, Kleene comprendió que el aparato técnico utilizado por Gödel para aritmetizar la lógica no era accidental, sino que constituía precisamente la formalización matemática de la noción de procedimiento efectivo. Lo que en Gödel fue instrumento, en Kleene se volvió teoría: **la teoría formal de la computabilidad**.

De la crisis filosófica a la máquina abstracta

El colapso del ideal universalista del Círculo de Viena y del Programa de Hilbert a manos de Gödel no supuso el fin de la lógica formal, sino su transformación radical. Al demostrar de manera irrefutable que existían límites absolutos en lo que podía deducirse mediante reglas formales, Gödel dejó flotando una pregunta ineludible en el ambiente intelectual de la época: si no todo es demostrable o calculable de forma mecánica, ¿cómo definimos con rigor matemático qué es exactamente un "procedimiento mecánico"? La respuesta a esta interrogante ya no provendría de la filosofía pura, sino del esfuerzo por modelar el cálculo

mismo. Es en este punto de inflexión histórica donde una nueva generación de mentes brillantes recoge el testigo, trasladando el debate desde los límites abstractos de la verdad hacia la arquitectura operativa del cálculo. Con este impulso, entramos en la era de los modelos formales de computación.

1.4 Modelos formales de computación.

Alonzo Church (Estados Unidos, 1903–1995)

Alonzo Church fue un lógico y matemático estadounidense cuya obra desempeñó un papel fundamental en la formalización del concepto de **computabilidad**. Sus investigaciones proporcionaron uno de los primeros marcos matemáticos rigurosos para describir procedimientos efectivos de cálculo. Entre sus contribuciones más importantes destacan:

• **Cálculo lambda (λ-cálculo):** desarrolló un sistema formal basado en funciones y reglas de sustitución que permite describir procesos de cálculo mediante transformaciones simbólicas. Este modelo constituye una de las bases teóricas de la programación funcional.

• **Definición formal de funciones computables:** propuso que las funciones computables podían caracterizarse mediante expresiones del cálculo lambda, proporcionando un marco matemático para estudiar los límites del cálculo efectivo.

• **Problema de la decisión (Entscheidungsproblem):** demostró que no existe un procedimiento general capaz de determinar automáticamente si una proposición lógica es universalmente válida, estableciendo una limitación fundamental en los sistemas formales.

• **Tesis de Church:** formuló la hipótesis de que las funciones efectivamente calculables coinciden con aquellas que pueden definirse mediante el cálculo lambda. Esta idea se consolidó posteriormente como la **tesis de Church-Turing**.

Trascendencia actual

El cálculo lambda desarrollado por Church constituye uno de los pilares teóricos de la informática moderna, especialmente en el diseño de **lenguajes de programación funcionales** y en el estudio formal de los programas como expresiones matemáticas. Además, la equivalencia entre el cálculo lambda y otros modelos de computación, como las **máquinas de Turing**, permitió establecer una caracterización precisa del concepto de computabilidad. La tesis de Church-Turing se considera hoy uno de los principios fundamentales de la teoría de la computación.

Alan Turing (Reino Unido, 1912–1954)

Alan Turing fue un matemático y lógico británico considerado uno de los fundadores de la **teoría de la computación**. Su trabajo proporcionó un marco formal para comprender qué significa que un procedimiento sea computable y cómo pueden describirse los procesos algorítmicos de manera precisa. Entre sus contribuciones más importantes destacan:

• **Máquina de Turing:** introdujo un modelo matemático abstracto capaz de describir cualquier procedimiento computacional mediante un conjunto finito de reglas que operan sobre una cinta de símbolos. Este modelo se convirtió en una de las bases formales del concepto de algoritmo.

• **Concepto de computabilidad:** demostró que muchos problemas matemáticos pueden formularse en términos de procedimientos mecánicos de cálculo, permitiendo caracterizar formalmente qué funciones pueden ser computadas por un algoritmo.

• **Problema de la parada:** probó que no existe un método general que permita determinar si un programa arbitrario terminará o continuará ejecutándose indefinidamente. Este resultado estableció una de las limitaciones fundamentales de la computación.

• **Máquina universal:** mostró que una sola máquina podía simular cualquier otra máquina de Turing mediante la codificación adecuada de sus instrucciones, anticipando el principio de las computadoras de propósito general.

Trascendencia actual

Las ideas de Turing establecieron los fundamentos teóricos de la informática moderna. La **máquina de Turing** se convirtió en uno de los modelos centrales para estudiar la computabilidad y la complejidad de los algoritmos. Además, el concepto de **máquina universal** anticipó la arquitectura de las computadoras modernas, en las cuales un mismo dispositivo puede ejecutar diferentes programas almacenados en memoria. Sus investigaciones también influyeron profundamente en áreas como la criptografía, la inteligencia artificial y la filosofía de la mente, consolidando su legado como una de las figuras más influyentes en la historia del pensamiento computacional.

Así, distintos formalismos revelaron **la misma frontera de lo computable** anticipada por Gödel.

Emil Post (1897–1954)

Resumen de aportes

Post desarrolló de manera independiente:

- **Máquina de Post**, modelo computacional equivalente a la máquina de Turing.
- **Forma normal de Post** para funciones recursivas.
- **Problema de Correspondencia de Post (PCP)**, ejemplo clásico de problema indecidible.

Trascendencia actual

Post reforzó la robustez de la noción de computabilidad al mostrar la equivalencia entre modelos alternativos, y su PCP sigue siendo herramienta estándar en teoría de la computación.

De la incompletitud al problema de la parada

El primer teorema de Gödel mostró límites en la demostrabilidad. Kleene formalizó el lenguaje para describir procedimientos efectivos. Turing tradujo esta estructura al terreno operativo con la **máquina universal** y el **problema de la parada** (1936):
No existe un algoritmo general capaz de determinar si cualquier programa se detendrá o no.
La correspondencia es directa:

- Gödel → límites en sistemas formales
- Kleene → formalización del procedimiento efectivo
- Turing → límites en los algoritmos

Conclusión del Capítulo I

El recorrido histórico nos llevó desde intuiciones filosóficas hasta modelos formales capaces de delimitar con exactitud qué significa calcular. Ese proceso culminó cuando la noción de algoritmo dejó de ser una práctica intuitiva y se convirtió en un objeto matemático dotado de reglas, límites y consistencia interna.
Al llegar a este punto, la historia ya ha cumplido su función: ha revelado el origen y el fundamento de la computabilidad**.** Lo que sigue no pertenece al dominio de la genealogía,

sino al dominio de la formalización. Para avanzar, debemos abandonar la perspectiva histórica y entrar en el terreno donde la computación se expresa con precisión matemática: la teoría de las funciones recursivas**.**

Allí es donde la pregunta que acompañó todo el Capítulo I —¿qué puede calcularse mediante reglas finitas? — recibe su formulación definitiva. Las funciones recursivas permiten describir, con exactitud formal, los mecanismos que subyacen a todo procedimiento efectivo. A través de ellas se trazan las fronteras entre lo computable y lo no computable, entre lo que siempre termina y lo que puede divergir, entre lo que una máquina puede decidir y lo que puede quedar fuera de ciertos procedimientos efectivos.

Con esta transición dejamos atrás la evolución histórica del concepto de algoritmo y entramos en el núcleo matemático de la computación.

Capítulo II Funciones Recursivas: el corazón de la computación formal

El capítulo anterior mostró la evolución histórica del concepto de procedimiento efectivo y culminó en una pregunta fundamental: ¿qué significa, con precisión matemática, que un problema sea resoluble mediante un método mecánico?

El presente capítulo responde a esa interrogante desde el plano formal. Nos adentramos ahora en la teoría de las funciones recursivas, el marco conceptual que permitió convertir la noción intuitiva de algoritmo en objeto matemático riguroso. La transición que aquí se realiza es decisiva: del relato histórico pasamos a la estructura formal; de la genealogía conceptual a la definición precisa. Las funciones recursivas no son una técnica adicional dentro de la teoría de la computación, sino el lenguaje matemático que delimita, con exactitud, el alcance y los límites de lo computable.

2.1 Procedimiento efectivo, algoritmo y la Tesis de Church–Turing

Una función es efectivamente computable si existe un procedimiento finito, preciso y mecánico que produce el resultado correcto para cada entrada válida en un número finito de pasos. Cuando tal método se expresa como una secuencia explícita de instrucciones ejecutables sin interpretación creativa, recibe el nombre de algoritmo.

La idea central es que el proceso debe poder llevarse a cabo de manera puramente mecánica: quien lo ejecuta —sea humano o máquina— no necesita comprender el problema, sino únicamente seguir reglas claramente especificadas. Esta exigencia elimina apelaciones a la intuición, la inspiración o el juicio contextual durante la ejecución.

La noción de computabilidad efectiva permite así distinguir entre:

- Problemas que admiten resolución mediante procedimientos formales.
- Problemas que, aunque bien definidos, no pueden resolverse algorítmicamente.

Esta distinción constituye el eje de la teoría de la computación.

2.1.1 Propiedades estructurales de un procedimiento efectivo

Todo algoritmo debe satisfacer condiciones precisas:

- Determinación: cada paso está completamente especificado; no hay ambigüedad en la transición entre estados.
- Finitud: para toda entrada válida, el proceso termina tras un número finito de pasos.
- Generalidad: el procedimiento funciona para cualquier elemento del dominio definido.
- Ejecutabilidad mecánica: puede realizarse mediante la aplicación ciega de reglas formales.

Estas propiedades no son convenciones pragmáticas, sino exigencias estructurales que permiten formalizar la noción de cálculo.

2.1.2 Formalización y la Tesis de Church–Turing

Durante la década de 1930, la necesidad de precisar matemáticamente el concepto de procedimiento efectivo condujo a varias formalizaciones independientes.

- Kurt Gödel y Stephen Kleene formalizaron la noción de función recursiva.
- Alan Turing propuso el modelo de la máquina de Turing.
- Alonzo Church desarrolló el cálculo lambda.

Lo notable fue que estos modelos, aunque formulados desde perspectivas distintas, resultaron ser equivalentes en poder expresivo. Toda función computable en uno de ellos lo es en los otros.

Esta convergencia dio lugar a la formulación de la Tesis de Church–Turing, que puede enunciarse así:

> *Una función es efectivamente computable si y solo si es computable por una máquina de Turing, o de manera equivalente, expresable en el cálculo lambda o como función recursiva.*

No se trata de un teorema demostrable —pues conecta una noción intuitiva ("procedimiento efectivo") con una formal—, sino de una tesis fundacional ampliamente aceptada. Su importancia radica en que fija el marco dentro del cual puede discutirse la computabilidad.

A partir de este punto, el problema deja de ser histórico o filosófico y se vuelve estrictamente formal: determinar qué funciones pueden construirse mediante los mecanismos recursivos permitidos y cuáles quedan fuera de ese dominio.

2.1.3 Funciones recursivas como modelo canónico

A la luz de la equivalencia establecida por la Tesis de Church–Turing, cualquiera de los modelos formales mencionados podría adoptarse como marco de referencia para estudiar la computabilidad. Sin embargo, las funciones recursivas ocupan un lugar privilegiado por su transparencia estructural y su construcción inductiva a partir de un conjunto mínimo de funciones básicas y reglas de formación.

Su definición no depende de un dispositivo mecánico externo, como la máquina de Turing, ni de un formalismo sintáctico orientado a la abstracción funcional, como el cálculo lambda, sino que se formula directamente en el lenguaje matemático de las funciones sobre los números naturales. Esta característica las convierte en un modelo particularmente adecuado para analizar la arquitectura interna del cálculo efectivo.

Además, la teoría de las funciones recursivas permite organizar las funciones computables en niveles bien definidos —por ejemplo, funciones primitivas recursivas y funciones recursivas generales— mostrando no solo qué es computable, sino cómo se estructura ese poder computacional. En esta jerarquización se revelan diferencias esenciales entre crecimiento, complejidad y decidibilidad.

Así, más que un modelo alternativo entre otros, las funciones recursivas se convierten en el eje conceptual desde el cual examinaremos, en las secciones siguientes, la construcción formal del algoritmo y las fronteras matemáticas de lo computable.

Una vez definida la noción de procedimiento efectivo, el siguiente paso consiste en analizar la estructura matemática de las funciones que pueden ser calculadas mediante tales procedimientos.

2.2 La jerarquía de funciones recursivas: de las primitivas a las generales

Las funciones recursivas se organizan en una jerarquía que refleja tanto su poder expresivo como sus propiedades de terminación. Esta jerarquía permite comprender gradualmente qué significa "ser computable" y dónde se encuentran los límites de la computación.

El punto de partida lo constituyen las funciones recursivas primitivas, que representan el primer dominio formal estable de lo computable.

2.2.1 Funciones recursivas primitivas (FRP)

Las funciones recursivas primitivas constituyen la clase más básica y restringida dentro de las funciones computables sobre los números naturales $\mathbb{N}$. Se definen a partir de un conjunto mínimo de funciones iniciales mediante dos esquemas constructivos que garantizan la terminación.

Funciones iniciales

1. **Función cero**

$$Z(x) = 0$$

2. **Función sucesor**

$$S(x) = x + 1$$

3. **Funciones de proyección**

$$P_i^n(x_1, \dots, x_n) = x_i, 1 \leq i \leq n$$

Estas funciones constituyen el núcleo generador del sistema.

Esquemas de construcción

Las funciones recursivas primitivas se obtienen a partir de las funciones iniciales mediante:

1. Composición

Si

- h es una función de m argumentos, y
- $g_1, \dots, g_m$ son funciones de n argumentos,

entonces la función f definida por

$$f(x_1, \dots, x_n) = h(g_1(x_1, \dots, x_n), \dots, g_m(x_1, \dots, x_n))$$

es recursiva primitiva.

2. Recursión primitiva

Si

- g es una función de n argumentos, y
- h es una función de $n + 2$ argumentos,

entonces la función f definida por:

$$f(0, x_1, \dots, x_n) = g(x_1, \dots, x_n)$$
$$f(y + 1, x_1, \dots, x_n) = h(y, f(y, x_1, \dots, x_n), x_1, \dots, x_n)$$

es recursiva primitiva.

Propiedad fundamental

La característica esencial de las FRP es que **todas son funciones totales**: están definidas para todo número natural y siempre producen un resultado tras un número finito de pasos.

Esto se debe a que la recursión primitiva es un esquema de **recursión acotada**: la profundidad de la definición recursiva está controlada por uno de los argumentos (en este caso, y). No existe posibilidad de iteración indefinida.

Desde el punto de vista estructural, las FRP representan el dominio de lo computable bajo garantía de terminación.

Ejemplos clásicos de FRP

Entre las funciones primitivas recursivas se muestran en la tabla 1:

Tabla 1: Ejemplos clásicos de funciones recursivas primitivas

Función	Notación	Definición recursiva	Comentario estructural
Suma	$\text{add}(x, y)$	$\text{add}(x, 0) = x$ $\text{add}(x, y + 1)$ $= S(\text{add}(x, y))$	Definida por recursión primitiva sobre el segundo argumento.
Multiplicación	$\text{mult}(x, y)$	$\text{mult}(x, 0) = 0$ $\text{mult}(x, y + 1)$ $= \text{add}(x, \text{mult}(x, y))$	Iteración de la suma mediante recursión primitiva.
Factorial	$n!$	$0! = 1$ $(n + 1)! = (n + 1) \cdot n!$	Ejemplo de recursión primitiva anidada.
Exponenciación	x^y	$x^0 = 1$ $x^{y+1} = x \cdot x^y$	Iteración de la multiplicación.
Resta truncada	$x \dot{-} y$	$x \dot{-} y = \max(0, x - y)$	Mantiene la función dentro de $\mathbb{N}$; es total.

Estos ejemplos muestran que las FRP capturan una amplia familia de operaciones aritméticas elementales. Sin embargo, como veremos, su poder expresivo no es ilimitado.

2.2.2 Funciones recursivas totales

El conjunto de las funciones recursivas totales comprende todas aquellas funciones computables que están definidas para cada elemento de $\mathbb{N}$. En términos formales, se trata de funciones totales $f: \mathbb{N}^k \to \mathbb{N}$ cuyo proceso de cálculo siempre termina.

Este conjunto es estrictamente más amplio que el de las funciones recursivas primitivas. Aunque toda función recursiva primitiva es total, no toda función recursiva total es primitiva recursiva.

La diferencia estructural radica en el tipo de mecanismo permitido para su definición. Mientras que la recursión primitiva impone una forma de iteración acotada por uno de los argumentos, existen funciones totales cuya construcción requiere esquemas recursivos más generales, no restringidos por dicha acotación.

Ejemplo paradigmático de función recursiva total: la función de Ackermann

La función de Ackermann $A(m, n)$constituye el ejemplo clásico de una función recursiva total que no pertenece a la clase de las funciones recursivas primitivas.

Su definición involucra una forma de recursión anidada cuya profundidad no está acotada por un parámetro fijo. A diferencia de la recursión primitiva —donde el número de iteraciones está controlado por uno de los argumentos—, la función de Ackermann introduce un crecimiento estructural que desborda ese esquema constructivo.

Este resultado muestra que las funciones recursivas primitivas no agotan el conjunto de las funciones computables totales. Existen funciones que, aun garantizando terminación para toda entrada, requieren mecanismos definicionales más generales.

La función de Ackermann será analizada en la sección 2.6 como ejemplo representativo del tránsito conceptual desde la recursión primitiva hacia formas más amplias de recursividad.

2.2.3 Funciones μ-recursivas (funciones recursivas generales)

Las funciones μ-recursivas —también denominadas funciones recursivas generales— se obtienen al añadir a las funciones recursivas primitivas un tercer esquema de construcción: el operador de minimización (μ).

Este operador introduce un mecanismo de búsqueda no acotada. Precisamente este rasgo amplía el poder expresivo del sistema más allá de la recursión primitiva, aunque al costo de perder la garantía universal de terminación.

Operador de minimización (μ)

Sea $f(x_1, \dots, x_n, y)$una función recursiva. Se define:

$$\mu y \, [\, f(x_1, \dots, x_n, y) = 0 \,]$$

como el menor número natural y tal que

$$f(x_1, \dots, x_n, y) = 0.$$

Si no existe tal y, la función no está definida para esos argumentos.

El operador μ realiza, por tanto, una búsqueda sistemática sin límite superior previamente fijado. Este es el elemento que incrementa la potencia expresiva del sistema, pero también el origen de su posible parcialidad: si la condición nunca se satisface, el proceso no termina.

Alcance estructural

Es fundamental precisar que:

- Las funciones μ-recursivas coinciden exactamente con las funciones recursivas parciales.
- Dentro de este conjunto, las funciones recursivas totales son aquellas para las cuales el proceso de minimización encuentra siempre un valor y, en consecuencia, termina para toda entrada.

De este modo, se obtiene la siguiente jerarquía formal:

Recursivas primitivas $\subset$ **Recursivas totales** $\subset$ **Recursivas parciales** $=$ **μ-recursivas**.

Esta inclusión expresa con claridad el crecimiento progresivo del poder definicional.

Propiedades fundamentales

Las funciones μ-recursivas:

1. Incluyen tanto funciones totales como parciales.
2. Son equivalentes en poder computacional a las máquinas de Turing.
3. Capturan formalmente la noción de computabilidad efectiva en el sentido de la Tesis de Church-Turing.

Ejemplo ilustrativo: división entera mediante minimización

La función de división entera puede definirse mediante minimización como:

$$\mathrm{div}(x, y) = \mu z\,[\, x - y \cdot z = 0\,].$$

Esta función no está definida para $y = 0$, lo que ilustra el carácter parcial que puede surgir cuando se emplea el operador μ.

Nota conceptual: división acotada vs. división μ-recursiva

Existe una diferencia estructural entre esta definición y la llamada división acotada, que puede formularse como función primitiva recursiva (véase sección 2.3.6).

En la división acotada, la búsqueda de z se realiza dentro de un límite previamente fijado. Ese límite garantiza la terminación del proceso y permite su definición dentro del esquema de recursión primitiva.

En cambio, al eliminar dicha cota y permitir una búsqueda sin límite superior, se obtiene una definición μ-recursiva cuya terminación ya no está garantizada.

Esta distinción ejemplifica con claridad el salto expresivo que introduce el operador μ: mayor poder definicional a costa de perder la garantía estructural de terminación.

2.2.4 Funciones recursivas parciales

Las funciones recursivas parciales son aquellas funciones μ-recursivas cuyo proceso de cálculo no está garantizado para todas las entradas. Formalmente, una función parcial

$$f\colon \mathbb{N}^n \to \mathbb{N}$$

es recursiva parcial si existe un procedimiento efectivo tal que, dada una entrada $(x_1, \ldots, x_n)$:

- Si $f(x_1, \ldots, x_n)$está definida, el procedimiento termina y produce el valor correcto.
- Si $f(x_1, \ldots, x_n)$no está definida, el procedimiento puede no terminar.

A diferencia de las funciones recursivas primitivas —que siempre son totales—, las funciones recursivas parciales admiten la posibilidad de no terminación. Por esta razón,

constituyen el modelo formal más general de la computación efectiva y reflejan con mayor fidelidad el comportamiento de los programas reales, donde algunos procesos concluyen y otros pueden entrar en ejecución indefinida.

Ejemplo: la conjetura de Collatz

Un ejemplo clásico que ilustra la complejidad que puede surgir a partir de reglas extremadamente simples es la **conjetura de Collatz**, propuesta por el matemático alemán Lothar Collatz en 1937. El problema se formula mediante un procedimiento iterativo definido sobre los números naturales.

Dado un número natural n, se aplica la siguiente regla:

- Si n es **par**, se reemplaza por $^{n}/_{2}$.
- Si n es **impar**, se reemplaza por $3n + 1$.

Este proceso se repite sucesivamente con el número obtenido en cada paso.

Por ejemplo, si se inicia con $n = 6$:

$$6 \to 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

La **conjetura de Collatz** afirma que, independientemente del número natural inicial elegido, la secuencia generada por este procedimiento terminará eventualmente alcanzando el número 1.

A pesar de su formulación extremadamente simple, la conjetura permanece sin demostración general. Se ha verificado mediante cálculos computacionales para valores muy grandes de n, pero aún no se ha encontrado una prueba matemática que garantice su validez para todos los números naturales.

Este ejemplo es particularmente relevante en el contexto de la teoría de la computación porque muestra que un **procedimiento algorítmico perfectamente definido puede generar comportamientos cuya terminación global es desconocida**. En otras palabras, incluso

cuando un algoritmo está completamente especificado, determinar si siempre termina puede ser un problema profundamente difícil.

2.2.5 La centralidad de las funciones recursivas parciales en la computación práctica

La informática moderna opera bajo una condición estructural: todo sistema de programación suficientemente expresivo —esto es, Turing-completo— debe permitir la existencia de procesos cuya terminación no está garantizada.

Desde el punto de vista formal, esto significa que los lenguajes de programación implementan, en esencia, el modelo de las funciones recursivas parciales. La posibilidad de iteración no acotada (bucles, recursión general) es precisamente lo que otorga potencia computacional plena, pero también lo que introduce la eventual no terminación.

Esta característica no es un accidente de la ingeniería, sino una consecuencia directa de la formalización matemática de la computabilidad desarrollada en la década de 1930.

Sintaxis decidible y semántica no decidible

En el análisis formal de los sistemas computacionales puede distinguirse entre:

1. **Propiedades sintácticas**, que son decidibles.
2. **Propiedades semánticas globales**, que no lo son en general.

La verificación de que un programa está bien formado —esto es, que cumple las reglas gramaticales del lenguaje— es un proceso finito y completamente determinable. Desde el punto de vista formal, corresponde a una función total y efectivamente computable. En términos estructurales, este tipo de verificación puede modelarse mediante funciones recursivas primitivas.

En cambio, determinar si un programa terminará para toda entrada posible, o si producirá un determinado resultado, pertenece al dominio de las propiedades semánticas. Aquí aparece la indecidibilidad: no existe un algoritmo general capaz de decidir el comportamiento de todos los programas posibles.

Este límite se expresa formalmente en el Problema de la Parada, que demuestra que la terminación universal no es decidible mediante un procedimiento efectivo.

Raíz lógica del límite computacional

La estructura de este fenómeno tiene su antecedente en la aritmetización de la sintaxis introducida por Kurt Gödel. Al codificar fórmulas y demostraciones como números naturales, Gödel mostró que preguntas metamatemáticas pueden traducirse en relaciones aritméticas efectivamente definibles.

Este paso permitió revelar una distinción estructural entre:

- Lo que puede verificarse mecánicamente dentro de un sistema formal.
- Lo que no puede decidirse mediante un procedimiento interno al sistema.

En el ámbito computacional, esta distinción reaparece como la separación entre:

- Procesos cuya corrección sintáctica puede verificarse algorítmicamente.
- Procesos cuya terminación o comportamiento global no puede determinarse en general.

Consecuencia estructural

Las funciones recursivas parciales constituyen un modelo matemático ampliamente utilizado para describir la computación real. Permiten describir tanto los programas que concluyen como aquellos cuya ejecución puede prolongarse indefinidamente.

Aceptar funciones parciales implica reconocer un principio fundamental: En modelos Turing-completos, la expresividad computacional general implica admitir procesos cuya terminación no puede garantizarse universalmente.

Este intercambio no es contingente, sino estructural. Esta es una consecuencia estructural de los sistemas Turing-completos.

2.3 Construcción detallada de funciones recursivas primitivas

Para comprender cabalmente el poder y las limitaciones de las funciones recursivas, es instructivo examinar en detalle cómo se construyen las funciones recursivas primitivas a partir de los elementos básicos. Esta sección presenta ejemplos detallados que ilustran el proceso de construcción.

2.3.1 La suma como función recursiva primitiva

La suma puede definirse como una función recursiva primitiva mediante el esquema de recursión sobre el primer argumento. Sea $\text{add}(a, b)$definida por:

Caso base:

$$\text{add}(0, b) = b$$

Paso recursivo:

$$\text{add}(S(a), b) = S(\text{add}(a, b))$$

donde $S(x) = x + 1$ es la función sucesor.

En esta definición, la profundidad de la recursión está controlada por el primer argumento a, lo que garantiza la terminación del proceso.

Ejemplo 1: Cálculo de $\text{add}(2,3)$

Recordemos que:

$$S\big(S(0)\big) = 2$$

Entonces:

$\text{add}(2,3) = \text{add}(S(1),3) = S(\text{add}(1,3))$

Continuando:

$$= S(S(\text{add}(0,3)))$$

Aplicando el caso base:

$\text{add}(0,3) = 3$

Por tanto:

$\text{add}(2,3) = S(S(3)) = S(4) = 5.$

La recursión desciende hasta alcanzar el caso base y luego asciende aplicando sucesivamente la función sucesor.

Ejemplo 2: Cálculo de $\text{add}(4,5)$

Se procede de manera análoga:

$$\text{add}(4,5) = S(\text{add}(3,5)) = S\left(S(\text{add}(2,5))\right)$$

$$= S\left(S\left(S(add(1,5))\right)\right)$$

$$= S(S(S(S(\text{add}(0,5)))))$$

Aplicando el caso base:

$\text{add}(0,5) = 5$

Entonces:

$\text{add}(4,5) = S(S(S(S(5)))) = 9.$

Observación estructural

En la definición de $\text{add}(a, b)$, el argumento a determina el número de aplicaciones del operador sucesor. La recursión está estrictamente acotada por el valor inicial de a, lo que garantiza que el proceso termina tras un número finito de pasos.

Este ejemplo ilustra el mecanismo fundamental de la recursión primitiva: una definición inductiva cuyo alcance está controlado por un parámetro natural que decrece hasta alcanzar el caso base.

2.3.2 La multiplicación como función recursiva primitiva

La multiplicación puede definirse como una función recursiva primitiva utilizando la suma previamente definida. Sea $\text{mult}(a, b)$definida por recursión sobre el primer argumento.

Caso base:

$$\text{mult}(0, b) = 0$$

Paso recursivo:

$$\text{mult}(S(a), b) = \text{add}(b, \text{mult}(a, b))$$

donde add es la función suma definida en la sección anterior.

Esta definición expresa que multiplicar $a + 1$ por b equivale a sumar b al resultado de multiplicar a por b.

Ejemplo 1: Cálculo de $\mathbf{mult}(\mathbf{2}, \mathbf{3})$

Recordemos que:

$$2 = S(S(0))$$

Entonces:

$$\text{mult}(2,3) = \text{mult}(S(1),3) = \text{add}(3, \text{mult}(1,3))$$

Continuando:

$$= \text{add}(3, \text{add}(3, \text{mult}(0,3)))$$

Aplicando el caso base:

$$\text{mult}(0,3) = 0$$

Por tanto:

$$\text{mult}(2,3) = \text{add}(3, \text{add}(3,0))$$

Sabemos que:

$$\text{add}(3,0) = 3$$

Entonces:

$$\text{mult}(2,3) = \text{add}(3,3) = 6.$$

Ejemplo 2: Cálculo de $\mathbf{mult}(\mathbf{4},\mathbf{2})$

Aplicando la definición recursiva:

$$\text{mult}(4,2) = \text{add}(2, \text{mult}(3,2)) = \text{add}(2, \text{add}(2, \text{mult}(2,2)))$$
$$= \text{add}(2, \text{add}(2, \text{add}(2, \text{mult}(1,2))))$$
$$= \text{add}(2, \text{add}(2, \text{add}(2, \text{add}(2, \text{mult}(0,2)))))$$

Aplicando el caso base:

$$\text{mult}(0,2) = 0$$

Entonces:

$$\text{mult}(4,2) = \text{add}(2, \text{add}(2, \text{add}(2, \text{add}(2,0)))) = 8.$$

Observación estructural

En la definición de $\text{mult}(a, b)$, la profundidad de la recursión está determinada por el argumento a. Cada paso recursivo añade una suma adicional de b, por lo que la multiplicación puede interpretarse como una **iteración de la suma**.

Este ejemplo muestra cómo las funciones recursivas primitivas pueden construirse jerárquicamente: operaciones más complejas se definen a partir de funciones previamente establecidas mediante composición y recursión primitiva.

La resta truncada como función recursiva primitiva

En el conjunto de los números naturales $\mathbb{N} = \{0,1,2, \dots\}$ no existen números negativos. Por esta razón, la operación de resta usual no siempre está definida dentro de $\mathbb{N}$. Para mantener la operación dentro del conjunto de los naturales se define la **resta truncada** (también llamada *resta limitada* o *monus*), que nunca produce valores negativos.

La resta truncada se define como una función recursiva primitiva:

$$sub(x, y) \text{ o } x \dot{-} y$$

donde el resultado es $x - y$ si $x \geq y$, y 0 en caso contrario.

Definición recursiva

Caso base

$$sub(x, 0) = x$$

Restar cero no modifica el valor inicial.

Paso recursivo

$$sub(x, S(y)) = pred(sub(x, y))$$

donde **pred** es la función predecesora limitada:

$$pred(0) = 0, \;\; pred(S(n)) = n$$

Es decir, cada paso recursivo disminuye el resultado en una unidad, sin permitir que el valor sea negativo.

Ejemplo: cálculo de $sub(5, 2)$

Aplicamos la definición recursiva:

$$sub(5,2) = pred(sub(5,1))$$

Ahora evaluamos el término interno:

$$sub(5,1) = pred(sub(5,0))$$

Aplicando el caso base:

$$sub(5,0) = 5$$

Entonces:

$$sub(5,1) = pred(5) = 4$$

Finalmente:

$$sub(5,2) = pred(4) = 3$$

Por lo tanto:

$$sub(5,2) = 3$$

Interpretación del proceso recursivo

También puede visualizarse el descenso recursivo de la siguiente forma:

$$sub(5,2) = pred(sub(5,1))$$
$$sub(5,1) = pred(sub(5,0))$$
$$sub(5,0) = 5$$

Luego comienza la fase de ascenso:

$$sub(5,1) = pred(5) = 4$$
$$sub(5,2) = pred(4) = 3$$

Observación estructural

En la definición de $sub(x, y)$, **la profundidad de la recursión está controlada por el segundo argumento** y. Cada paso recursivo aplica la función predecesora al resultado anterior, lo que equivale a restar una unidad.

Debido a que el número de pasos recursivos está acotado por el valor inicial de y, el proceso siempre termina tras un número finito de iteraciones. Por esta razón, la **resta truncada es una función recursiva primitiva**.

Además, este ejemplo ilustra nuevamente cómo las funciones recursivas primitivas se construyen jerárquicamente: la resta se define utilizando otra función previamente establecida (el predecesor) dentro del esquema de recursión primitiva.

La exponenciación como función recursiva primitiva

La operación de exponenciación puede definirse dentro del marco de las funciones recursivas primitivas utilizando la función de multiplicación previamente definida. Intuitivamente, elevar un número a una potencia consiste en multiplicarlo por sí mismo un número determinado de veces.

Sea la función:

$$exp(x, y)$$

que representa la potencia x^y.

Definición recursiva

Caso base

$$exp(x, 0) = 1$$

Elevar cualquier número natural a la potencia cero produce el valor uno.

Paso recursivo

$$exp(x, S(y)) = mult(exp(x, y), x)$$

Esto significa que para calcular x^{y+1}se multiplica x por el resultado de x^y.

Ejemplo: cálculo de $\boldsymbol{exp(5,3)}$

Aplicando la definición recursiva:

$$exp(5,3) = mult(exp(5,2),5)$$
$$exp(5,2) = mult(exp(5,1),5)$$
$$exp(5,1) = mult(exp(5,0),5)$$

Aplicando el caso base:

$$exp(5,0) = 1$$

Ahora comienza el ascenso del proceso recursivo:

$$exp(5,1) = mult(1,5) = 5$$
$$exp(5,2) = mult(5,5) = 25$$
$$exp(5,3) = mult(25,5) = 125$$

Por lo tanto:

$$exp(5,3) = 125$$

Interpretación del proceso recursivo

El cálculo puede visualizarse como una expansión recursiva:

$$exp(5,3) = mult(exp(5,2),5)$$
$$exp(5,2) = mult(exp(5,1),5)$$
$$exp(5,1) = mult(exp(5,0),5)$$
$$exp(5,0) = 1$$

Posteriormente se realiza la fase de evaluación ascendente hasta obtener el resultado final.

Observación estructural

En la función $exp(x, y)$, **la profundidad de la recursión está determinada por el segundo argumento** y. Cada paso recursivo aplica una multiplicación adicional por x.

Por lo tanto, la exponenciación puede interpretarse como **una iteración de la multiplicación**, de la misma forma que la multiplicación es una iteración de la suma. Esta construcción muestra cómo las funciones recursivas primitivas se organizan jerárquicamente: operaciones cada vez más complejas se definen a partir de funciones previamente establecidas mediante recursión primitiva.

La función factorial como función recursiva primitiva

La función factorial es una de las funciones clásicas en matemáticas discretas y aparece frecuentemente en combinatoria, teoría de probabilidades y análisis de algoritmos.

El factorial de un número natural n, denotado por $n!$, se define como el producto de todos los números naturales positivos menores o iguales que n.

Sea la función:

$$fact(n)$$

Definición recursiva

Caso base

$$fact(0) = 1$$

Por definición, el factorial de cero es igual a uno.

Paso recursivo

$$fact(S(n)) = mult(S(n), fact(n))$$

Esto significa que:

$$(n+1)! = (n+1) \times n!$$

Ejemplo: cálculo de $fact(4)$

Aplicando la definición recursiva:

$$fact(4) = mult(4, fact(3))$$
$$fact(3) = mult(3, fact(2))$$
$$fact(2) = mult(2, fact(1))$$
$$fact(1) = mult(1, fact(0))$$

Aplicando el caso base:

$$fact(0) = 1$$

Ahora comienza el proceso de evaluación ascendente:

$$fact(1) = mult(1,1) = 1$$
$$fact(2) = mult(2,1) = 2$$
$$fact(3) = mult(3,2) = 6$$
$$fact(4) = mult(4,6) = 24$$

Por lo tanto:

$$fact(4) = 24$$

Interpretación del proceso recursivo

El desarrollo recursivo puede visualizarse como una cadena de multiplicaciones sucesivas:

$$fact(4) = 4 \times 3 \times 2 \times 1$$

donde cada paso recursivo introduce un nuevo factor multiplicativo hasta alcanzar el caso base.

Observación estructural

En la definición de $fact(n)$, **la profundidad de la recursión está controlada por el argumento** n. Cada llamada recursiva reduce el argumento en una unidad hasta alcanzar el caso base $fact(0)$.

Debido a que el número de pasos recursivos está acotado por el valor inicial de n, el proceso siempre termina tras un número finito de iteraciones. Por esta razón, la función factorial constituye **un ejemplo clásico de función recursiva primitiva**.

2.3.6 La división entera mediante división acotada

La división entera entre dos números naturales consiste en determinar cuántas veces un número b cabe dentro de otro número a. El resultado se conoce como **cociente entero** y se denota como:

$$\left\lfloor \frac{a}{b} \right\rfloor$$

donde el resultado es el mayor número natural q tal que:

$$b \cdot q \leq a$$

Sin embargo, la división entera no puede definirse directamente como función recursiva primitiva, ya que el número de iteraciones necesarias para calcular el cociente depende de los valores de a y b y no está explícitamente acotado.

Para resolver este problema se introduce la **división acotada**, que incorpora un tercer parámetro que limita el número máximo de pasos recursivos.

Sea la función:

$$div\_acotada(a, b, c)$$

donde c representa el número máximo de iteraciones permitidas.

Definición recursiva

Caso base

$$div\_acotada(a, b, 0) = 0$$

Si no se permiten más iteraciones, el resultado es cero.

Paso recursivo

Si $a < b$:

$$div\_acotada(a, b, c) = 0$$

Si $a \geq b$:

$$div\_acotada(a, b, S(c)) = 1 + div\_acotada(a - b, b, c)$$

En cada paso se resta b del valor actual de a y se incrementa el contador del cociente.

División entera como caso particular

La división entera usual puede obtenerse fijando el límite suficientemente grande. En particular, basta elegir:

$$c = a$$

ya que el cociente nunca puede ser mayor que a.

Por lo tanto:

$$\left\lfloor \frac{a}{b} \right\rfloor = div\_acotada(a, b, a)$$

Ejemplo: cálculo de $\lfloor 25/5 \rfloor$

Utilizando $c = 25$:

$$div\_acotada(25{,}5{,}25)$$

Aplicando la definición recursiva:

$$div\_acotada(25{,}5{,}25) = 1 + div\_acotada(20{,}5{,}24)$$
$$div\_acotada(20{,}5{,}24) = 1 + div\_acotada(15{,}5{,}23)$$
$$div\_acotada(15{,}5{,}23) = 1 + div\_acotada(10{,}5{,}22)$$
$$div\_acotada(10{,}5{,}22) = 1 + div\_acotada(5{,}5{,}21)$$
$$div\_acotada(5{,}5{,}21) = 1 + div\_acotada(0{,}5{,}20)$$

Ahora se alcanza el caso base porque $0 < 5$:

$$div\_acotada(0{,}5{,}20) = 0$$

Entonces:

$$1 + 1 + 1 + 1 + 1 + 0 = 5$$

Por lo tanto:

$$\left\lfloor \frac{25}{5} \right\rfloor = 5$$

Interpretación del proceso recursivo

El algoritmo puede interpretarse como **una sucesión de restas repetidas** del divisor b sobre el dividendo a. Cada resta válida incrementa el valor del cociente.

El parámetro c actúa como un **control explícito del número máximo de iteraciones**, garantizando que el proceso siempre termina.

Observación estructural

La función $div_acotada(a, b, c)$es recursiva primitiva porque la profundidad de la recursión está controlada por el parámetro c. Esto asegura que el número de llamadas recursivas es finito.

Aunque la división entera no es directamente una función recursiva primitiva, puede expresarse como un **caso particular de una función recursiva primitiva acotada**, lo que permite integrarla dentro del marco formal de las funciones computables sobre los números naturales.

2.3.7 Otras funciones recursivas primitivas

Además de las operaciones aritméticas fundamentales como suma, multiplicación, exponenciación y factorial, es posible construir diversas funciones auxiliares dentro del marco de las **funciones recursivas primitivas**. Estas funciones permiten expresar comparaciones, operaciones lógicas y operadores aritméticos básicos utilizando únicamente las operaciones previamente definidas, especialmente la **resta truncada** $x \dot{-} y$.

A continuación, se presentan algunos ejemplos importantes.

Valor absoluto de la distancia

Definición

El valor absoluto de la diferencia entre dos números naturales se define como:

$$| x - y | = (x \dot{-} y) + (y \dot{-} x)$$

donde $x \dot{-} y$ representa la **resta truncada**, definida como:

$$x \dot{-} y = \begin{cases} x - y & \text{si } x \geq y \\ 0 & \text{si } x < y \end{cases}$$

Construcción

La función se construye mediante **composición** de:

- la resta truncada
- la suma

Ambas funciones son recursivas primitivas, por lo que su composición también lo es.

Ejemplo

Calcular:

$$| 2 - 4 |$$

Aplicando la definición:

$$| 2 - 4 | = (2 \dot{-} 4) + (4 \dot{-} 2)$$

Sabemos que:

$$2 \dot{-} 4 = 0$$

y

$$4 \dot{-} 2 = 2$$

Por lo tanto:

$$| 2 - 4 | = 0 + 2 = 2$$

Función de igualdad

Definición

La función de igualdad $eq(x, y)$se define como:

$$eq(x, y) = 1 \dot{-} ((x \dot{-} y) + (y \dot{-} x))$$

Esta función devuelve:

$$eq(x,y) = \begin{cases} 1 & \text{si } x = y \\ 0 & \text{si } x \neq y \end{cases}$$

Construcción

La función se construye utilizando:

- resta truncada
- suma
- composición de funciones

Todas ellas pertenecen a la clase de funciones recursivas primitivas.

Ejemplo

$$eq(3,3)$$
$$|\ 3 - 3\ | = 0$$
$$eq(3,3) = 1$$

En cambio:

$$q(3,5) = 0$$

Función menor o igual

Definición

La función característica de **menor o igual**, $le(x,y)$, se define como:

$$le(x,y) = 1 \dot{-} (x \dot{-} y)$$

Esta función toma los valores:

$$le(x,y) = \begin{cases} 1 & \text{si } x \leq y \\ 0 & \text{si } x > y \end{cases}$$

Construcción

Se basa en una propiedad fundamental de la resta truncada:

- si $x \leq y$ entonces $x \dot{-} y = 0$
- si $x > y$ entonces $x \dot{-} y > 0$

A partir de esta propiedad es posible construir la función característica mediante composición.

Ejemplo

$$le(3,5) = 1$$

porque

$$3 \dot{-} 5 = 0$$

En cambio:

$$le(6,2) = 0$$

Función de negación

Definición

La función de negación booleana se define como:

$$neg(x) = 1 \dot{-} x$$

Esta función invierte valores lógicos:

$$neg(x) = \begin{cases} 1 & \text{si } x = 0 \\ 0 & \text{si } x = 1 \end{cases}$$

Construcción

La negación se construye directamente mediante la **resta truncada**, utilizando el valor constante 1.

Ejemplo

$$neg(1) = 0$$
$$neg(0) = 1$$

Función máximo

Definición

La función máxima se define como:

$$max(x, y) = x + (y \dot{-} x)$$

Construcción

La función se obtiene combinando:

- suma
- resta truncada

Ejemplo

Calcular:

$$max(3,7)$$
$$7 \dot{-} 3 = 4$$
$$max(3,7) = 3 + 4 = 7$$

2.3.7.6 Función mínimo

Definición

La función mínima puede definirse como:

$$min(x, y) = x - (x \dot{-} y)$$

Construcción

La función utiliza:

- resta truncada
- composición de funciones

Ejemplo

Calcular:

$$min(3,7)$$
$$3 \dot{-} 7 = 0$$
$$min(3,7) = 3 - 0 = 3$$

Observación

Estos ejemplos muestran que, a partir de un conjunto reducido de funciones básicas y mediante los esquemas de **composición y recursión primitiva**, es posible construir una amplia variedad de operaciones aritméticas y lógicas. Sin embargo, a pesar de su expresividad, las funciones recursivas primitivas presentan limitaciones importantes que impiden representar todas las funciones computables.

Estas limitaciones se analizan en la siguiente sección.

2.4 Limitaciones de las funciones recursivas primitivas

Las **funciones recursivas primitivas (FRP)** constituyen una clase importante dentro de las funciones computables sobre los números naturales. Se construyen a partir de funciones básicas mediante los esquemas de **composición** y **recursión primitiva**, lo que garantiza que todas ellas son **funciones totales**, es decir, su cálculo siempre termina tras un número finito de pasos.

Sin embargo, a pesar de su aparente versatilidad, las FRP **no son suficientes para capturar toda la computabilidad efectiva**. Esta limitación surge directamente de su propia definición.

2.4.1 La barrera de la recursión acotada

En la recursión primitiva, la definición de una función tiene la forma:

$$f(S(x), \vec{y}) = g(x, f(x, \vec{y}), \vec{y})$$

En este esquema, la profundidad de la recursión está **acotada por el valor inicial del argumento**. Esto significa que el número de llamadas recursivas está determinado de antemano por el tamaño de la entrada.

Por ejemplo, al calcular la suma $add(x, y)$, sabemos que el proceso requerirá exactamente x pasos recursivos.

Esta propiedad tiene dos consecuencias importantes:

- **Ventaja:** garantiza que el proceso siempre termina.
- **Limitación:** impide expresar procesos cuya duración no pueda acotarse previamente.

Por esta razón, existen funciones computables que **no pueden definirse mediante recursión primitiva**.

2.5 Funciones recursivas generales

Para superar esta limitación se introduce un mecanismo más poderoso: el **operador de minimización** o **μ-operador**.

2.5.1 El operador de minimización

El operador de minimización permite definir funciones de la forma:

$$f(\vec{x}) = \mu y\, [g(\vec{x}, y) = 0]$$

Esto significa que $f(\vec{x})$es el **menor valor de** y para el cual se cumple la condición $g(\vec{x}, y) = 0$.

A diferencia de la recursión primitiva, este proceso **no tiene un límite preestablecido** para el número de pasos necesarios. En algunos casos puede ocurrir que **no exista tal valor de** y, lo que implica que el proceso nunca termina.

Por esta razón aparecen dos tipos de funciones:

Funciones recursivas totales

Son funciones computables que **terminan para todas las entradas**.

Funciones recursivas parciales

Son funciones computables cuyo proceso puede **no terminar para algunas entradas**.

En la práctica, los programas escritos en lenguajes de programación generales corresponden a **funciones recursivas parciales**, ya que siempre existe la posibilidad de bucles infinitos o

recursión no terminante (la función se llama a sí misma pero **nunca alcanza el caso base** o no tiene uno).

2.5.2 Significado filosófico y práctico

La distinción entre **funciones totales** y **funciones parciales** refleja una diferencia fundamental en la forma de concebir los algoritmos y los problemas computacionales.

1. Funciones totales.
Corresponden a problemas para los cuales el algoritmo siempre produce un resultado tras un número finito de pasos. Ejemplos típicos son la **suma**, la **multiplicación**, el **factorial** o los algoritmos de **ordenamiento**.

2. Funciones parciales.
Representan problemas para los cuales el procedimiento computacional puede no terminar para algunas entradas. En estos casos, el algoritmo produce un resultado sólo cuando la ejecución concluye. Un ejemplo importante incluye la **evaluación general de programas**.

En la práctica, la mayoría de los **lenguajes de programación de propósito general** permiten escribir programas con bucles potencialmente infinitos o recursión no acotada. Por esta razón, desde el punto de vista teórico, los programas corresponden en general a **funciones recursivas parciales**, más que a funciones totales.

Esta observación muestra que el modelo matemático de las **funciones recursivas parciales** captura de manera más fiel el comportamiento real de los programas.

Ejemplo ilustrativo

Consideremos el siguiente procedimiento conceptual:

$f(n)$:

$mientras\ n \neq 0\ hacer$

$n \leftarrow n + 1$

$regresar\ 0$

fin

Si la entrada es $n = 0$, el algoritmo termina inmediatamente y devuelve 0. Sin embargo, si $n \neq 0$, el valor de n aumenta indefinidamente y el programa **nunca termina**.

Por lo tanto, esta función:

- está **definida para algunas entradas**,
- pero **no produce resultado para otras**.

En consecuencia, corresponde a una **función parcial**.

Este ejemplo ilustra cómo un algoritmo aparentemente simple puede no garantizar la terminación para todas las entradas.

2.5.3 El problema de determinar la totalidad

Una cuestión profunda dentro de la teoría de la computación es la siguiente:

¿Es posible determinar mediante un algoritmo si una función recursiva parcial dada es en realidad total?

Es decir, dado el código de un programa o la descripción de una función μ-recursiva, ¿podemos decidir automáticamente si el cálculo terminará para **todas** las entradas posibles?

La respuesta es **negativa**. No existe un algoritmo general capaz de determinar si un programa arbitrario termina para todas sus entradas.

Esta limitación tiene consecuencias importantes tanto en la teoría como en la práctica.

Verificación de software.
Implica que no es posible construir verificadores completamente automáticos que garanticen la terminación de cualquier programa sin intervención humana.

Sistemas formales y teoría de tipos.
En algunos sistemas lógicos utilizados en verificación formal, como el **Cálculo de Construcciones**, se imponen restricciones sintácticas sobre la recursión para garantizar que todos los programas terminen. Estos sistemas renuncian deliberadamente a la **completitud**

de Turing con el fin de asegurar propiedades como la **normalización fuerte**, que garantiza que toda evaluación converge a un resultado.

2.6 La función de Ackermann y la jerarquía de computabilidad

La **función de Ackermann** constituye uno de los ejemplos más importantes para comprender las limitaciones de las funciones recursivas primitivas.

2.6.1 Definición

La función de Ackermann $A(m, n)$se define recursivamente como:

$$A(0, n) = n + 1$$
$$A(m + 1,0) = A(m, 1)$$
$$A(m + 1, n + 1) = A(m, A(m + 1, n))$$

Esta función presenta varias propiedades fundamentales:

1. **Es total**: está definida para todos los números naturales.
2. **Es computable**: existe un procedimiento efectivo para calcularla.
3. **No es primitiva recursiva**: no puede definirse utilizando únicamente composición y recursión primitiva.

La razón es que su crecimiento es **más rápido que cualquier función primitiva recursiva**.

2.6.2 Consecuencias para la teoría de la computación

La existencia de funciones como la de Ackermann demuestra que:

1. La clase de funciones primitivas recursivas **no agota la computabilidad**.
2. Existen funciones totales computables **más allá de las FRP**.
3. Para capturar toda la computabilidad es necesario considerar **funciones μ-recursivas**.

Esto conduce a la siguiente jerarquía fundamental:

FRP ⊂ Funciones recursivas totales ⊂ Funciones recursivas parciales

Las funciones recursivas parciales coinciden en poder expresivo con:

- las **μ-recursivas**
- las **máquinas de Turing**
- el **cálculo lambda**

lo que refleja la equivalencia entre los principales modelos formales de computación.

2.6.3 Terminación y límites de la computación

Aceptar funciones recursivas parciales implica reconocer un principio fundamental de la computación: **la potencia expresiva completa solo es posible renunciando a la garantía universal de terminación**.

Un ejemplo ilustrativo de estos límites aparece en la famosa conjetura de Collatz. Aunque el procedimiento que la define consiste en una regla extremadamente simple aplicada de forma iterativa, todavía no se sabe si siempre termina en el valor 1 para cualquier número natural inicial. Este tipo de problemas revela una idea fundamental de la teoría de la computación: incluso cuando un algoritmo está perfectamente definido, determinar si su ejecución siempre termina puede ser profundamente difícil. Esta intuición se conecta directamente con uno de los resultados más importantes de la computación teórica: el **problema de parada**, demostrado por Alan Turing en 1936, el cual establece que no existe un algoritmo general capaz de decidir si cualquier programa arbitrario se detendrá o continuará ejecutándose indefinidamente. De este modo, ejemplos aparentemente simples como la conjetura de Collatz anticipan una conclusión profunda: existen límites intrínsecos a lo que la computación puede decidir.

$Collatz \rightarrow problema\ de\ la\ parada \rightarrow Turing \rightarrow limites\ de\ la\ computación.$

Capítulo III · Impacto de la computabilidad en la computación contemporánea

Los límites establecidos por la teoría de la computabilidad en la década de 1930 no permanecen confinados a la lógica matemática. Hoy constituyen los principios estructurales que determinan qué es posible y qué no en la ingeniería de software, en el diseño de lenguajes de programación, en la verificación automática de sistemas, en la inteligencia artificial y en la seguridad informática.
Este capítulo expone, de manera precisa, cómo esos resultados teóricos influyen en la práctica computacional moderna.

3.1 Computabilidad y verificación de sistemas

La teoría de la computabilidad demuestra la existencia de problemas que ninguna máquina puede resolver. De este hecho surge la división fundamental entre problemas decidibles e indecidibles. Una consecuencia directa es la imposibilidad de construir un verificador universal de programas: ningún algoritmo puede determinar, para cualquier programa imaginable, si siempre terminará, si nunca producirá errores o si es correcto en todos los casos.

Incluso los asistentes de prueba más avanzados (Coq, Lean, Isabelle) operan dentro de límites matemáticos estrictos: sólo pueden garantizar corrección en el marco lógico que aceptan. Por ello, la verificación total es inalcanzable. En los sistemas críticos se adoptan estrategias mixtas: verificación parcial, pruebas exhaustivas, redundancia y análisis de riesgo. El ideal de un software infalible no es técnicamente posible debido a restricciones formales, no de ingeniería.

2.2 Lenguajes de programación y compiladores

Los lenguajes de programación Turing-completos están sujetos a las mismas limitaciones formales que los sistemas lógicos en los que se originó la teoría de la computabilidad. En particular, el Teorema de Rice establece que ninguna propiedad semántica no trivial de los

programas puede decidirse algorítmicamente en el caso general. Este resultado afecta preguntas fundamentales como:

- ¿Termina este programa para toda entrada?
- ¿Puede garantizarse que nunca producirá un error?
- ¿Implementa correctamente una especificación dada?
- ¿Evita todo acceso fuera de rango?

Dado que estas propiedades dependen del comportamiento dinámico del programa durante su ejecución, no existe un procedimiento universal capaz de decidirlas para todos los programas posibles.

Como consecuencia, los diseñadores de lenguajes de programación enfrentan una disyuntiva estructural entre expresividad computacional y verificabilidad formal:

A. Restringir la expresividad

Lenguajes como Coq o Agda limitan la recursión general y controlan explícitamente los efectos computacionales. Estas restricciones permiten garantizar propiedades como terminación y corrección formal, aunque a costa de reducir la expresividad directa y la flexibilidad del lenguaje.

B. Aceptar la indecidibilidad

Los lenguajes de propósito general, como C, Python o Java, privilegian la expresividad computacional completa. Esto permite implementar cualquier procedimiento efectivamente computable, pero implica aceptar que no existe un método general capaz de verificar automáticamente todas las propiedades relevantes de los programas, incluyendo terminación, corrección total o ausencia absoluta de errores

3.3 La indecidibilidad como estructura del software moderno

Todo compilador moderno está construido sobre una negociación con la indecidibilidad. Puede verificar sintaxis, tipos básicos o estructuras de control mediante procedimientos totalmente decidibles, pero debe recurrir a aproximaciones, heurísticas, análisis

conservadores o pruebas empíricas para razonar sobre comportamiento dinámico o corrección global.

En la práctica, la ingeniería del software aplica un principio permanente:

> *todo lo decidible debe verificarse por completo; todo lo indecidible debe aproximarse*.

3.3 Inteligencia artificial y el límite de lo algorítmico

Los resultados de Gödel, Church y Turing también tienen consecuencias para la comprensión de la inteligencia artificial. Todo sistema algorítmico opera dentro de un marco formal, y los marcos formales suficientemente expresivos presentan límites de completitud y de demostrabilidad.

Algunos autores, como Roger Penrose, han planteado una reflexión específica:

> *Si la mente humana fuera reducible a un algoritmo, debería existir un sistema formal capaz de capturar completamente su capacidad demostrativa.*

Sin embargo, los teoremas de incompletitud muestran que ningún sistema formal suficientemente potente puede contener todas las verdades matemáticas que él mismo formula. De este modo, si la actividad demostrativa de la mente fuera puramente algorítmica, su capacidad quedaría necesariamente restringida por esos límites.

Penrose argumenta que el matemático humano puede reconocer la verdad de una proposición indecidible del sistema al situarse metateóricamente sobre él. Bajo esta hipótesis, la comprensión humana incluiría una forma de razonamiento que no puede explicarse solo mediante manipulación sintáctica.

Más allá del debate filosófico, la conclusión operativa es nítida: ningún sistema de IA, por sofisticado que sea, puede escapar a los límites de la computabilidad. Una IA no puede ser completa, no puede autojustificarse en todos los casos

y no puede garantizar corregibilidad global de su propio comportamiento. Sus límites no son tecnológicos, sino estructurales.

Esto no reduce su valor práctico, pero sí establece con claridad el dominio de lo que puede y no puede lograrse mediante algoritmos.

3.4 Seguridad informática y criptografía

La seguridad informática moderna depende significativamente de principios derivados de la computabilidad y de la complejidad. Muchas técnicas criptográficas se basan en la existencia de **funciones unidireccionales**: fáciles de ejecutar, pero difíciles de invertir sin información adicional. La imposibilidad práctica de resolver ciertos problemas garantiza la resistencia de los sistemas de cifrado.

Del mismo modo, la indecidibilidad tiene implicaciones directas: no existe un detector universal de malware, ni un analizador capaz de anticipar todos los comportamientos anómalos, ni un sistema que garantice seguridad absoluta. La seguridad depende siempre de suposiciones específicas y modelos de amenaza. La perfección, en este ámbito, está matemáticamente descartada.

2.5 Computación moderna: una ciencia de límites

El resultado más profundo de la teoría de la computabilidad es conceptual: **la computación es inherentemente limitada**.

Desde los algoritmos probabilísticos hasta los modelos híbridos humano-máquina, las herramientas actuales tienen éxito porque operan dentro de los límites formales establecidos por Gödel, Church y Turing. La ingeniería moderna acepta estos límites y adopta métodos como:

- aproximación,
- heurística,
- análisis parcial,

- verificación fragmentaria,
- redundancia,
- experimentación.

La computación contemporánea no nació de la idea de omnipotencia algorítmica, sino del reconocimiento crítico de lo que ningún algoritmo puede garantizar. Precisamente por asumir esos límites, la computación pudo consolidarse como una ciencia rigurosa.

Capítulo VI · Herramientas resultantes: de la teoría a la práctica

La historia de la computación demuestra que los lenguajes y herramientas actuales no emergieron por evolución espontánea, sino como consecuencias directas de los modelos matemáticos desarrollados durante el siglo XX.
La Máquina de Turing consolidó el paradigma imperativo y secuencial; el cálculo lambda de Church formalizó el paradigma funcional y la noción de función como transformación pura; y las funciones recursivas establecieron las bases para razonar sobre terminación, expresividad y estructura del cómputo.
Como resultado, las herramientas modernas —desde lenguajes hasta frameworks educativos— son encarnaciones prácticas de modelos teóricos rigurosos**.** Esto coincide con el análisis histórico y formal previamente expuesto.

4.1 Lisp y Scheme: lenguajes basados en el cálculo lambda

Lisp y Scheme representan la materialización más directa del cálculo lambda en un entorno de programación práctica. No son únicamente lenguajes funcionales: son **laboratorios conceptuales** donde los principios fundamentales de la computabilidad se ejecutan sin mediaciones adicionales. Su relevancia deriva de cuatro pilares teóricos:

1. Origen formal: continuidad directa con el cálculo lambda

Lisp nace de la observación de McCarthy de que el cálculo lambda puede implementarse como un sistema real de programación.
Esto demuestra que el modelo de Church no es solo una abstracción matemática, sino un **marco operativo capaz de describir y ejecutar algoritmos**, lo que coincide con la equivalencia entre modelos de computación destacada en el capítulo histórico.

2. Recursión como mecanismo estructural fundamental

En Lisp y Scheme, **toda construcción iterativa puede expresarse como recursión**, lo cual alinea estos lenguajes con el marco conceptual de las funciones recursivas primitivas y generales, discutido en capítulos previos.

El flujo de control no depende de bucles, sino de definiciones recursivas que reflejan directamente:

- composición,
- recursión primitiva,
- y minimización acotada o no acotada.

Así, estos lenguajes enseñan recursividad no como técnica, sino como **estructura matemática del cálculo**.

3. Homoiconicidad: código como datos

La propiedad según la cual las expresiones del lenguaje tienen la misma estructura que los datos manipulados por el lenguaje permite que Lisp y Scheme materialicen la aritmetización de la sintaxis presentada por Gödel.
Esto convierte a estos lenguajes en entornos privilegiados para:

- análisis sintáctico,
- generación automática de código,
- interpretación metaprogramática,
- construcción de sistemas simbólicos completos.

En este sentido, Lisp aproxima el ideal de sistemas autorreferenciales utilizado en las demostraciones de incompletitud.

4. Optimización de recursión de cola (TCO)

Scheme garantiza la optimización de llamadas en cola, lo que equipara el costo espacial de la recursión al de la iteración imperativa**.**
Esta propiedad establece un puente entre el modelo matemático (donde la recursión es dominante) y las limitaciones físicas de los sistemas reales, un tema ampliamente tratado al examinar funciones de gran profundidad como Ackermann.

Lisp y Scheme no son lenguajes históricos marginales: son **manifestaciones concretas de los fundamentos matemáticos de la computación**. Por ello, siguen siendo cruciales en

cursos avanzados de computabilidad, lógica matemática, inteligencia artificial simbólica y diseño de lenguajes.

4.2 Lisp o Scheme, ventajas y desventajas como primer lenguaje de programación

La elección de un lenguaje inicial influye profundamente en la estructura cognitiva con la que los estudiantes conceptualizan la computación. Los lenguajes funcionales permiten introducir al estudiante en el pensamiento matemático del algoritmo, mientras que los lenguajes imperativos priorizan la intuición operacional (ver tabla 2).

Tabla 2 Comparación conceptual

Aspecto	Lenguajes funcionales (Lisp/Scheme)	Lenguajes imperativos (Python/Java)
Ventajas	- Fomentan pensamiento recursivo y abstracto. - Introducen directamente conceptos de computabilidad (funciones recursivas, expresividad). - REPL interactivo: experimentación inmediata, ideal para explorar semántica formal.	- Sintaxis intuitiva y familiar. - Modelo paso-a-paso más accesible. - Relevancia industrial y abundancia de recursos.
Desventajas	- Sintaxis prefija y abundancia de paréntesis. - Curva inicial exigente. - Menor presencia industrial directa.	- Pueden ocultar los fundamentos teóricos (recursión, semántica). - Riesgo de adoptar malas prácticas si no se enseñan estructuras formales. - La sintaxis puede enmascarar complejidad y dificultar el análisis matemático.

4.3 Importancia epistemológica y pedagógica de los lenguajes funcionales

El uso de lenguajes derivados del cálculo lambda como primera aproximación a la programación tiene una justificación teórica profunda:

1. Modelan directamente la noción formal de algoritmo

Los estudiantes no solo aprenden a programar, sino a comprender **qué significa calcular**, un tema clave recogido en la genealogía del Capítulo I.

2. Permiten observar los límites del cómputo

La recursión no controlada, la divergencia y la no terminación se manifiestan de manera explícita, lo que conecta de inmediato con la teoría de funciones parciales y el problema de la parada.

3. Facilitan la enseñanza de metateoría y semántica

La homoiconicidad permite mostrar:

- cómo se construye un intérprete,
- cómo se implementa un evaluador lambda,
- cómo se analiza semánticamente un programa.

Todo ello coincide con el énfasis del documento en la separación sintaxis/semántica y su representación formal.

4. Son ideales para entender la equivalencia entre modelos de computación.

Lisp/Scheme permiten ilustrar en la práctica:

- la tesis de Church–Turing,
- la equivalencia entre máquinas de Turing y sistemas lambda,
- la relación entre recursividad y definición funcional.

Estos puntos aparecen reiterados como eje conceptual en el análisis del capítulo histórico.

4.4 Lugar de Lisp y Scheme en la computación moderna

Lejos de ser lenguajes “antiguos”, Lisp y Scheme siguen teniendo usos críticos:

- Metaprogramación avanzada
- Compiladores y transformaciones sintácticas

- Sistemas de inferencia lógica
- Inteligencia artificial simbólica
- Investigación en teoría de lenguajes
- Educación superior en computación teórica

Expresan el paradigma en su forma más pura: no hay distancia entre la teoría matemática y la implementación.

La ingeniería del software moderna descansa sobre principios matemáticos profundos. Lisp y Scheme no solo ejemplifican esa raíz teórica, sino que muestran cómo un modelo de computación formal —el cálculo lambda— puede convertirse en un sistema práctico poderoso.
Como tales, constituyen herramientas esenciales para comprender:

- la estructura formal del algoritmo,
- la semántica de los lenguajes,
- la equivalencia entre modelos de computación,
- y los límites fundamentales del cálculo.

Por ello, su estudio no es opcional ni anecdótico: es una puerta directa hacia el corazón matemático de la computación.

Capítulo V. Algoritmos y su clasificación

En los capítulos anteriores se estableció el marco formal de la computabilidad, demostrando que la informática moderna es, en esencia, una ciencia de límites. Como se analizó en el Capítulo III, las fronteras trazadas por Gödel, Church y Turing no son meras curiosidades teóricas, sino los principios estructurales que rigen la ingeniería de software contemporánea. La indecidibilidad y la imposibilidad de construir un verificador universal nos han enseñado que, en la práctica, todo lo decidible se verifica exhaustivamente, mientras que lo indecidible debe abordarse mediante aproximaciones, heurísticas y modelos probabilísticos.

Lejos de paralizar el avance tecnológico, esta conciencia de los límites impulsó la creación de herramientas prácticas extraordinarias. Tal como se expuso en el Capítulo IV, los lenguajes de programación modernos y los paradigmas formales —ilustrados por la encarnación del cálculo lambda en Lisp y Scheme— demuestran cómo la teoría matemática pura se materializa en arquitecturas operativas. Hoy disponemos de sistemas que nos permiten expresar la estructura formal del algoritmo y navegar con precisión la frontera entre una sintaxis decidible y una semántica de comportamiento incomputable.

Sin embargo, conocer que un problema es computable y poseer las herramientas formales para expresarlo no agota el estudio del cálculo. Una vez establecida la existencia de un procedimiento efectivo, surge una cuestión igualmente fundamental: *cómo* se organizan concretamente las instrucciones para resolver un problema de la manera más adecuada. Mientras la teoría de la computabilidad responde a la pregunta sobre *qué* puede calcularse en principio, el estudio de los algoritmos se ocupa de la organización estructural y la estrategia del proceso de cálculo.

Un algoritmo, entendido como una secuencia finita de instrucciones bien definidas que transforma datos en resultados de manera mecánica, puede adoptar arquitecturas muy diversas. Dependiendo de cómo organicen el flujo de ejecución, del modelo computacional subyacente y de la estrategia lógica que empleen para atacar el problema, los algoritmos requieren ser sistematizados.

Este capítulo examina los principales criterios desarrollados por la informática para clasificar estas estructuras conceptuales. Se analizarán, en primer lugar, las diferencias en su naturaleza de ejecución (determinista frente a no determinista) y los entornos computacionales donde operan (secuenciales, paralelos y distribuidos). Posteriormente, se estudiarán los paradigmas clásicos de diseño algorítmico —como *divide y vencerás*, *programación dinámica* o *backtracking*— que constituyen los esquemas generales de resolución. Finalmente, se abordará la expansión del paradigma algorítmico clásico hacia el aprendizaje automático, donde la lógica formal y la recursividad se entrelazan con la estadística y la probabilidad para inferir conocimiento a partir de datos inciertos.

El propósito de este análisis es demostrar que los algoritmos no son meras secuencias aisladas de código, sino profundas estructuras conceptuales organizadas en torno a estrategias generales de cálculo. Comprender su clasificación y diseño es el paso previo e indispensable para adentrarse en el estudio de la complejidad computacional, donde se medirá el costo, la eficiencia y la viabilidad real de implementar estas estrategias.

5.1. Clasificación según la naturaleza de la ejecución

Un primer criterio de clasificación considera **la forma en que el algoritmo determina su comportamiento durante la ejecución**. Desde esta perspectiva, los algoritmos pueden clasificarse en **deterministas** y **no deterministas**.

Un **algoritmo determinista** produce siempre el mismo resultado para una misma entrada, siguiendo exactamente la misma secuencia de operaciones. Este tipo de algoritmos constituye la base de la mayoría de los programas utilizados en la práctica. Un ejemplo clásico es el algoritmo **Merge Sort**, que ordena una lista de elementos mediante una estrategia sistemática de división y combinación.

Por otro lado, un **algoritmo no determinista** es un modelo teórico en el cual el procedimiento computacional puede explorar múltiples alternativas de solución de manera conceptual. En este tipo de algoritmos se asume que el sistema puede elegir entre diversas opciones posibles en determinados pasos de la ejecución. Un ejemplo representativo es el algoritmo no determinista utilizado para resolver el **problema del camino hamiltoniano**, en

el cual se “adivina” una posible secuencia de vértices y posteriormente se verifica si constituye un camino válido en el grafo.

Aunque los algoritmos no deterministas no pueden implementarse directamente en computadoras convencionales, desempeñan un papel fundamental en la **teoría de la complejidad computacional**, particularmente en la definición de la clase de problemas **NP** (los problemas NP se explican en un capítulo posterior).

5.2. Clasificación según el modelo de ejecución

Además de la naturaleza lógica de la ejecución, los algoritmos también pueden clasificarse según el **modelo de ejecución**, es decir, la forma en que las instrucciones se llevan a cabo dentro de un sistema de cómputo.

Los **algoritmos secuenciales** ejecutan sus instrucciones una tras otra en un único flujo de control. Este modelo corresponde al funcionamiento tradicional de las computadoras basadas en la **arquitectura de von Neumann**. La mayoría de los algoritmos clásicos, como los algoritmos de ordenamiento o búsqueda, se diseñan inicialmente bajo este paradigma.

En contraste, los **algoritmos paralelos** permiten ejecutar múltiples operaciones simultáneamente utilizando varios procesadores o núcleos de procesamiento. Este enfoque resulta particularmente útil en problemas que pueden dividirse en tareas independientes. Un ejemplo representativo es la **multiplicación paralela de matrices**, ampliamente utilizada en aplicaciones científicas y en aprendizaje automático.

Además de estos modelos, los sistemas de cómputo modernos emplean **algoritmos distribuidos**, en los cuales el procesamiento se realiza en múltiples máquinas conectadas a través de una red. Este tipo de algoritmos es fundamental en sistemas de gran escala, como los centros de datos y las plataformas de **computación en la nube**.

Cabe señalar que la distinción entre algoritmos secuenciales, paralelos o distribuidos se refiere únicamente a la forma en que el algoritmo se ejecuta en el sistema de cómputo. Esta

clasificación no debe confundirse con la estructura interna del algoritmo. Por ejemplo, un algoritmo puede estar definido de manera **recursiva** —es decir, mediante llamadas a sí mismo— y, sin embargo, ejecutarse de forma secuencial, paralela o incluso distribuida dependiendo del entorno computacional. La recursión, por tanto, constituye una propiedad estructural del algoritmo y se analiza con mayor detalle dentro de los paradigmas de diseño algorítmico.

5.3. Clasificación según el paradigma de diseño

Más allá del entorno de ejecución, los algoritmos también pueden clasificarse según el **paradigma de diseño**, que describe la estrategia conceptual utilizada para construir la solución de un problema.

Los paradigmas algorítmicos constituyen **esquemas generales de resolución**, aplicables a una amplia variedad de problemas computacionales. Estos paradigmas permiten identificar patrones recurrentes en la construcción de algoritmos y facilitan el diseño de soluciones eficientes. Entre los paradigmas clásicos más importantes se encuentran en la tabla 3

Tabla 3 Paradigmas fundamentales de diseño algorítmico

Paradigma	Descripción	Ejemplos
Divide y vencerás	Descompone el problema en subproblemas independientes que se resuelven recursivamente y luego se combinan para obtener la solución global.	Merge Sort, QuickSort, algoritmo de Strassen
Programación dinámica	Divide el problema en subproblemas superpuestos y almacenas soluciones parciales para evitar recomputaciones.	Floyd–Warshall, Bellman–Ford, algoritmo de mochila
Voraz (Greedy)	Construye la solución paso a paso seleccionando en cada etapa la opción localmente óptima.	Algoritmo de Dijkstra, Kruskal, Prim
Backtracking	Explora sistemáticamente el espacio de soluciones posibles, retrocediendo cuando una solución parcial no conduce a una solución válida.	Problema de las N-reinas, Sudoku
Ramificación y acotamiento	Reduce el espacio de búsqueda mediante estimaciones que permiten descartar regiones completas del espacio de soluciones.	Problema del agente viajero (TSP), problemas de optimización entera

Algoritmos probabilísticos	Utilizan decisiones aleatorias durante la ejecución para mejorar el rendimiento promedio o simplificar el diseño.	QuickSort aleatorizado, algoritmo de Miller–Rabin
Algoritmos aproximados	Producen soluciones cercanas a la óptima cuando encontrar la solución exacta resulta computacionalmente costoso.	Aproximación para Vertex Cover, aproximación para TSP
Optimización matemática	Formulan el problema mediante funciones objetivo y restricciones matemáticas, resolviéndolo mediante algoritmos especializados.	Método Simplex, métodos de punto interior

Estos paradigmas constituyen herramientas fundamentales para el diseño de algoritmos eficientes y permiten abordar problemas complejos de manera sistemática.

5.4. Clasificación según el tipo de problema

Otro criterio de clasificación se basa en **la naturaleza del resultado que produce el algoritmo**.

Desde esta perspectiva es posible distinguir entre **algoritmos cuantitativos** y **algoritmos cualitativos**.

Los **algoritmos cuantitativos** se orientan a calcular valores numéricos, optimizar funciones objetivo o realizar simulaciones. Este tipo de algoritmos es común en problemas de optimización, cálculo científico y análisis numérico.

Por otro lado, los **algoritmos cualitativos** se centran en determinar propiedades estructurales o lógicas de los datos, como la validez sintáctica de una expresión o la pertenencia de una cadena a un lenguaje formal (Tabla 4).

Tabla 4 ***Clasificación de algoritmos según el tipo de problema***

Tipo	Descripción	Ejemplo
Cuantitativo	Calcula valores numéricos u optimiza funciones objetivo	Asignación de registros
Cualitativo	Determina propiedades estructurales o sintácticas	Análisis sintáctico

Esta distinción resulta particularmente útil en áreas como **compiladores, verificación de programas y lenguajes formales**.

5.5. Clasificación según la estrategia de búsqueda

En problemas de optimización combinatoria o exploración de espacios de soluciones complejos, los algoritmos también pueden clasificarse según **la estrategia utilizada para explorar el espacio de búsqueda**.

Los **algoritmos heurísticos** utilizan reglas aproximadas o intuiciones específicas del problema para encontrar soluciones razonables en tiempos computacionales aceptables. Aunque no garantizan siempre una solución óptima, suelen ser muy eficientes en la práctica.

Los **algoritmos metaheurísticos**, en cambio, constituyen estrategias generales que pueden aplicarse a una amplia variedad de problemas de optimización. Estos métodos guían el proceso de búsqueda mediante mecanismos inspirados en fenómenos naturales o procesos físicos (Tabla 5).

*Tabla 5**Clasificación de algoritmos según la estrategia de búsqueda***

Tipo	Descripción	Ejemplo
Heurístico	Utiliza reglas aproximadas para obtener soluciones aceptables	Nearest Neighbor
Metaheurístico	Estrategia general de búsqueda	Algoritmos genéticos, Simulated Annealing

Entre las metaheurísticas más conocidas se encuentran también **Particle Swarm Optimization** y **Ant Colony Optimization**.

5.6. Algoritmos en aprendizaje automático

El desarrollo histórico de la computación ha estado sustentado en distintos pilares matemáticos. En los capítulos anteriores se han examinado dos de ellos: por un lado, la **lógica formal**, que permitió definir rigurosamente el concepto de procedimiento mecánico, y por otro, la **teoría de la recursividad**, que proporcionó el marco matemático para caracterizar qué funciones puede calcularse mediante algoritmos. En las últimas décadas, un tercer componente ha adquirido una importancia creciente en la construcción de sistemas computacionales: **la probabilidad y la estadística**.

El auge de los datos masivos, la incertidumbre inherente a muchos fenómenos reales y la necesidad de construir sistemas capaces de adaptarse a información incompleta han impulsado el desarrollo de algoritmos que no se limitan a seguir reglas deterministas, sino que **aprenden patrones a partir de datos**. Este conjunto de métodos constituye el núcleo del **aprendizaje automático (machine learning)** y se utiliza ampliamente en tareas de clasificación, predicción, reconocimiento de patrones y análisis de grandes volúmenes de información.

A diferencia de los algoritmos clásicos, que ejecutan una secuencia fija de instrucciones definidas por el programador, los algoritmos de aprendizaje automático construyen **modelos matemáticos a partir de ejemplos observados**. El objetivo no es únicamente producir un resultado para una entrada dada, sino **inferir relaciones generales que permitan realizar predicciones sobre nuevos datos**.

Desde el punto de vista matemático, la mayoría de estos algoritmos se basan en **modelos estadísticos y probabilísticos**. En lugar de producir respuestas estrictamente deterministas, estos métodos estiman **distribuciones de probabilidad o parámetros estadísticos** que describen la relación entre las variables del problema. Este enfoque permite manejar situaciones en las que los datos contienen ruido, son incompletos o presentan variabilidad inherente.

Dentro de este marco probabilístico destacan los **modelos bayesianos**, fundamentados en el teorema de Bayes. Este principio permite actualizar la probabilidad de una hipótesis a medida que se dispone de nueva evidencia, proporcionando un mecanismo formal para el aprendizaje a partir de datos. Los algoritmos bayesianos han sido ampliamente utilizados en aplicaciones como el filtrado automático de correo no deseado, el diagnóstico médico asistido por computadora y diversos sistemas de inferencia probabilística.

Gracias a estos enfoques, el aprendizaje automático ha impulsado avances significativos en campos como la **visión por computadora, el procesamiento del lenguaje natural, los sistemas de recomendación y el análisis predictivo de datos**.

Entre los principales tipos de algoritmos utilizados en aprendizaje automático se observan en la tabla 6:

Tabla 6 ***Algoritmos de aprendizaje automático***

Tipo	Descripción	Ejemplo
Árboles de decisión	Modelos jerárquicos que representan reglas de decisión mediante estructuras de árbol	ID3, C4.5
Redes neuronales	Modelos inspirados en redes neuronales biológicas capaces de aprender representaciones complejas de los datos	Perceptrón, redes neuronales profundas
Máquinas de soporte vectorial	Métodos de clasificación que separan los datos mediante hiperplanos óptimos en espacios de alta dimensión	SVM
Modelos probabilísticos bayesianos	Algoritmos basados en el teorema de Bayes que permiten estimar probabilidades condicionales y realizar inferencias a partir de datos	Naive Bayes, Redes Bayesianas
Algoritmos evolutivos	Métodos de optimización inspirados en procesos de evolución biológica	Algoritmos genéticos

La incorporación de modelos probabilísticos introduce una diferencia fundamental respecto a los algoritmos deterministas tradicionales. Mientras que estos últimos ejecutan una secuencia fija de operaciones, los algoritmos de aprendizaje automático **aprenden a partir de los datos y estiman probabilidades o parámetros estadísticos**, lo que les permite adaptarse a situaciones complejas y variables.

En este sentido, el aprendizaje automático representa una expansión del paradigma algorítmico clásico. Si la lógica formal permitió definir la noción de procedimiento y la recursividad delimitó el alcance de lo computable, **la probabilidad proporciona el marco matemático para abordar problemas donde la información es incierta o incompleta**. De esta manera, la informática contemporánea integra estos tres pilares —lógica, recursividad y probabilidad— en la construcción de sistemas capaces no solo de ejecutar instrucciones, sino también de **inferir conocimiento a partir de los datos**.

- lógica → algoritmos
- recursividad → computabilidad

- **probabilidad → aprendizaje automático**

5.7. Clasificación según el modelo computacional

Finalmente, los algoritmos también pueden clasificarse según el **modelo computacional para el cual fueron diseñados**.

La mayoría de los algoritmos clásicos se desarrollan para computadoras convencionales basadas en la arquitectura de **Von Neumann**, donde el procesamiento se realiza de forma secuencial mediante una unidad central de procesamiento que interactúa con la memoria.

Sin embargo, el desarrollo de nuevas tecnologías ha dado lugar a **modelos computacionales alternativos** que exploran enfoques distintos para la resolución de problemas (Tabla 5.5).

Tabla 7 ***Algoritmos según la arquitectura computacional***

Arquitectura	Ejemplo de algoritmo	Enfoque
Computación clásica	Ordenamiento, grafos	Procesamiento secuencial
Computación cuántica	Shor, Grover	Superposición cuántica
Computación neuromórfica	Redes neuronales de picos	Paralelismo biológico
Computación con ADN	Experimento de Adleman	Paralelismo bioquímico
Computación óptica	Transformada de Fourier	Cálculo mediante luz

Estos modelos representan **nuevas fronteras de investigación en computación** y podrían ofrecer ventajas significativas para ciertos tipos de problemas.

Importancia del diseño de algoritmos

La clasificación de los algoritmos permite comprender la diversidad de estrategias existentes para resolver problemas computacionales. Cada paradigma de diseño proporciona herramientas conceptuales que permiten estructurar soluciones eficientes para distintos tipos de problemas.

Sin embargo, además del diseño conceptual, resulta fundamental analizar el **costo computacional asociado a la ejecución de un algoritmo**. Dos algoritmos pueden resolver

correctamente un mismo problema, pero diferir significativamente en el tiempo de ejecución o en la cantidad de memoria requerida.

Por esta razón, el estudio de los algoritmos se complementa con el **análisis de complejidad**, el cual permite evaluar formalmente el crecimiento de los recursos computacionales en función del tamaño de la entrada.

En las secciones siguientes se introducirán las herramientas formales utilizadas para este análisis, particularmente la **notación asintótica** y las **clases de complejidad computacional**, que constituyen la base del estudio moderno de la eficiencia algorítmica.

Capítulo VI. Algoritmos, su complejidad

El capítulo anterior presentó los principales criterios para clasificar y diseñar algoritmos, mostrando las distintas estrategias que pueden emplearse para resolver problemas computacionales. Sin embargo, conocer la estructura de un algoritmo no es suficiente para evaluar su utilidad práctica. Dos algoritmos pueden resolver correctamente el mismo problema y, aun así, diferir significativamente en la cantidad de recursos computacionales que requieren para ejecutarse.

Para abordar esta cuestión surge el **análisis de complejidad**, disciplina que estudia el comportamiento de los algoritmos en función de los recursos que consumen, principalmente **tiempo de ejecución** y **espacio de memoria**. En lugar de medir tiempos concretos dependientes del hardware, el análisis algorítmico se centra en el **crecimiento del número de operaciones necesarias conforme aumenta el tamaño de la entrada**.

La relación entre este capítulo y los fundamentos desarrollados previamente es directa. En el Capítulo II se introdujo la recursividad como una estructura matemática fundamental para definir procedimientos computables. Muchos algoritmos clásicos —como el algoritmo de Euclides, Merge Sort o las Torres de Hanoi— utilizan precisamente esta estructura recursiva. Analizar su complejidad permite comprender cómo un mismo principio matemático puede producir comportamientos computacionales muy distintos: desde algoritmos altamente eficientes hasta procesos cuyo crecimiento se vuelve rápidamente intratable.

A lo largo de este capítulo se introducirán las herramientas formales utilizadas para estudiar la eficiencia de los algoritmos y se aplicarán al análisis de varios ejemplos representativos, con el fin de mostrar cómo la complejidad computacional permite distinguir entre soluciones **teóricamente correctas** y soluciones **computacionalmente viables**.

6.1 Eficiencia y recursos computacionales

La eficiencia de un algoritmo se refiere a la cantidad de recursos computacionales que consume para completar una tarea. En el análisis de algoritmos, los dos recursos más relevantes —y generalmente más limitados— son **el tiempo de ejecución** y **el espacio de**

memoria. Comprender la relación entre estos recursos es fundamental para el diseño de soluciones computacionales eficientes y aplicables en contextos reales.

El estudio sistemático de estos recursos permite comparar distintos algoritmos que resuelven un mismo problema y determinar cuál de ellos resulta más adecuado según las restricciones del sistema.

6.2 Complejidad temporal y complejidad espacial

En el análisis de algoritmos, la eficiencia se describe mediante dos medidas principales:

Complejidad temporal.

Mide el número de operaciones elementales que un algoritmo ejecuta en función del tamaño de la entrada n. En lugar de medir tiempo en segundos —lo cual dependería de la velocidad del hardware o del lenguaje de programación—, el análisis se centra en el **número de pasos computacionales** que el algoritmo realiza.

Complejidad espacial.

Mide la cantidad de memoria adicional requerida por el algoritmo durante su ejecución, excluyendo el espacio ocupado por los datos de entrada. Este espacio puede incluir variables auxiliares, estructuras de datos adicionales y la pila de llamadas utilizada en algoritmos recursivos.

Trade-off entre tiempo y memoria

Un principio importante en el diseño de algoritmos es el **intercambio entre tiempo y espacio** (*time–space trade-off*). En muchos casos es posible reducir el tiempo de ejecución de un algoritmo utilizando más memoria, o bien disminuir el uso de memoria a costa de incrementar el tiempo de cálculo.
Este equilibrio entre recursos constituye una decisión de diseño que depende del contexto computacional en el que se ejecutará el algoritmo.

Ejemplo ilustrativo: encontrar el primer número repetido

Consideremos el problema de identificar el primer elemento que aparece repetido en un arreglo desordenado de números naturales.

Enfoque 1: minimizar el uso de memoria

Una estrategia consiste en comparar cada elemento del arreglo con todos los elementos que aparecen después de él utilizando dos bucles anidados.

- **Complejidad temporal:** en el peor caso se realizan aproximadamente $n^2/_2$comparaciones, lo que corresponde a una complejidad $O(n^2)$.
- **Complejidad espacial:** el algoritmo utiliza únicamente el arreglo original, por lo que el espacio adicional requerido es constante $O(1)$.

Enfoque 2: minimizar el tiempo de ejecución

Otra alternativa consiste en utilizar una estructura de datos auxiliar —por ejemplo, una tabla hash o un arreglo de marcas— para registrar los elementos que ya han sido observados.

- **Complejidad temporal:** el arreglo se recorre una sola vez y las operaciones de consulta o inserción en la estructura auxiliar tienen costo constante promedio, por lo que la complejidad es $O(n)$.
- **Complejidad espacial:** es necesario almacenar información adicional para cada elemento procesado, lo que implica un uso de memoria $O(n)$.

Este ejemplo ilustra claramente el **trade-off tiempo–espacio**: el segundo enfoque reduce significativamente el tiempo de ejecución, pero requiere mayor cantidad de memoria. La elección entre ambos algoritmos depende del contexto de aplicación: si el tiempo de ejecución es crítico y la memoria es abundante, se prefiere el segundo método; si la memoria disponible es limitada, podría ser necesario optar por el primero.

La dualidad de la recursión: elegancia y eficiencia

La recursión constituye una de las herramientas conceptuales más importantes de las ciencias de la computación. Su principal ventaja radica en la posibilidad de expresar soluciones

complejas mediante definiciones compactas basadas en la descomposición del problema en subproblemas de menor tamaño. Gracias a esta propiedad, muchos algoritmos pueden formularse de manera clara y elegante.

Sin embargo, esta misma característica puede ocultar costos computacionales significativos si el algoritmo no se analiza cuidadosamente. Una implementación recursiva que resulta conceptualmente simple puede generar un número muy elevado de llamadas recursivas o consumir grandes cantidades de memoria. Por esta razón, la recursión no debe considerarse únicamente como una técnica de programación, sino también como un escenario particularmente útil para estudiar los principios del análisis de complejidad.

1. Análisis de recursos: el costo de la pila de llamadas

A diferencia de los algoritmos iterativos, cada llamada recursiva requiere almacenar información adicional en la **pila de llamadas** (*call stack*) del sistema. Esta estructura conserva el estado de cada llamada, incluyendo variables locales, parámetros y la dirección de retorno.

Desde el punto de vista del análisis de complejidad:

- Si un algoritmo presenta una **profundidad máxima de recursión D**, entonces su **complejidad espacial adicional es O(D)**.

Por ejemplo, en el cálculo recursivo del factorial:

$$n! = n \cdot (n-1)!$$

cada llamada genera una nueva activación en la pila. Como la profundidad máxima es n, el consumo de memoria adicional crece linealmente, produciendo una **complejidad espacial O(n)**. En implementaciones con entradas muy grandes, esta acumulación puede provocar un **desbordamiento de pila (stack overflow)**.

Subproblemas superpuestos y explosión de complejidad

Otro riesgo frecuente en algoritmos recursivos es el **re-cálculo innecesario de subproblemas**, fenómeno conocido como **subproblemas superpuestos**.

Esto ocurre cuando un algoritmo resuelve repetidamente la misma instancia de un subproblema en diferentes ramas de la recursión.

Caso de estudio: la sucesión de Fibonacci

La definición recursiva directa de Fibonacci es:

$$f(n) = f(n-1) + f(n-2)$$

con condiciones iniciales $f(0) = 0 \;\; y \; f(1) = 1$.

Aunque esta formulación es matemáticamente simple, su implementación recursiva ingenua genera un árbol de llamadas altamente redundante. Muchos valores intermedios se recalculan múltiples veces, lo que provoca que el número total de operaciones crezca exponencialmente.

El análisis del árbol de recursión muestra que la complejidad temporal de este algoritmo es:

$$O(\varphi^n)$$

donde φ es el número áureo (φ≈1.618).

El número de llamadas recursivas del algoritmo ingenuo de Fibonacci crece exponencialmente. Para $n = 50$, el árbol de recursión contiene aproximadamente **40 mil millones de llamadas**, lo que vuelve impracticable este enfoque incluso en computadoras modernas. Este comportamiento ilustra claramente cómo una definición recursiva correcta desde el punto de vista matemático puede resultar computacionalmente ineficiente.

Este ejemplo ilustra un principio fundamental: **una formulación recursiva correcta desde el punto de vista matemático no garantiza una implementación eficiente**. El análisis de complejidad resulta indispensable para identificar estas situaciones y diseñar soluciones más eficientes, como las basadas en **programación dinámica** o **memorización**.

6.3 Herramienta de análisis: árboles de recursión

Los **árboles de recursión** constituyen una herramienta conceptual útil para analizar el comportamiento de los algoritmos recursivos. Este método permite representar gráficamente el proceso de descomposición del problema y estimar el costo total del algoritmo.

En este modelo, cada llamada recursiva se representa como un nodo del árbol:

- **Nodos (costo temporal):** el número total de nodos del árbol corresponde al número total de llamadas u operaciones realizadas durante la ejecución del algoritmo.
- **Altura del árbol (costo espacial):** la longitud máxima de una rama desde la raíz hasta una hoja representa la **profundidad máxima de la recursión**, lo que determina el uso máximo de memoria en la pila de llamadas.

Este enfoque resulta especialmente útil para visualizar cómo el trabajo computacional se distribuye entre los distintos niveles de la recursión.

Metodología para el análisis de algoritmos

Para analizar un algoritmo de manera sistemática es necesario abstraerse de los detalles de implementación y del hardware utilizado. El objetivo es estudiar el **crecimiento del costo computacional** en función del tamaño de la entrada.

Análisis a priori (teórico).

Se realiza sobre la descripción del algoritmo —generalmente expresada en pseudocódigo— y consiste en contar el número de operaciones fundamentales que se ejecutan en función del tamaño de la entrada n. El resultado es una función matemática $T(n)$ que describe el crecimiento del tiempo o del espacio requerido. Este análisis es independiente del lenguaje de programación, del compilador y de la arquitectura de la máquina.

Análisis a posteriori (empírico).

Se realiza ejecutando una implementación concreta del algoritmo sobre conjuntos de datos específicos y midiendo el tiempo real de CPU o el uso de memoria. Este enfoque permite

validar el análisis teórico, estimar constantes ocultas y comparar distintas implementaciones en un entorno particular. Sin embargo, sus resultados dependen del hardware, del sistema operativo y del compilador utilizados.

En teoría de algoritmos, el **análisis a priori** constituye el enfoque fundamental, ya que proporciona resultados generales sobre el comportamiento del algoritmo.

Operaciones de costo constante y medida del tamaño de la entrada

El primer paso del análisis a priori consiste en identificar qué operaciones pueden considerarse de **costo constante O(1)**.

Generalmente se incluyen en esta categoría:

- operaciones aritméticas básicas (+,−,∗,/,%)
- comparaciones lógicas (<,>,==,!=)
- accesos a memoria (lectura o escritura de una variable o de un elemento de un arreglo indexado)
- llamadas a funciones cuyo costo es conocido y constante

Otro aspecto fundamental es definir correctamente el parámetro n, que representa el **tamaño de la entrada**. Este parámetro depende de la naturaleza del problema.

Por ejemplo:

- **Arreglo o lista:** n representa el número de elementos.
- **Matriz cuadrada:** n puede representar el número de filas (o columnas).
- **Grafo:** n suele representar el número de vértices, aunque en algunos algoritmos el número de aristas E resulta más relevante.
- **Número entero:** en algoritmos numéricos, n puede referirse al valor del número o, más comúnmente, al número de bits necesarios para representarlo $(Log_2\ n)$.

Definir adecuadamente el parámetro n es esencial para realizar un análisis correcto de la complejidad.

Cálculo de $T(n)$: descomposición estructural

Los algoritmos estructurados se construyen a partir de un conjunto básico de **estructuras de control**. El costo total de un algoritmo puede estimarse sumando el costo de cada una de estas estructuras.

Sentencias simples

Una operación elemental de costo constante (por ejemplo, una asignación o una operación aritmética) contribuye con O(1) al tiempo total.

Secuencia de instrucciones

Cuando varias instrucciones se ejecutan en secuencia, el costo total es la suma de los costos individuales:

$$T_{sec}(n) = T_1 + T_2 + \cdots$$

Estructuras condicionales

En una estructura **if–then–else**, el costo corresponde al de evaluar la condición más el costo de la rama más costosa:

$$T_{if} = T_{condicion} + máx\{T_{then}(n), T_{else}(n)\}$$

Bucles

El costo de un bucle depende del número de iteraciones.

Bucle simple

```
for (i = 0; i < n; i++)
{
   // operación O(1)
}
```

Se ejecuta n veces, por lo que su complejidad es:

$$O(n)$$

Bucles anidados

```
for (i = 0; i < n; i++)
{
   for (j = 0; j < n; j++)
   {
      // operación O(1)
   }
}
```

Se ejecutan $n \cdot n$ iteraciones, por lo que la complejidad es:

$$O(n^2)$$

Bucle con crecimiento multiplicativo

```
while (c < n)
{
   c = 2c;
}
```

El valor de c sigue la secuencia:

$$1,2,4,8,\ldots$$

hasta superar n. El número de iteraciones es:

$$Log_2\, n$$

por lo que la complejidad es:

$$O(Log_2\, n)$$

Relaciones de recurrencia

En algoritmos recursivos, la complejidad temporal suele expresarse mediante una **relación de recurrencia**, que define el tiempo requerido para resolver un problema de tamaño n en función del tiempo necesario para resolver instancias más pequeñas.

Método de expansión (o sustitución)

Una técnica común para resolver recurrencias consiste en expandir la relación repetidamente hasta detectar un patrón y luego resolver para la condición base.

Ejemplo: Merge Sort

La recurrencia del algoritmo es:

$$T(n) = 2T(n/2) + n$$

con condición base $T(1) = 1$.

Para simplificar el análisis se asume que $n = 2^k$.

1. **Expansión**

$$T(n)=2[2T(n/4) +n/2]+n$$

2. **Patrón general**

Después de k expansiones:

$$T(n)=2^kT(n/2^k)+kn$$

3. **Condición de parada**

La recursión se detiene cuando:

$$n/2^k = 1$$

lo que implica:

$$k = Log_2\, n$$

4. **Sustitución**

$$T(n) = nT(1) + nlog_2 n$$

5. **Resultado**

$$T(n) = \Theta(nlog_2 n)$$

En la tabla 6.1 se dan las relaciones de recurrencia más comunes.

Tabla 6.1 Relaciones de recurrencia comunes

Relación de Recurrencia	Complejidad Temporal (Solución)	Ejemplo de Algoritmo
$T(n) = T(n/2) + O(1)$	**$O(\text{Log } n)$**	Búsqueda Binaria
$T(n) = T(n-1) + O(1)$	**$O(n)$**	Recorrido Lineal, Factorial no recursivo
$T(n) = 2T(n/2) + O(1)$	**$O(n)$**	Recorrido de un árbol binario
$T(n) = 2T(n/2) + O(n)$	**$O(n \text{ Log } n)$**	Merge Sort, QuickSort (caso promedio)
$T(n) = T(n-1) + O(n)$	**$O(n^2)$**	Ordenación por Inserción (peor caso), Selección
$T(n) = 2T(n-1) + O(1)$	**$O(2^n)$**	Torres de Hanói, Fibonacci ingenuo

La diferencia entre estos órdenes de crecimiento ilustra cómo decisiones aparentemente pequeñas en el diseño de un algoritmo pueden producir comportamientos computacionales radicalmente distintos

6.4 Notación asintótica y órdenes de complejidad

Una vez establecido cómo calcular el tiempo de ejecución T(n) de un algoritmo, surge una dificultad práctica: las expresiones exactas suelen ser complejas y dependen de constantes específicas del hardware, del lenguaje de programación y de la implementación concreta. Para comparar algoritmos de manera general y rigurosa es necesario utilizar un lenguaje que capture únicamente **la tasa de crecimiento de las funciones cuando el tamaño de la entrada n se vuelve grande**. Este es precisamente el objetivo de la **notación asintótica**.

Notación asintótica: comparando tasas de crecimiento

Motivación: comportamiento en el límite

En muchos casos conocer la función exacta T(n) no es lo más útil para analizar un algoritmo. Aunque dichas funciones describen con precisión el número de operaciones realizadas, contienen constantes y términos de bajo orden que complican la comparación entre distintos algoritmos.
Por ejemplo, las funciones

$$T_1(n) = 3n^2 + 2n + 100$$

y

$$T_2(n) = 7n^2 + 50n$$

son diferentes en detalle, pero ambas presentan el mismo comportamiento dominante cuando n crece: el término cuadrático domina completamente el crecimiento.

Desde esta perspectiva, ambas funciones crecen esencialmente como

$$n^2$$

Esta observación conduce a una idea central del análisis de algoritmos: **no nos interesa el valor exacto de T(n), sino su comportamiento cuando n tiende a infinito**.

Para formalizar esta idea se utiliza la **notación asintótica**, que describe las tasas de crecimiento mediante **cotas superiores, inferiores y ajustadas**.

Cota superior asintótica: notación O (Big-O)

La notación O(g(n)) describe un **límite superior** para el crecimiento de una función.

Definición formal

$$T(n)=O(g(n))$$

si existen constantes positivas c y n_0 tales que

$$0 \leq T(n) \leq c \cdot g(n) \text{ para todo } n \geq n_0$$

Interpretación intuitiva

La función T(n) crece **a lo sumo tan rápido como g(n)**, salvo por un factor constante.
En análisis de algoritmos, la notación O suele utilizarse para describir el **peor caso** de un algoritmo.

Ejemplos

$$\boldsymbol{3n^2 + 2n + 1000 \equiv O(n^2)}$$

$$\boldsymbol{5n + 10 \equiv O(n)}$$

$$\boldsymbol{1000 \equiv O(1)}$$

$$\boldsymbol{nLog\ n + n \equiv O(nLog\ n)}$$

En la práctica, para obtener la notación O se conserva **el término dominante** de la función y se ignoran constantes multiplicativas.

Cota inferior asintótica: notación Ω

Mientras que O(g(n)) establece un límite superior, la notación Ω(g(n)) describe un **límite inferior** del crecimiento.

Definición formal

T(n)=Ω(g(n)) si existen constantes positivas c y n_0n0n_0n0 tales que

$$0 \leq c \cdot g(n) \leq T(n) \text{ para todo } n > n_0$$

Interpretación intuitiva

La función T(n) crece **al menos tan rápido como g(n)**.

Ejemplos

- Cualquier algoritmo que deba leer una entrada de tamaño n tiene complejidad al menos Ω(n)
- En la búsqueda lineal en un arreglo no ordenado, el mejor caso es Ω(1) cuando el elemento buscado aparece en la primera posición.

Cota ajustada: notación Θ

La notación Θ(g(n)) proporciona una **cota ajustada** para el crecimiento de una función.

Definición formal

T(n)=Θ(g(n)) si existen constantes positivas c_1, c_2 y n_0 tales que

$$0 \leq c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot g(n)$$

para todo $n > n_0$

Interpretación intuitiva

La función T(n) crece **exactamente al mismo ritmo que g(n)**, salvo por constantes multiplicativas.

Si T(n)=Θ(g(n))
entonces automáticamente T(n)=O(g(n)) y T(n)=Ω(g(n))

Ejemplo

El algoritmo para encontrar el máximo en un arreglo realiza siempre n−1n-1n−1 comparaciones, por lo que T(n)=Θ(n)

La búsqueda binaria tiene complejidad

$$\Theta(log_2\, n)$$

La figura 6.1 muestra cómo las funciones c·g(n) encierran a T(n), ilustrando las definiciones de (a) O (cota superior), (b) Ω (cota inferior) y (c) Θ (cota ajustada).

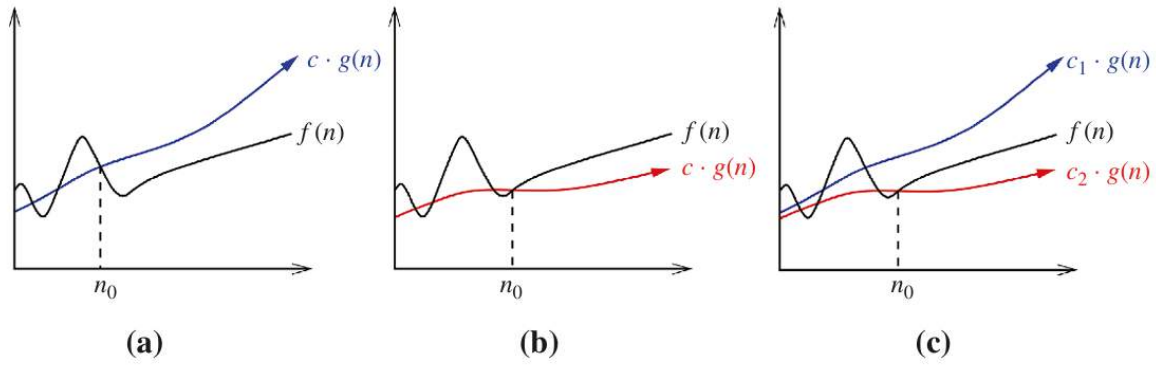

Figura 6.1: Interpretación gráfica de la notación asintótica.

Órdenes de complejidad comunes

Las funciones utilizadas en la notación asintótica no son arbitrarias. Un conjunto relativamente pequeño describe la mayoría de los comportamientos algorítmicos.

La figura 6.2 ilustra de manera dramática la diferencia entre un crecimiento polinomial "tratable" (n^2, n^3) y uno exponencial o factorial "intratable" 2^n

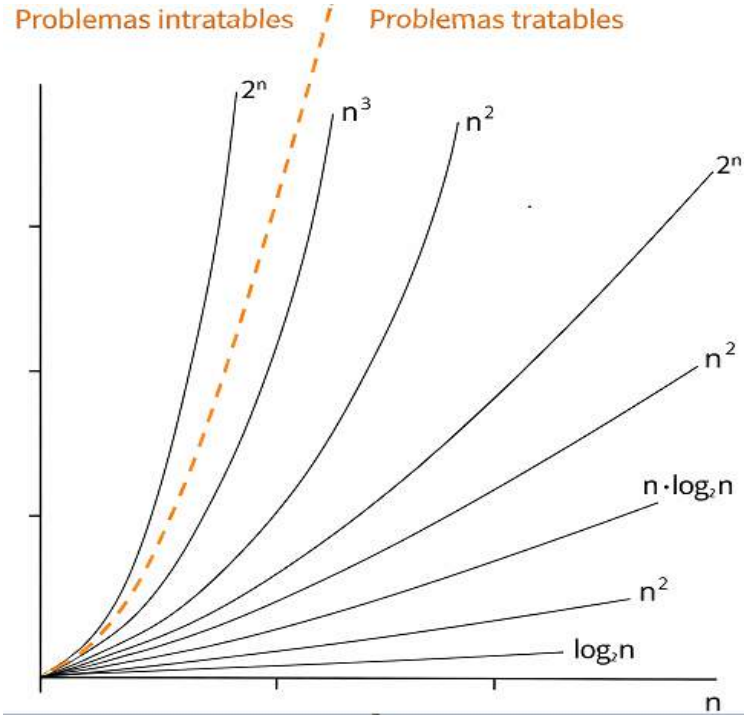

Figura 6.2: Gráfica comparativa del crecimiento de funciones comunes (Báez, s. f.)

En términos prácticos, los algoritmos **polinomiales** suelen considerarse tratables, mientras que los algoritmos **exponenciales o factoriales** se vuelven rápidamente imprácticos a medida que crece el tamaño de la entrada

6.5 Implicaciones prácticas y la frontera de la tratabilidad

La jerarquía de órdenes de complejidad no constituye únicamente una clasificación matemática abstracta. En la práctica, establece **límites claros sobre qué problemas pueden resolverse de manera eficiente mediante algoritmos** y cuáles se vuelven rápidamente impracticables a medida que crece el tamaño de la entrada.

Desde esta perspectiva, los problemas computacionales suelen distinguirse en tres grandes categorías.

Problemas tratables.

Se consideran tratables aquellos problemas para los cuales se conoce al menos un algoritmo cuya complejidad temporal es **polinomial**, es decir, $O(n^c)$ para alguna constante c. Aunque el exponente pueda ser relativamente grande, el crecimiento de estos algoritmos permanece controlado a medida que aumenta el tamaño de la entrada. En teoría de la complejidad, este conjunto de problemas se asocia con la **clase P** y suele interpretarse como el conjunto de problemas que pueden resolverse eficientemente mediante algoritmos.

Problemas intratables.

Existen problemas para los cuales los mejores algoritmos conocidos presentan complejidad **exponencia**l $O(c^n)$ o **factorial** $O(n!)$. En estos casos, incluso incrementos moderados en el tamaño de la entrada producen tiempos de ejecución extremadamente grandes. Por ejemplo, algoritmos con complejidad exponencial pueden requerir años o siglos de cómputo para instancias relativamente pequeñas. Muchos problemas de optimización combinatoria —como el Problema del Agente Viajero o ciertos problemas de planificación— pertenecen a esta categoría.

Problemas no computables.

Finalmente, existen problemas para los cuales **no puede existir ningún algoritmo que los resuelva para todas las entradas posibles**. Estos problemas no están limitados por recursos computacionales, sino por restricciones lógicas fundamentales. El ejemplo clásico es el **Problema de la Parada**, demostrado indecidible por Alan Turing en 1936.

Hasta este punto del libro hemos analizado **algoritmos individuales** y estudiado cómo medir su eficiencia. La teoría de la complejidad computacional avanza un paso más: en lugar de analizar algoritmos concretos, **clasifica los problemas computacionales según los recursos mínimos que cualquier algoritmo necesitaría para resolverlos**.

Esta perspectiva permite comprender que ciertas dificultades no dependen de la habilidad del programador ni de la potencia de la computadora, sino de **propiedades intrínsecas de los propios problemas**. El estudio de estas fronteras —entre lo tratable, lo intratable y lo no computable— constituye uno de los temas centrales de la teoría de la computación moderna y será abordado en el capítulo siguiente.

Capítulo VII. Clases de Complejidad de Problemas

Para estudiar la dificultad intrínseca, nos centramos en un tipo especial de problema: los **problemas de decisión**. Un problema de decisión es aquel cuya respuesta es simple: **SÍ** o **NO**. Ejemplos: "¿Existe un camino entre los nodos A y B en este grafo?" o "¿Tiene esta fórmula booleana una asignación que la haga verdadera?". Esta simplificación permite un análisis matemático más limpio, y muchos problemas de optimización o búsqueda pueden reformularse como problemas de decisión equivalentes.

7.1 La Clase P (Tiempo Polinomial Determinista)

Definición: La clase **P** contiene todos los problemas de decisión para los cuales existe un **algoritmo determinista** que los resuelve en un tiempo acotado por una **función polinomial** del tamaño de la entrada. Es decir, un tiempo $O(n^C)$ para alguna constante c.

Interpretación: P representa los problemas que consideramos tratables o eficientemente resolubles en la práctica. Un algoritmo polinomial, aunque pueda ser lento para c grande, garantiza que el tiempo de cómputo no se disparará de manera inmanejable al aumentar el tamaño de la entrada.

Ejemplos:

- ¿Está ordenada esta lista?
- ¿Son estos dos números primos entre sí? (Algoritmo de Euclides)
- ¿Existe un camino entre dos nodos en un grafo? (Búsqueda BFS/DFS)
- ¿Tiene este grafo un ciclo Euleriano?

7.2 La Clase NP (Tiempo Polinomial No Determinista)

Definición 1 (Verificabilidad): La clase **NP** (Polinomial No Determinista) contiene los problemas de decisión para los cuales, dada una **solución candidata** (llamada *certificado* o *testigo*), podemos **verificar** si es correcta en tiempo polinomial con un algoritmo determinista.

Definición 2 (Resolubilidad no determinista): Equivalente a la anterior, NP contiene los problemas que podrían ser resueltos en tiempo polinomial por una **Máquina de Turing No**

Determinista teórica, una máquina que puede "adivinar" la solución correcta y bifurcarse en múltiples caminos de cómputo simultáneamente.

Interpretación: NP representa los problemas cuyas **soluciones son fáciles de verificar**, aunque encontrar esa solución desde cero pueda ser extremadamente difícil. El nombre a veces lleva a confusión: **NP NO significa "No Polinomial"**. Es la clase de problemas "Polinomiales en una máquina No determinista" o "Polinomialmente verificables".

Ejemplos (fácil verificar, difícil encontrar):

- **Problema del Ciclo Hamiltoniano:** Dado un grafo, ¿existe un ciclo que visite cada nodo exactamente una vez? *Verificar:* Dado un ciclo candidato, es fácil comprobar en tiempo lineal si visita todos los nodos una sola vez.
- **SAT (Satisfacibilidad Booleana):** Dada una fórmula lógica, ¿existe una asignación de valores de verdad a sus variables que la haga verdadera? *Verificar:* Dada una asignación candidata, es fácil evaluar la fórmula en tiempo lineal.
- **Problema de la Mochila (decisión):** Dado un conjunto de objetos con pesos y valores, y una capacidad máxima, ¿existe un subconjunto de objetos cuyo peso total no exceda la capacidad y cuyo valor total sea al menos K? *Verificar:* Dado un subconjunto candidato, es fácil sumar pesos y valores.

Relación entre P y NP: Por definición, todo problema en **P** está también en **NP**. Si puedes resolver un problema en tiempo polinomial, ciertamente puedes verificar una solución candidata en tiempo polinomial (simplemente resuélvelo y compara). Por lo tanto: **P ⊆ NP**. La gran pregunta abierta es si esta contención es estricta o no.

7.3 NP-Completo y reducciones polinomiales

La clase NP contiene problemas cuyas soluciones pueden verificarse en tiempo polinomial. Sin embargo, dentro de NP existen problemas que representan el núcleo de dificultad de toda la clase.

Para formalizar esta noción se introduce el concepto de reducción polinomial.

Definición (Reducción polinomial)

Sean $L_1, L_2 \subseteq \Sigma^*$ dos lenguajes.

Decimos que L_1 **se reduce polinomialmente a** L_2, y lo denotamos por:

$$L_1 \leq_p L_2$$

si existe una función $f: \Sigma^* \to \Sigma^*$ tal que:

1. La función f es computable en tiempo polinomial.
2. Para toda cadena $x \in \Sigma^*$, se cumple que:

$$x \in L_1 \iff f(x) \in L_2$$

En este caso, se dice que L_1 no es más difícil que L_2. Por tanto, si L_2 puede resolverse en tiempo polinomial, entonces L_1 también puede resolverse en tiempo polinomial.

Definición (NP-Hard)

Un lenguaje L es **NP-Hard** si satisface que:

$$\forall L' \in \text{NP}, L' \leq_p L$$

Es decir, todo problema en la clase NP puede reducirse a L mediante una reducción polinomial. En este sentido, L es al menos tan difícil como cualquier problema en NP.

Definición (NP-Completo)

Un lenguaje $L \subseteq \Sigma^*$ es **NP-Completo** si cumple simultáneamente:

$$L \in \text{NP y} \forall L' \in \text{NP}, L' \leq_p L$$

Los problemas NP-completos constituyen el subconjunto más difícil dentro de la clase NP. En particular, si alguno de estos problemas pudiera resolverse en tiempo polinomial, entonces se tendría que:

$$\text{P} = \text{NP}$$

El primer problema demostrado NP-completo fue el problema de satisfacibilidad booleana (SAT), a partir del teorema de Cook–Levin (1971), lo que marcó el inicio de la teoría estructural de la complejidad computacional.

Observación estructural

Es importante destacar que no todo lenguaje NP-Hard es necesariamente NP-Completo. Para que un lenguaje sea NP-completo debe satisfacer dos condiciones: pertenecer a NP y ser NP-Hard. Sin embargo, existen lenguajes que, aun siendo al menos tan difíciles como cualquier problema en NP, no pertenecen a dicha clase.

En particular, un problema puede ser NP-Hard sin ser NP-Completo si no es decidible en tiempo polinomial no determinista, o incluso si no es decidible en absoluto.

De esta manera, se cumple la inclusión estricta:

$$\text{NP-Completo} \subseteq \text{NP-Hard}$$

pero la recíproca no es válida.

Un ejemplo ilustrativo es el **Problema de la Parada**, el cual es NP-Hard bajo ciertas nociones de reducción, pero no pertenece a NP, ya que es indecidible.

7.4. NP-Completitud y el legado de Cook

La teoría de la NP-completitud constituye uno de los pilares de la complejidad computacional moderna. Introducida a partir del teorema de Cook–Levin (1971), esta teoría permite identificar problemas que, aun siendo verificables eficientemente, parecen resistirse a toda solución algorítmica eficiente.

Para comprender este marco conceptual, es fundamental distinguir entre los problemas **NP-Hard** y los problemas **NP-Completo**, así como analizar la dinámica de reducción que los vincula.

1. Definiciones formales

NP-Hard (NP-Difícil):

Un problema H es NP-Hard si satisface la siguiente condición de universalidad:

$$\forall A \in \text{NP}, A \leq_p H$$

Es decir, todo problema en NP puede reducirse polinomialmente a H. En consecuencia, Hes al menos tan difícil como cualquier problema en NP.

NP-Completo (NPC):

Un problema L es NP-Completo si cumple simultáneamente:

1. $L \in$ NP (es verificable en tiempo polinomial).

2. L es NP-Hard.

Los problemas NP-completos representan los problemas más difíciles dentro de la clase NP.

2. Diferencia clave: dinámica de reducción

La distinción entre NP-Completo y NP-Hard se comprende mejor analizando el comportamiento de las reducciones polinomiales (Ver tabla 8).

Tabla 8. ***Relación entre problemas NP-Completo y NP-Hard***

Situación	Relación	Explicación
Entre dos problemas NP-Completo	Equivalencia total $A \leq_p B$ y $B \leq_p A$	Todos los problemas NP-completos son computacionalmente equivalentes. Resolver uno en tiempo polinomial implica resolver todos.
De NP-Completo a NP-Hard	Unidireccional $A \leq_p H$, pero no necesariamente $H \leq_p A$	Un problema NP-Hard puede ser más general (optimización) o incluso indecidible, lo que impide su reducción a problemas en NP.

3. Comparación estructural

La diferencia entre ambas clases puede analizarse también en términos de decidibilidad, formato y significado computacional (ver tabla 9).

Tabla 9. ***Comparación entre NP-Completo y NP-Hard***

Característica	NP-Completo (NPC)	NP-Hard
¿Pertenece a NP?	Sí	No necesariamente
¿Es decidible?	Sí, siempre	Variable: puede ser decidible o indecidible
Tipo de problema	Decisión (SÍ/NO)	Decisión, optimización o búsqueda
Significado	Problemas más difíciles dentro de NP	Problemas al menos tan difíciles como NP

4. Ejemplos ilustrativos

Los ejemplos de la tabla 10 permiten visualizar la distinción entre ambas clases:

Tabla 10 ***Clasificación de problemas según su complejidad***

Problema	Clase	Justificación
SAT (Satisfacibilidad)	NP-Completo	Pertenece a NP y todo problema en NP se reduce a él (Teorema de Cook–Levin)
TSP (versión de decisión)	NP-Completo	Pregunta: "¿Existe una ruta menor a X?"; verificable en tiempo polinomial
TSP (optimización)	NP-Hard	Busca la ruta mínima; no es problema de decisión
Problema de la Parada	NP-Hard	Es indecidible; no existe algoritmo general que lo resuelva

---

Observación conceptual

La distinción entre NP-Completo y NP-Hard revela que la dificultad computacional no depende únicamente del costo de ejecución, sino también de la naturaleza del problema. Mientras que los problemas NP-completos son difíciles pero verificables, los problemas NP-Hard pueden incluso exceder los límites de la computación algorítmica.

En este sentido, la NP-completitud no solo identifica problemas difíciles, sino que delimita una frontera fundamental entre lo eficientemente resoluble y lo intrínsecamente complejo.

7.5 El Teorema de Cook-Levin

El trabajo seminal de **Stephen Cook (1971)** y **Leonid Levin** estableció el primer pilar de la NP-Completitud.

- **Teorema:** El problema de **Satisfacibilidad Booleana (SAT)** es NP-Completo.
- **Significado:** Demostraron que SAT es un "lenguaje universal". Cualquier problema en NP puede codificarse como una fórmula booleana. Por tanto, debido a la propiedad de inter-reducibilidad, si encontráramos un algoritmo rápido para SAT, tendríamos automáticamente un algoritmo rápido para *todos* los problemas de NP.

7.6 La Pregunta P vs. NP

Es la incógnita abierta más importante en ciencias de la computación ("Problema del Milenio").

- **La Pregunta:** ¿Es P = NP? En otras palabras, ¿es la dificultad de *encontrar* una solución igual a la dificultad de *verificarla*?
- **Implicaciones si P = NP:** Sería revolucionario. Dado que todos los problemas NP-Completos son reducibles entre sí, resolver uno solo en tiempo polinomial colapsaría toda la clase NP dentro de P. La criptografía moderna caería, pero la optimización sería trivial.
- **Consenso:** La mayoría de la comunidad científica cree que **P ≠ NP**. La intuición sugiere que la creatividad necesaria para hallar una solución es fundamentalmente más compleja que la mecánica de verificarla.

7.7 Otras Clases de Complejidad

El panorama es más rico que P y NP. Otras clases importantes son:

- **Co-NP:** La clase de problemas cuyo **complemento** (cambiar la respuesta SÍ/NO) está en NP. Ejemplo: TAUTOLOGÍA (¿es una fórmula booleana verdadera para *todas* las asignaciones?) es el complemento de SAT.
- **PSPACE:** Problemas resolubles con una cantidad de **memoria (espacio)** polinomial. Se sabe que **P ⊆ NP ⊆ PSPACE**. Se cree que las inclusiones son estrictas.
- **EXPTIME:** Problemas resolubles en tiempo exponencial $O(2^{(p(N))})$. Contiene estrictamente a PSPACE.

En la figura 1 se muestra las clases de complejidad bajo dos hipótesis principales.

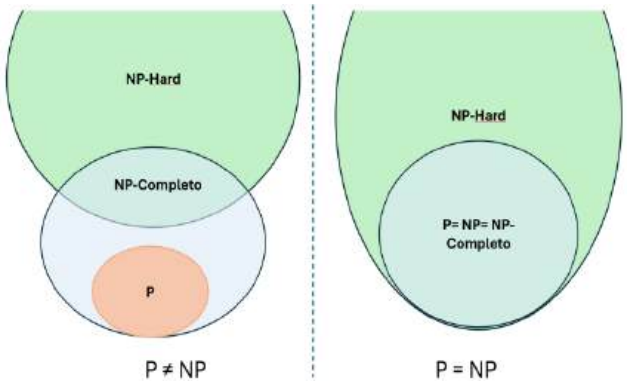

Figura 1 Relación entre las clases de complejidad bajo las dos hipótesis principales.

7.8 Reflexiones Finales: Límites y Herramientas

Dificultad Matemática vs. Dificultad Algorítmica

Es crucial distinguir estos dos conceptos, a menudo confundidos:

- **Dificultad Matemática:** Se refiere a la complejidad conceptual de **demostrar un enunciado general**. Ejemplo: El **Último Teorema de Fermat (UTF)** tomó siglos y matemáticas avanzadas para demostrar que $a^n + b^n = c^n$ no tiene soluciones enteras para $n>2$.
- **Dificultad Algorítmica:** Se refiere al costo computacional de **resolver instancias concretas**. Ejemplo: El problema de decisión UTF ("Dados a, b, c, n, ¿cumplen $a^n + b^n = c^n$?") está en **P**. Para una instancia concreta, basta calcular las potencias y comparar, operación polinomial en el número de dígitos.

Conclusión clave: Un problema puede ser **matemáticamente profundo,** pero **algorítmicamente trivial** (UTF), y otro puede ser **matemáticamente simple** de formular pero **algorítmicamente duro** (SAT).

El Papel de la Inteligencia Artificial

La IA moderna, especialmente el aprendizaje automático, no cambia los límites fundamentales establecidos por la teoría de la complejidad, pero sí altera drásticamente el panorama práctico.

- Para Problemas en P:
  La IA puede ofrecer implementaciones optimizadas o adaptativas, descubriendo patrones para acelerar el cálculo, pero no cambia la clasificación fundamental; estos problemas ya eran tratables.
- Para Problemas NP-Completos (como SAT) y NP-Hard Computables (como TSP):
  Aquí es donde la IA actúa como una herramienta invaluable. Dado que los problemas NP-Completos son los más difíciles dentro de la clase NP y equivalentes entre sí, encontrar un algoritmo polinomial exacto para uno resolvería todos (probando P=NP), algo que la IA no hace.
  Sin embargo, la IA transforma la viabilidad práctica:

- Enfoque en el caso promedio: La teoría de complejidad suele basarse en el "peor caso" posible. La IA explota la estructura estadística de los datos del mundo real, que raramente son tan caóticos como el peor escenario teórico.
- Aproximación inteligente: Utiliza heurísticas avanzadas, redes neuronales para predicción de ramificaciones y métodos de búsqueda local (como recocido simulado o algoritmos genéticos). Estos no garantizan la solución *perfecta* u óptima en todos los casos, pero encuentran soluciones de alta calidad en tiempo razonable, reduciendo la dificultad práctica aunque la barrera teórica permanezca intacta.

- Para Problemas No Computables o NP-Hard Indecidibles:
  La IA no puede superar la barrera lógica. Para el Problema de la Parada o variantes indecidibles, la IA solo puede hacer conjeturas probabilísticas o funcionar bien en casos particulares muy restringidos; nunca podrá garantizar una solución correcta para todas las entradas posibles. **Su papel aquí es el de un explorador inteligente, no de un oráculo infalible.**

La teoría de la complejidad computacional trasciende los algoritmos individuales para clasificar problemas. Las clases **P**, **NP**, **NP-Completo** y **NP-hard** definen el paisaje de lo que podemos resolver eficientemente, lo que podemos verificar eficientemente, y lo que es fundamentalmente difícil incluso para los verificadores más potentes.

1. Más allá de la intratabilidad práctica, encontramos la **no computabilidad**: problemas como el de la parada que ningún algoritmo puede resolver, recordándonos que existen límites formales al poder de la computación.
2. La pregunta **P vs. NP** sigue siendo el mayor enigma abierto, una cuestión que toca los fundamentos de las matemáticas, la creatividad y la naturaleza misma de la resolución de problemas.
3. Herramientas como la Inteligencia Artificial no invalidan estos límites teóricos, pero nos equipan para navegar de manera práctica y creativa el vasto territorio de los problemas complejos, encontrando soluciones útiles donde las soluciones perfectas son inalcanzables.

Comprender estos principios no es un ejercicio académico abstracto. Es un **marco de pensamiento esencial** para cualquier ingeniero, científico o tomador de decisiones en la era digital. Nos dota del criterio para saber cuándo un problema puede ser vencido con ingenio algorítmico, cuándo debemos conformarnos con aproximaciones, y cuándo nos enfrentamos a un muro lógico infranqueable. En última instancia, el estudio de la complejidad es un recordatorio de humildad y poder a la vez: humildad ante los límites de la computación, y poder por el vasto dominio que, dentro de esos límites, podemos conquistar.

Capítulo VIII. De lo eficiente a lo imposible: un recorrido por la complejidad algorítmica.

El cálculo de la complejidad algorítmica es una tarea matemática muy interesante, se requiere comprensión plena del problema y destreza matemática que permita llegar a la complejidad real del algoritmo. En este capítulo se ha optado por mostrar únicamente el cálculo de la complejidad computacional de cinco algoritmos representativos: Euclides para MCD recursivo, Merge Sort, Torres de Hanoi, cálculo del Determinante por cofactores y la Función de Ackermann. La elección de estos algoritmos no es arbitraria, sino que responde a criterios didácticos, teóricos y pedagógicos alineados con los objetivos de la enseñanza del cómputo. Como se observará en el cálculo de los ejemplos escogidos, la complejidad temporal va desde algoritmos tratables (como MCD Y Merge Sort) tanto intratables como las Torres de Hanoi y cálculo del Determinante por cofactores. Y un caso especial de algoritmo intratable "La función de Ackermann".

Cada uno de ellos permite ilustrar de manera clara y progresiva los distintos niveles de dificultad computacional, así como el papel de la recursividad y la eficiencia algorítmica en la enseñanza de la informática.

1. **Euclides MCD recursivo:** el algoritmo de Euclides para calcular el máximo común divisor (MCD) es uno de los ejemplos más antiguos, elegantes y eficientes de algoritmo recursivo, el algoritmo de Euclides opera con una eficiencia logarítmica, es decir, su complejidad es O(log n), donde n es el menor de los dos números. Esta característica lo convierte en un ejemplo excepcional de recursión eficiente, ideal para mostrar que la recursividad no está reñida con la eficiencia algorítmica.

2. **Merge Sort**: este algoritmo divide y conquista es un ejemplo clásico de algoritmo eficiente, con complejidad $O(n*\log_2 n)$. Su análisis permite introducir el cálculo de complejidad en algoritmos recursivos eficientes, típicamente asociados a problemas de la clase P. Es una excelente puerta de entrada para que los estudiantes comprendan cómo la recursividad puede aprovecharse para lograr eficiencia.

3. **Torres de Hanoi**: este problema permite ilustrar el crecimiento exponencial del tiempo de ejecución, específicamente $O(2^n)$. Su inclusión es relevante para mostrar cómo una estructura recursiva tan simple (el algoritmo tiene 6 líneas de código) puede derivar en problemas intratables para entradas grandes, relacionándose con el estudio de problemas NP o incluso de tipo NP-hard, desde una perspectiva pedagógica.

4. **Determinante por cofactores**: el algoritmo clásico para calcular el determinante mediante cofactores tiene una complejidad factorial O(n!), lo que lo convierte en un ejemplo ideal de un algoritmo ineficiente por naturaleza, aunque correcto. Este caso permite discutir con claridad la diferencia entre algoritmos correctos y algoritmos prácticos, además de reforzar la necesidad de soluciones optimizadas en la vida real, como los métodos de eliminación gaussiana.

5. **Función de Ackermann:** finalmente, la función de Ackermann es un ejemplo de función no primitiva recursiva, cuya tasa de crecimiento es hiperexponencial. Se incluye porque permite (al igual que las Torres de Hanoi, con solo unas cuantas líneas de código) demostrar los límites de los algoritmos recursivos y conectar el análisis algorítmico con los fundamentos de la teoría de la computabilidad. Es un recurso clave para que los estudiantes visualicen funciones que, aunque computables, crecen más rápido que cualquier función recursiva primitiva, evidenciando los límites teóricos del cómputo.

En conjunto, estos cinco algoritmos permiten cubrir un espectro completo de complejidades — desde logarítmica hasta más allá del exponencial — y constituyen una base sólida y representativa para introducir el análisis de algoritmos en ambientes educativos. Su selección busca promover el pensamiento crítico y el entendimiento profundo de la eficiencia computacional, en lugar de una mera repetición mecánica de definiciones o fórmulas.

8.1 Euclides recursivo para el cálculo de MCD.

El Máximo Común Divisor de dos números naturales a y b (con $a \geq b$) es el número entero más grande que divide a ambos sin dejar residuo. Por ejemplo, el MCD de 48 y 18 es 6, porque 6 es divisor máximo de ambos números.

El algoritmo de Euclides en pseudocódigo es:

```
int MCD(int a, int b){
    si b=0{
        retorna a
    }
    sino retorna MCD (b, a modulo b) // el operado modulo retorna el residuo de a/b
}
```

La complejidad del algoritmo presenta una relación directa con la sucesión de Fibonacci, la cual se define de la siguiente manera:

$$f_0 = 0, f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}, \text{para } n > 1,\ n \in \mathbb{N}$$

Por lo tanto, la sucesión se genera como:

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

Esta relación es particularmente relevante al analizar el número de llamadas recursivas del algoritmo de Euclides para el cálculo del máximo común divisor (MCD), ya que el peor caso ocurre cuando los argumentos del algoritmo son números consecutivos de Fibonacci.

Para ilustrarlo, consideremos el caso:

$$\text{MCD}(55,34) = \text{MCD}(f_{10}, f_9)$$

A partir de esta elección, es posible observar y cuantificar el número de llamadas recursivas que realiza el algoritmo, evidenciando así cómo la sucesión de Fibonacci establece una cota superior para su complejidad en el peor de los casos.

Esta pareja corresponde a dos números consecutivos de la sucesión de Fibonacci.

Las llamadas son:

MCD(55, 34) → MCD(34, 21) → MCD(21, 13) → MCD(13, 8) → MCD(8, 5) →

MCD(5, 3) → MCD(3,2) → MCD(2, 1) → MCD(1, 0)

En total: 9 llamadas recursivas

Por lo que el tiempo de ejecución del algoritmo se estima mediante el teorema de Lamé.

Teorema de Lamé y la sucesión de Fibonacci

La demostración del Teorema de Lamé se encuentra estrechamente vinculada con la estructura de la sucesión de Fibonacci. En su formulación clásica, Lamé estableció que el peor caso del algoritmo de Euclides para el cálculo del máximo común divisor (MCD) se presenta cuando los enteros de entrada son números consecutivos de dicha sucesión.

En particular, al calcular MCD(F_{k+1}, F_k), donde $F_k$denota el k-ésimo número de Fibonacci, el algoritmo ejecuta exactamente $k$iteraciones (o llamadas recursivas). Este resultado no es accidental, sino que deriva de la propiedad recursiva de la sucesión, la cual reproduce la dinámica misma del algoritmo de Euclides.

Dado que los números de Fibonacci crecen de manera exponencial —aproximadamente como $F_k \sim \frac{\varphi^k}{\sqrt{5}}$, donde φes la razón áurea—, se sigue que el número de iteraciones necesarias para completar el algoritmo crece de forma logarítmica en función del tamaño de los datos de entrada. En consecuencia, el algoritmo de Euclides presenta una complejidad temporal de orden $O(\log n)$, donde $n$corresponde al menor de los dos enteros involucrados.

Más formalmente, el número de pasos está acotado por la posición que ocupa *b* en la sucesión de Fibonacci.

Explicación paso a paso

1. Relación con la sucesión de Fibonacci: Lamé observó que el peor caso del algoritmo de Euclides ocurre cuando los números de entrada son dos números consecutivos de Fibonacci: MCD(f_{n+1}, f_n). En ese caso, el algoritmo hace exactamente n llamadas recursivas.

¿Por qué esto es importante?, como la sucesión de Fibonacci crece exponencialmente:

$$f_n = \frac{\emptyset^n}{\sqrt{5}} \quad \text{Donde } \emptyset = \frac{1+\sqrt{5}}{2}$$

Siendo $\emptyset = 1.6180339887$ ..., la proporción áurea o número de Dios. (La proporción áurea se puede calcular como $\phi = {f_{n+1}}/{f_n}\,, cuando\ n \longrightarrow \infty$).

Si b=f_n, despejando a n en ϕ^nse tiene:

$$f_n = \frac{\emptyset^n}{\sqrt{5}}$$

n$\approx log_{\emptyset}(b\sqrt{5} = log_{\emptyset}(b) + log_{\emptyset}\sqrt{5} \Rightarrow \boldsymbol{\vartheta(log_{\phi}\, b)}$

Sabiendo que:

$$log_a(x) = \frac{log_b(x)}{log_b(a)}$$

Según el ejemplo anterior, para f_n=34, se tiene que el número de iteraciones es:

$$n \approx log_{\phi}(34) + log_{\phi}(\sqrt{5})$$

$log_{\phi}(34) = \frac{log_{10}(34)}{log_{10}(\phi)}$ = 7.32

$$n \approx 7.32 + \frac{log_{10}(\sqrt{5})}{log_{10}(\phi)} \approx 9$$

Es decir, el número de llamadas recursivas es como máximo logarítmico respecto al menor de los dos números.

8.2 Merge sort.

El algoritmo Merge sort pertenece a la estrategia Divide y vencerás. El concepto de Divide y vencerás se define como: Dada una función para computar n entradas, la estrategia divide y conquistaras sugiere dividir la entrada en k subconjuntos, 1< k ≤ n produciendo k subproblemas. Estos subproblemas deben ser resueltos y entonces un método debe ser encontrado para combinar las sub-soluciones dentro de una solución como un todo. Si el subproblema es aún demasiado grande, entonces puede ser reaplicada la estrategia.

Frecuentemente los subproblemas resultantes de un diseño divide y conquistaras son del mismo tipo del problema original. El principio divide y conquistaras se expresa en forma natural por un procedimiento recursivo. Por lo que se obtienen subproblemas de menor dimensión de la misma clase, eventualmente se producen subproblemas que son lo suficientemente pequeños que son resueltos sin la necesidad de dividirlos.

Una forma general de ver el modelo es:

```
void DANDC (int a[], p, q){
// a [ ] es el arreglo a trabajar ; p y q son los subespacios a dividir
int m, p, q;
        si pequeño(p, q)
                retorna(G((p,q))
        sino{
                m=divide(p, q);
                retorna (combinación (DANDC (a, p, m), DANDC (a, m+1, q))
        }
}
```

(Ellis Horowitz, 1978)

Merge Sort es un algoritmo que ordena elementos y tiene la propiedad de que el peor caso en complejidad será: O (n log n). Los elementos van a ser ordenados en forma creciente. Dados n elementos, éstos se dividirán en 2 subconjuntos. Cada subconjunto será ordenado y el resultado será unido para producir una secuencia de elementos ordenados. Su pseudocódigo es el siguiente:

```
void MERGE_SORT (a, low, high){
//a es un arreglo tipo entero donde se guardan números en desorden
//low es el inicio del sub-arreglo a ordenar;  high es el fin del sub-arreglo a ordenar.
 Si low < high entonces{
        mid ← (low + high) / 2
         MERGE_SORT (a, low, mid)
         MERGE_SORT (a, mid + 1, high)
         MERGE (a, low, mid, high)
        }
}
void MERGE(a, low, mid, high){
 Crear arreglo temporal b del mismo tamaño que a
 h ← low
 i ← low
```

```
j ← mid + 1
Mientras ( h ≤ mid y j ≤ high hacer ){
        Si a[h] ≤ a[j] entonces{
        b[i] ← a[h]
        h ← h + 1
        }
        Sino{
              b[i] ← a[j]
              j ← j + 1
        }

        i ← i + 1
}
Si h > mid entonces
        Para k desde j hasta high hacer{
                b[i] ← a[k]
                i ← i + 1

        }

        Sino
        Para k desde h hasta mid hacer{
        b[i] ← a[k]
        i ← i + 1
        }
Para k desde low hasta high hacer
        a[k] ← b[k]
}
```

Considere el arreglo de diez elementos A (310, 285, 179, 652, 351, 423, 861, 254, 450, 520). MERGESORT inicia por dividir A en dos subarreglos de tamaño cinco. Los elementos A(1:5) son a su vez divididos en arreglos de tamaño tres y dos. Entonces los elementos A(1:3) son divididos en dos sub-arreglos de tamaño dos y uno. Los dos valores en A(1:2) son divididos en un sub-arreglo de un solo elemento y la fusión inicia. Hasta este momento ningún movimiento ha sido realizado. Pictóricamente el arreglo puede ser visto de la siguiente forma:

(310|285|179|652, 351|423, 861, 254, 450, 520)

Las barras verticales indican el acotamiento de los subarreglos. A (1) y A (2) son fusionados produciendo

(285, 310| 179|652, 351| 423, 861, 254, 450, 520)

Entonces A(3) es fusionado con A(1:2) produciendo

(179, 285, 310|652, 351|423, 861,254, 450, 520)

Los elementos A(4) y A(5) son fusionados:

(179, 285, 310|351, 652| 423, 861, 254, 450, 520)

Siguiendo la fusión de A(1:3) y A(4:5) se tiene

(179, 285, 310, 351, 652| 423, 861, 254, 450, 520)

En este punto el algoritmo ha retornado a la primera invocación de MERGESORT y se realizará la segunda llamada recursiva. Las llamadas recursivas a la derecha producen los siguientes sub-arreglos:

(179, 285, 310, 351, 652|423|861|254|450, 520)

A(6) y A(7) son fusionados y entonces A(8) se fusiona con A(6:7) dando:

(179, 285, 310, 351, 652,|254, 423, 861|450, 520)

El siguiente paso es A(9) y A(10) son fusionados siguiendo A(6:8) y A(9:10)

(179, 285, 310, 351, 652|254, 423, 450, 520, 861)

En este momento se tienen dos subarreglos ordenados y la fusión final produce la ordenación completa

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

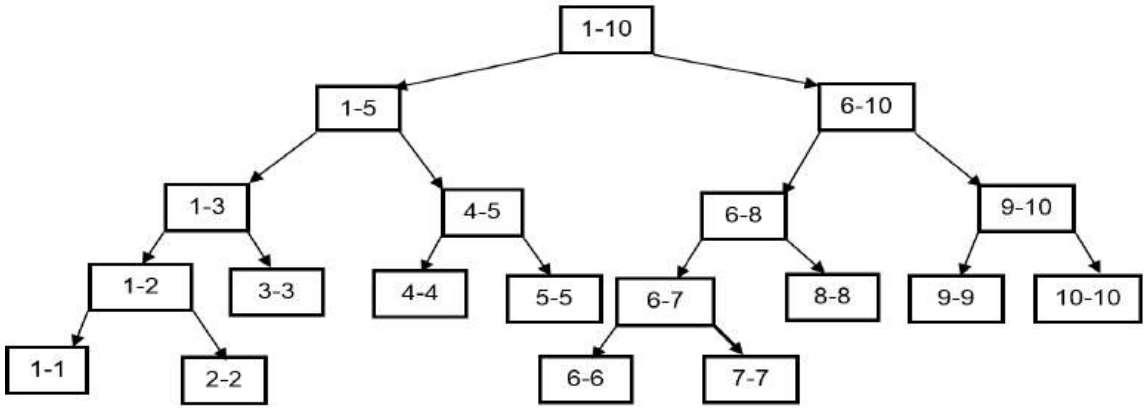

Figura 8.1 Árbol recursivo para el algoritmo MergeSort. Los números dentro de cada rectángulo representan el intervalo de cada subarreglo.

La figura 8.1 muestra el árbol de recursividad producido por MERGESORT. En cada rectángulo se muestran los índices del arreglo low y high en el cual muestra la forma en que se van a dividir los sub-arreglos. Note que la división continúa hasta contener un simple elemento.

La figura 8.2 muestra la parte de división de un arreglo, la parte de ordenación de cada uno de los sub-arreglos y sus respectivas fusiones hasta llegar a obtenerse el arreglo ordenado.

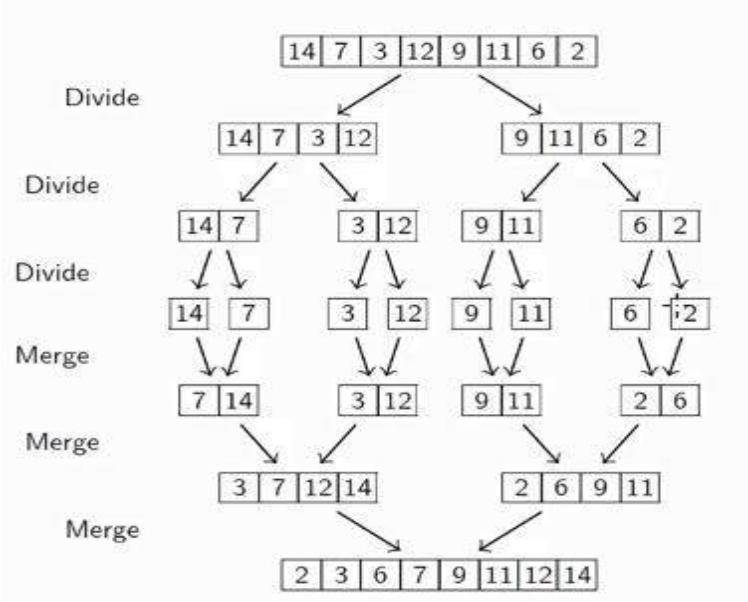

Figura 8.2 Método Merge Sort (University of Pennsylvania. Departament of Engineering)

La relación de recurrencia corresponde al análisis de la complejidad temporal del algoritmo Merge Sort. La función de tiempo T(n) está definida como:

$$T(n) = \begin{cases} a, & n = 1 \; a \; es \; una \; constante \\ 2T\left(\frac{n}{2}\right) + Cn, & n > 1 \; c \; es \; una \; constante \end{cases}$$

Interpretación paso a paso

Caso base T (1) = a:

Si el tamaño del problema es 1, se realiza una operación constante. Esto corresponde al momento en que la lista ya no puede dividirse más.

Caso general T(n)=2T(n/2)+Cn:

- El problema se divide en 2 subproblemas de tamaño n/2 al realizarse dos llamadas recursivas.
- La parte Cn representa el costo de combinar (merge) los resultados, que es lineal respecto al tamaño del arreglo.
- La base de la recursión es T(1)=0, es trivial ordenar un arreglo de un elemento.
- Este patrón es característico del paradigma de divide y vencerás.

Para observar la complejidad del algoritmo se puede calcular de dos formas.

1. Por el Teorema maestro que indica: este teorema ofrece una solución directa para determinar la complejidad temporal asintótica (usando la notación Big O, Θ y Ω) de una amplia gama de relaciones de recurrencia, especialmente de aquellos de naturaleza recursiva como los de "divide y vencerás", evitando la necesidad de realizar expansiones recursivas tediosas o pruebas por inducción.

El Teorema Maestro se aplica a relaciones de recurrencia que siguen una forma específica:

T(n) = a T(n/b) + f(n)

Donde:

- T (n): es la complejidad temporal del problema de tamaño n.
- a: es el número de subproblemas en los que se divide el problema original en cada paso recursivo. Debe ser una constante mayor o igual a 1 (a ≥ 1).
- b >1 es el factor de reducción del tamaño en cada subproblema.
- n/b: es el tamaño de cada uno de los subproblemas. Se asume que todos los subproblemas tienen el mismo tamaño. b debe ser una constante estrictamente mayor que 1 (b>1).
- f(n): es el costo del trabajo realizado fuera de las llamadas recursivas. Esto incluye el costo de dividir el problema y el de combinar las soluciones de los subproblemas. Debe ser una función asintóticamente positiva.

8.2.1 Enunciado del Teorema Maestro

Considérese la recurrencia de la forma:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n),$$

donde $a \geq 1$, $b > 1$y $f(n)$es una función asintóticamente positiva.

El Teorema Maestro permite determinar la complejidad de $T(n)$comparando la función $f(n)$con el término:

$$n^{\log_b a},$$

el cual representa el costo del trabajo puramente recursivo.

Se distinguen tres casos fundamentales:

Caso 1 (subcrítico):

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right), \text{ para algún } \varepsilon > 0.$$

Esto significa que $f(n)$crece asintóticamente más lento que $n^{\log_b a}$. En este caso, el costo está dominado por la expansión recursiva:

$$T(n) = \Theta\left(n^{\log_b a}\right).$$

Caso 2 (crítico):

$$f(n) = \Theta\left(n^{\log_b a} \cdot (\log n)^k\right), \text{para algún } k \geq 0.$$

Aquí, $f(n)$tiene el mismo orden de crecimiento que el trabajo recursivo, posiblemente modulado por factores logarítmicos. Entonces:

$$T(n) = \Theta\left(n^{\log_b a} \cdot (\log n)^{k+1}\right).$$

En particular, si $k = 0$:

$$T(n) = \Theta\left(n^{\log_b a} \log n\right).$$

Caso 3 (supercrítico):

$$f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right), \text{para algún } \varepsilon > 0,$$

y además se cumple la **condición de regularidad**:

$$a\, f\left(\frac{n}{b}\right) \leq c\, f(n), \text{para alguna constante } c < 1 \text{ y } n \text{ suficientemente grande.}$$

Esto implica que $f(n)$ crece más rápido que $n^{\log_b a}$, por lo que el costo está dominado por el trabajo no recursivo:

$$T(n) = \Theta(f(n)).$$

8.2.2. **Limitaciones del Teorema Maestro**

- No aplica si f(n) no es polinomial o no se ajusta a los casos.

- No cubre recurrencias donde n/b no es entero (se usa piso o techo).
- No aplica si *a* no es constante o *b* no es mayor que 1.

Para aplicar el Teorema Maestro, se compara la función f(n) con el término $n^{log_b a}$. Este término representa, a grandes rasgos, el costo del trabajo realizado en los niveles de hojas del árbol de recursión. La relación entre estas dos funciones determina cuál de los tres casos del teorema se aplica.

Para aplicar el Teorema Maestro, se calcula el **exponente crítico**:

$$c_{\text{crit}} = \log_b a.$$

A partir de este valor, se compara el crecimiento de $f(n)$con $n^{\log_b a}$, lo que da lugar a tres interpretaciones intuitivas:

Caso 1: Dominado por las hojas

(El costo de los subproblemas es mayor)

Si

$$f(n) = O\left(n^{\log_b a - \varepsilon}\right), \varepsilon > 0,$$

entonces $f(n)$crece polinómicamente más lento que $n^{\log_b a}$. En consecuencia, la complejidad total está dominada por el costo acumulado de las llamadas recursivas (niveles inferiores del árbol):

$$T(n) = \Theta\left(n^{\log_b a}\right).$$

Caso 2: Equilibrado

(El costo se distribuye uniformemente en el árbol de recursión)

Si

$$f(n) = \Theta\left(n^{\log_b a}\right),$$

entonces el trabajo se distribuye de manera aproximadamente uniforme en todos los niveles del árbol de recursión. Por ello, la complejidad total resulta de multiplicar:

(costo por nivel) × (número de niveles).

- **Costo por nivel:** $\Theta(n^{\log_b a})$
- **Número de niveles:** $\log_b n$

Por tanto:

$$T(n) = \Theta(n^{log_b a} \log n)$$

Caso 3: Dominado por la raíz

(El costo de dividir o combinar es mayor)

Si

$$f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right), \varepsilon > 0,$$

y además se satisface la **condición de regularidad**, entonces el costo está dominado por el trabajo no recursivo (niveles superiores del árbol):

$$T(n) = \Theta(f(n)).$$

Aplicación: Merge Sort

Para el algoritmo *Merge Sort*, la recurrencia es:

$$T(n) = 2T\left(\frac{n}{2}\right) + n.$$

Identificamos los parámetros:

- $a = 2$ (dos subproblemas),
- $b = 2$ (cada subproblema es de tamaño $n/2$),
- $f(n) = n$ (costo lineal de mezcla).

Calculamos el exponente crítico:

$$\log_b a = \log_2 2 = 1.$$

Ahora comparamos:

$$f(n) = n = \Theta(n^1) = \Theta(n^{\log_b a}),$$

por lo que se cumple el **Caso 2 (equilibrado)**.

Aplicando el resultado:

$$T(n) = \Theta(n \log n).$$

Interpretación intuitiva

La intuición coincide con el análisis formal: en cada nivel del árbol de recursión de *Merge Sort*, el costo total de combinar las sublistas es siempre $\Theta(n)$. Dado que el número de niveles es $\log n$, el costo total resulta:

$$T(n) = n\log n.$$

8.2.3 Resolución por inducción (sustitución sucesiva)

Una forma clara de analizar la complejidad de algoritmos recursivos —como *Merge Sort*— consiste en resolver su recurrencia mediante sustitución sucesiva, suponiendo que el tamaño de entrada es una potencia de dos. Sea entonces:

$$n = 2^k.$$

Partimos de la recurrencia:

$$T(n) = 2T\left(\frac{n}{2}\right) + Cn,$$

donde $2T(n/2)$representa las dos llamadas recursivas y Cnel costo lineal de combinar (o procesar) los subproblemas.

Expansión de la recurrencia

Aplicando sustituciones sucesivas:

Primer nivel:

$$T(n) = 2T\left(\frac{n}{2}\right) + Cn$$

Segundo nivel:

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + C\frac{n}{2}\right) + Cn$$

Tercer nivel:

$$T(n) = 2\left(2\left(2T\left(\frac{n}{8}\right) + C\frac{n}{4}\right) + C\frac{n}{2}\right) + Cn$$

Simplificando:

$$T(n) = 8T\left(\frac{n}{8}\right) + 4C\frac{n}{4} + 2C\frac{n}{2} + Cn$$

Forma general

Tras k niveles de expansión:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \; C\left(\frac{n}{2^i}\right).$$

Dado que $n = 2^k$, el proceso termina cuando:

$$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n.$$

Evaluación de la suma

Observamos que en cada nivel:

$$2^i \cdot C\left(\frac{n}{2^i}\right) = Cn,$$

por lo que cada nivel del árbol de recursión tiene un costo lineal $\Theta(n)$.

Como hay $k = \log_2 n$niveles, se obtiene:

$$T(n) = 2^k T(1) + kCn.$$

Si $T(1) = \Theta(1)$, entonces:

$$T(n) = \Theta(n) + Cn\log_2 n = \Theta(n\log n).$$

Conclusión

El análisis por sustitución sucesiva muestra que:

- El costo por nivel del árbol de recursión es $\Theta(n)$,
- La profundidad del árbol es $\log_2 n$,

por lo tanto:

$$T(n) = \Theta(n \log n).$$

Esto implica que la altura del árbol binario de recursión es proporcional a $\log_2(n)$, lo que explica el comportamiento eficiente del algoritmo.

8.3 Torres de Hanói.

El juego de las Torres de Hanoi involucra tres postes y n discos de diámetros 1, 2, 3, …, n. El juego inicia con todos los discos apilados sobre uno de los postes en orden creciente de la parte superior hacia abajo. El juego consiste en colocar todos los discos en otro poste obedeciendo las siguientes reglas:

1. Mover sólo un disco a la vez
2. Nunca colocar un disco sobre un disco de menor diámetro.

El juego se basa en una leyenda: la leyenda de las Torres de Hanói fue creada por el matemático francés Édouard Lucas en 1883. Lucas era un especialista en teoría de números y un gran aficionado a los rompecabezas matemáticos. Inventó este juego (originalmente llamado "La Torre de Brahma" o "Torre de Hanói") y lo comercializó.

La leyenda:

Cuenta la leyenda que el gran templo de Benarés, bajo la cúpula que señala el centro del Mundo, reposa una bandeja de cobre en la que están plantadas tres agujas cuyo diámetro es más fino que le aguijón de una abeja. En el momento de la creación, Dios colocó en una de

las agujas 64 discos de oro puro ordenados por tamaño: desde el mayor que rebosa sobre la bandeja, hasta el más pequeño, en lo más alto del montículo. Es la torre de Brahma.

Incansablemente, día tras día, los sacerdotes del templo mueven los discos haciéndoles pasar de una aguja a otra, de acuerdo con las leyes fijas e inmutables de Brahma que dictan que el sacerdote en ejercicio no mueva más de un disco al día, ni lo sitúe encima de un disco de menor tamaño. El día en que los 64 discos hayan sido trasladados desde la aguja en que Dios los puso al crear el mundo a una cualquiera de las otras dos agujas, ese día la Torre, el Templo y, con gran estruendo, el Mundo desaparecerán.

El pseudocódigo:

```
void Hanói (n, origen, destino, auxiliar) {
        si n=1
                imprime “mover disco de” origen “a” destino
        sino{
                Hanói(n-1, origen, auxiliar, destino)
                  imprime “mover disco de” origen “a” destino
                Hanói(n-1, auxiliar, destino, origen)
        }
} fin de proceso
```

(Cairó/Gardati, 2000)

En la figura 8.3 se observa un árbol recursivo de las torres de Hanoi con 3 anillos. Observar que los números en rojo representan cada llamada recursiva que se realiza, primero se hacen los movimientos a la izquierda, y cuando está agotado el movimiento a la izquierda se invoca la función de Hanoi hacia la derecha para luego continuar a la izquierda. Los números en azul muestran el orden de la impresión donde se observa el movimiento a realizarse.

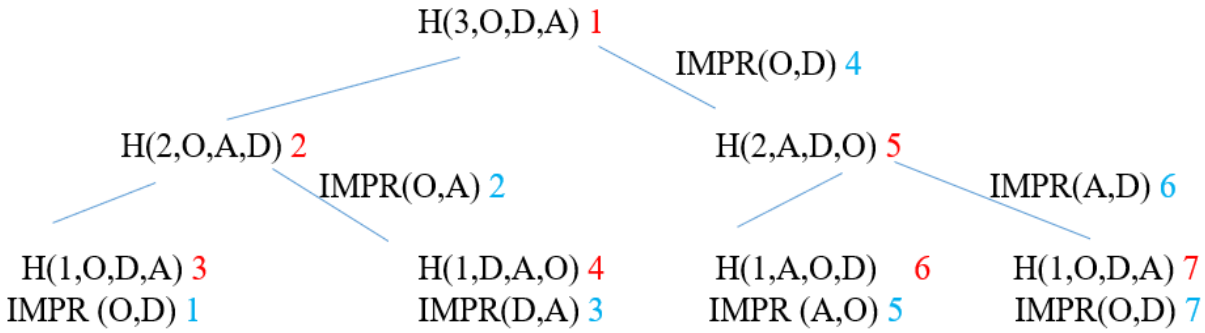

Figura 8.3: Árbol recursivo del algoritmo las torres de Hanoi

Análisis en Profundidad de la Complejidad Algorítmica de las Torres de Hanói

El problema de las Torres de Hanói es un clásico en ciencias de la computación por varias razones:

1. Es un excelente ejemplo de un problema que se resuelve elegantemente con la recursión.

2. Ilustra de manera vívida la naturaleza del crecimiento exponencial en la complejidad algorítmica.

3. Proporciona una clara demostración de cómo se deriva y se resuelve una ecuación de recurrencia.

4. En cada llamada recursiva a la izquierda o derecha del árbol se realiza con *n-1* discos, esto nos indica que la profundidad del árbol=*n*. A comparación del Merge Sort donde la profundidad del árbol es $\log_2(n)$. Esto nos permite observar en forma intuitiva que la complejidad temporal de las Torres de Hanoi es mucho mayor a la complejidad temporal de Merge Sort

La Ecuación de Recurrencia

Sea T(n) el número mínimo de movimientos requeridos para transferir una torre de n discos.

Basándonos en la estrategia recursiva:

- Paso 1: Mover n−1 discos desde origen a auxiliar. Esto requiere T(n−1) movimientos.
- Paso 2: Mover el disco más grande (el n-ésimo) de origen a destino. Esto requiere 1 movimiento.
- Paso 3: Mover n−1 discos desde auxiliar a destino. Esto requiere otros T(n−1) movimientos.

Observar la figura 8.4.

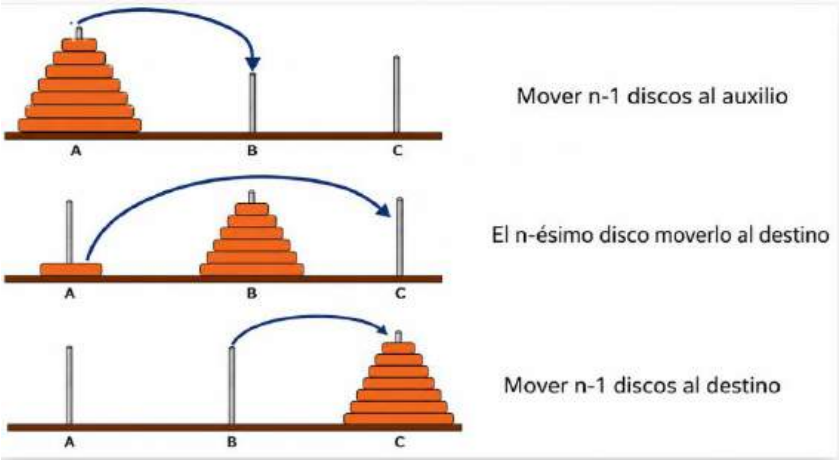

Figura 8.4 Torres de Hanoi.

Sumando los movimientos de cada paso, obtenemos la ecuación de recurrencia:

$$T(n)=T(n-1)+1+T(n-1)$$

$$T(n)=2T(n-1)+1$$

Condición base

Para detener la recursión es necesario establecer una condición base. En el problema de las Torres de Hanói, si se tiene un solo disco, basta un único movimiento para trasladarlo del origen al destino:

$$T(1) = 1.$$

Solución de la ecuación de recurrencia

Sea $n$el número de discos. La recurrencia que describe el número mínimo de movimientos es:

$$T(n) = 2T(n-1) + 1.$$

Desarrollando por sustitución sucesiva:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 = 4T(n-2) + 2 + 1 \\ &= 2^2T(n-2) + (2+1) \\ &= 2^3T(n-3) + (4+2+1) \\ &\vdots \\ &= 2^kT(n-k) + \sum_{i=0}^{k-1} 2^i . \end{aligned}$$

Para finalizar el proceso, tomamos $k = n - 1$, de modo que:

$$T(1) = 1.$$

Sustituyendo:

$$T(n) = 2^{n-1}T(1) + \sum_{i=0}^{n-2} 2^i$$

Sabemos que la suma geométrica cumple:

$$\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1.$$

Por lo tanto:

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

Conclusión

El número mínimo de movimientos necesarios para resolver el problema de las Torres de Hanói es:

$$T(n) = 2^n - 1.$$

En consecuencia, la complejidad algorítmica es:

$$T(n) \in O(2^n),$$

lo que indica un crecimiento exponencial.

Interpretación estructural

El árbol de recursión asociado es binario, ya que cada llamada genera dos subproblemas. Además, su profundidad es igual a n, el número de discos, lo que explica el crecimiento exponencial del número total de llamadas.

Implicaciones de la complejidad exponencial

El crecimiento exponencial implica que el número de operaciones se duplica aproximadamente cada vez que $n$aumenta en una unidad:

- $n = 10 \Rightarrow 2^{10} - 1 \approx 10^3$movimientos
- $n = 20 \Rightarrow 2^{20} - 1 \approx 10^6$movimientos
- $n = 30 \Rightarrow 2^{30} - 1 \approx 10^9$movimientos
- $n = 64 \Rightarrow 2^{64} - 1 \approx 1.84 \times 10^{19}$movimientos

Este crecimiento ilustra por qué el problema se vuelve rápidamente intratable desde el punto de vista computacional conforme aumenta el número de discos.

- Problema Intratable para N Grandes: Un algoritmo con complejidad $O(2^n)$ se considera intratable o infactible para valores de n que no son muy pequeños. Esto significa que no importa cuán rápido sea tu procesador, para un *n* suficientemente grande, el tiempo requerido será prohibitivamente largo.

- Clase de Complejidad: Problemas que requieren un tiempo de ejecución exponencial se encuentran fuera de la clase de problemas "P" (polinomiales), que son considerados "eficientemente solubles". En cambio, pertenecen a clases de complejidad mucho más altas.

En resumen, las Torres de Hanói son una ilustración perfecta de cómo un problema recursivo sencillo puede llevar a un costo computacional masivo debido a su naturaleza exponencial, lo que lo convierte en un problema solo práctico para un número limitado de elementos.

La Realidad Matemática Detrás de la Leyenda

Lo que hace esta leyenda tan intrigante es su conexión directa con un problema matemático y computacional muy conocido. El número de movimientos necesarios para trasladar n discos es 2^n-1.

Para 64 discos, el número de movimientos es: $2^{64}-1=18,446,744,073,709,551,615$ movimientos. Si los sacerdotes pudieran mover un disco por segundo, sin descanso y sin errores, el tiempo que tardarían sería:18,446,744,073,709,551,615 segundos≈584.9 mil millones de años.

Para poner esto en perspectiva, la edad estimada del universo es de aproximadamente 13.8 mil millones de años. Así que, afortunadamente para nosotros, ¡el fin del mundo por esta causa no es una preocupación inminente!

¿Y si lo resolviera una computadora moderna?

Supongamos ahora que usamos un procesador moderno capaz de realizar **mil millones de operaciones por segundo**. Si cada movimiento de disco pudiera representarse como una operación (lo cual es una simplificación extrema), el tiempo requerido sería:

$$\frac{2^{64}-1}{10^9} \approx 18,446,744,073 \text{ segundos} \approx 585 \text{ años}$$

Incluso con esta capacidad computacional, ¡resolver el problema completo tomaría más de medio milenio! Y eso sin considerar que cada movimiento implica lógica, almacenamiento, y verificación, lo que en la práctica requeriría aún más tiempo.

Este ejemplo ilustra cómo **la complejidad exponencial** puede desafiar incluso a la tecnología más avanzada. Las Torres de Hanoi no solo es un juego, sino una poderosa metáfora de los límites del cómputo y del tiempo y cómo incluso tareas aparentemente simples pueden requerir un tiempo astronómico cuando se escalan a un número suficiente de elementos.

8.4 Cálculo del determinante por cofactores.

El cálculo de un determinante mediante recursividad o formalmente conocido como teorema de Laplace. El teorema afirma que el determinante de una matriz es igual a la suma de los productos de cada elemento (de un renglón o columna) por la determinante de su matriz adjunta, lo que reduce un determinante de dimensión *n* a *n* determinantes de dimensión n-1. Aplicado de forma sucesiva, permite llegar a matrices 2x2 (en el que el determinante es el producto de la diagonal principal menos el de la secundaria).

El cálculo del determinante por menores y cofactores de una matriz se puede definir en forma recursiva como se muestra a continuación:

$$Det(A) = \begin{cases} a_{1,1} & si\ j = 1 \\ \sum_{k=1}^{j} (-1)^{1+k} * a_{i,k} * \det(\alpha_{1,k}) & si\ j > 1 \end{cases}$$

(Cálculo de un determinante mediante recursividad en MATLAB, 2025)

Donde $a_{1,k}$ hace referencia al elemento ubicado en la primera fila y k-ésima columna, y $\alpha_{1,k}$ es el adjunto del elemento mencionado anteriormente.

Enseguida se muestra el pseudocódigo:

```
int determinante(a,tam) {
//a. matriz cuadrada ; tam. tamaño de la matriz
si tam=1
        retorna a[0][0];
si tam=2
```

```
        retorna a[0][0]*a[1][1]-a[1][0]*a[0][1];
m=0;
tam1=tam-1;
para i desde 0 hasta tam-1 hacer {//es el número de llamadas recursivas por nível.
        m←signo(i)*determinante(subm(i,a,tam1),tam1)*a[0][i]+m;
retorna m;
 }
```

La función signo retorna $(-1)^{1+k}$ tiene un orden O (1)

La función subm retorna la adjunta, siendo del orden O(n^2)

Para una matriz de 4x4, en la primera etapa recursiva en profundidad, se requieren 4 llamadas recursivas con matrices de 3x3. En la segunda etapa, cada matriz de 3x3 requiere tres llamadas recursivas con matrices de 2x2. Siendo un criterio de paro el cálculo de matrices de 2x2 con un costo de una unidad. Para una matriz de NxN, la primer etapa en profundidad tendrá N llamadas con matrices de tamaño (N-1)x (N-1), En la segunda llamada en profundidad, cada matriz de tamaño (N-1)x(N-1) tendrá N-1 llamadas con matrices de tamaño (N-2)X(N-2), (ver figura 8.5).

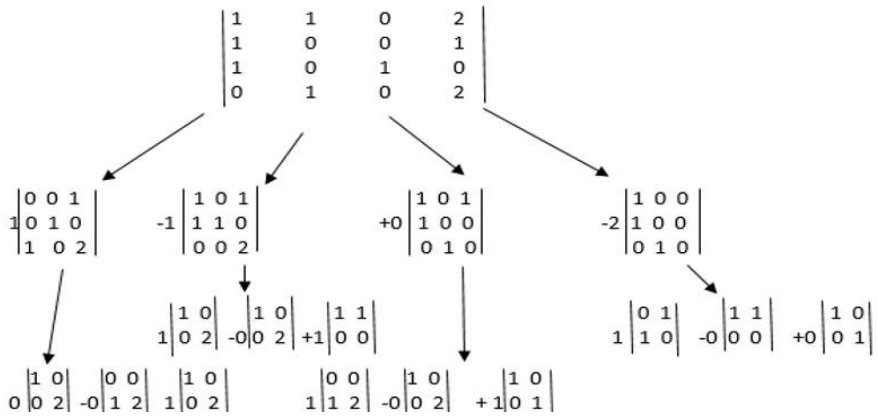

Figura 8.5: Árbol recursivo del cálculo del determinante de una matriz de 4x4

8.4.1. Cálculo de la complejidad.

La forma simplificada para calcular el determinante de una matriz cuadrada A ∈ $\mathbb{R}^{nxn}$ mediante expansión de cofactores posee un comportamiento recursivo en el que, para una matriz de orden *n*, se deben calcular *n* determinantes de orden *n*−1, y así sucesivamente. Si solo se toma en forma directa cada una de las llamadas recursivas, esto conduce a una relación de recurrencia de la forma:

T(n) = n * T(n−1) para el primer nivel de recursividad, donde se hacen n llamadas recursivas para matrices del orden (n-1)x(n-1). Por lo que se puede conducir a la siguiente relación de recursividad:

T(n-1)=(n-1)*T(n-2)

T(n-2)=(n-2)*T(n-3)

Por lo tanto se tiene:

T(n)=n*T(n-1)=n*(n-1)*T(n-2)=n*(n-1)*(n-2)*T(n-3)…

Obteniendo la siguiente relación de recursividad:

T(n) =n*(n-1)*(n-2)*(n-3)*…*T(n-(n-1))=n!

Como condición de paro se tiene: T(1) =1, la cual da como resultado una complejidad de tipo factorial: T(n) ∈ Θ (n!).

Sin embargo, este análisis tradicional considera únicamente el número de llamadas recursivas, omitiendo el costo de construir cada submatriz menor en cada paso. En la práctica, para cada expansión por cofactores, es necesario generar explícitamente *n* submatrices de orden *n* − 1, lo que implica recorrer una cantidad considerable de memoria para eliminar filas y columnas. Si cada operación de generación de un menor se modela con un costo de orden $O(n^2)$, el costo total por nivel recursivo se incrementa, y la recurrencia se refina como: $T(n)=n\cdot T(n-1)+O(n^3)$

¿Por qué O(n^3) y no solo n^2?

Porque:

- Hay n cofactores a calcular,
- Cada uno requiere construir un menor: O(n^2),
- En total: n·O(n^2)=O(n^3)

Entonces, la nueva relación de recurrencia es:

$$T(n) = n \cdot T(n-1) + c\, n^3,$$

la cual modela el costo computacional del cálculo del determinante de una matriz $n \times n$mediante expansión por cofactores.

Expansión de la recurrencia

Mediante sustitución sucesiva, se obtiene:

$$T(n) = n(n-1)\, T(n-2) + n\, c\, (n-1)^3 + c\, n^3,$$
$$T(n) = n(n-1)(n-2)\, T(n-3) + n(n-1)\, c\, (n-2)^3 + n\, c\, (n-1)^3 + c\, n^3,$$
$$T(n) = n(n-1)(n-2)(n-3)\, T(n-4) + n(n-1)(n-2)\, c\, (n-3)^3 + n(n-1)\, c\, (n-2)^3 + n\, c\, (n-1)^3 + c\, n^3.$$

Generalizando, tras k expansiones:

$$T(n) = \frac{n!}{(n-k)!}\, T(n-k) + c \sum_{i=0}^{k-1} \frac{n!}{(n-i)!} (n-i)^3.$$

Tomando $k = n-1$y considerando la condición base $T(1) = 1$, se obtiene:

$$T(n) = n! \cdot T(1) + c\, n! \sum_{k=2}^{n} \frac{k^3}{k!}.$$

Análisis de la sumatoria

Se estudia la serie:

$$\sum_{k=2}^{\infty} \frac{k^3}{k!}.$$

Aplicando el **criterio de la razón de D'Alembert**:

$$\lim_{k\to\infty} \frac{\frac{(k+1)^3}{(k+1)!}}{\frac{k^3}{k!}} = \lim_{k\to\infty} \frac{(k+1)^3}{(k+1)\,k^3} = \lim_{k\to\infty} \frac{(k+1)^2}{k^3} = 0.$$

Dado que el límite es menor que 1, la serie converge. En consecuencia:

$$\sum_{k=2}^{n} \frac{k^3}{k!} \in \Theta(1).$$

Complejidad asintótica

Sustituyendo el resultado anterior:

$$T(n) = n! \cdot T(1) + c\, n! \cdot \Theta(1),$$

por lo tanto:

$$T(n) \in \Theta(n!).$$

Esto demuestra que el algoritmo de expansión por cofactores presenta un crecimiento factorial, significativamente superior al crecimiento exponencial.

Interpretación estructural del árbol de recursión

El árbol de recursión asociado presenta las siguientes características:

- **Profundidad:** n,
- **Factor de ramificación variable:** en el nivel i existen aproximadamente $n - i$ subproblemas,
- **Número total de nodos:** proporcional a $n!$.

A diferencia de algoritmos como *Merge Sort* (altura $\log n$) o las Torres de Hanói (árbol binario de altura n), el árbol en este caso presenta una expansión inicial muy amplia que decrece progresivamente, lo que incrementa drásticamente el costo total.

Implicaciones prácticas

El crecimiento factorial implica que el algoritmo es computacionalmente inviable para matrices de tamaño moderado. Esto se evidencia en el número de llamadas recursivas (Ver Tabla 8.1):

Tabla 8.1 Número de llamadas recursivas por tamaño de matriz

n	Número de llamadas
1	1
2	3
3	10
4	41
5	206

6	1237
7	8660
8	69281

Comparando distintas clases de complejidad (Ver tabla 8.2):

Tabla 8.2 Evaluación de distintos tipos de complejidad

n	$\log n$	$n\log n$	2^n	$n!$
5	2.32	11.61	32	120
8	3.00	24.00	256	40320
10	3.32	33.2	1024	3628800

Se observa que el crecimiento factorial domina ampliamente a las demás clases, confirmando que este método tiene principalmente valor teórico y didáctico.

Conclusión

El análisis formal demuestra que el cálculo del determinante mediante expansión por cofactores tiene complejidad $\Theta(n!)$. Este resultado evidencia la necesidad de emplear métodos alternativos más eficientes, como la eliminación gaussiana, en aplicaciones prácticas con un $O(n^3)$.

8.5. La impresionante función de Ackermann

Hasta ahora hemos explorado diversas funciones recursivas cuya complejidad algorítmica, aunque creciente, se mantiene dentro de límites razonables para su análisis computacional.

Sin embargo, la función de Ackermann representa un punto de inflexión en este recorrido: es uno de los ejemplos más famosos de una función computable que no es recursiva primitiva, y cuya tasa de crecimiento supera con creces a funciones comunes como las exponenciales o factoriales.

La función de Ackermann no solo desafía nuestra intuición sobre el crecimiento de funciones, sino que también sirve como una poderosa herramienta para distinguir entre distintos niveles de complejidad computacional. Su estructura profundamente recursiva genera resultados extremadamente grandes con entradas sorprendentemente pequeñas, lo que la convierte en un caso de estudio esencial para entender los límites de ciertos enfoques computacionales y los alcances de la recursión general.

En Teoría de la Computación, la función de Ackermann es una función matemática recursiva encontrada en 1926 por Wilhelm Ackermann, es de interés para la ciencia computacional teórica. Hoy en día, hay una serie de funciones que son llamadas funciones Ackermann. Todas ellas tienen una forma similar a la función de Ackermann y también tienen un comportamiento de crecimiento similar. Esta función toma dos números naturales como argumentos y devuelve un único número natural. Como norma general se define en la ecuación siguiente:

$$A(M,N)\begin{cases} N+1 & \text{Si } M=0 \\ A(M-1,1) & \text{Si } N=0 \text{ y } M>0 \\ A(M-1, A(M,N-1)) & \text{En cualquier otro caso} \end{cases}$$

Enseguida se muestra el algoritmo en pseudocódigo:

```
int funcion Ackermann (m, n){
si m=0
        retorna n+1
```

```
si n=0
        retorna Ackermann(m-1,1)
retorna Ackermann(m-1, Ackermann(m,n-1))
} fin de procedimiento
```

Cálculo de la complejidad temporal.

Para comprender mejor su complejidad, se estudiará A(n,m) para valores de n=1,2,3,4,5

Sabemos que

A(0,n)=n+1

A(m,0)=A(m-1,1)

A(m, n)=A(m-1, A(m,n-1))

Entendiendo la definición recursiva para A(1,n), se sabe que:

Si n=0, se usa la segunda línea:

A(1,0) = A(0,1) =1+1=2

Calcular algunos valores de A(1,n), expliquemos paso a paso:

1. A(1,1)

Se sabe de antemano que A(1,0) =2, por lo tanto: A(1,1) =A(0, A(1,0)) =A(0,2)=3

2. A(1,2)

Se sabe de antemano que A(1,1) =3, por lo tanto: A(1,2) =A(0,A(1,1)) =A(0,3) =4

3. A(1,3)

Se sabe de antemano que A(1,2) =4, por lo tanto A(1,3) =A(0,A(1,2)) =A(0,4) =5

Observación:

A(1, n)=n+2, n≥0

De esta forma, se puede observar el recorrido en profundidad:

$$A(1,3)=A(0,A(1,2))=1+A(1,2)=1+A(0,A(1,1))=1+1+A(1,1)=1+1+A(0,A(1,0))$$
$$=1+1+1+A(1,0)=1+1+1+A(0,1)=1+1+1+2=5$$

Observar que n=3.

Entendiendo la definición recursiva para A(2,n)

$$A(2,n) \begin{cases} A(1,1) & \text{Si } N=0 \\ A(1, A(2, n-1)) & \text{Si } N>0 \end{cases}$$

Y como ya se sabe: A(1,n)=n+2

Se Calcula paso a paso los primeros valores

1. A(2,0) ⟹ A(2,0) =A(1,1) =1+2=3
2. A(2,1) ⟹ A(2,1) =A(1,A(2,0)) =A(1,3) =3+2=5
3. A(2, 2) ⟹ A(2,2) =A(1,A(2,1)) =A(1,5) =5+2=7
4. A(2,3) ⟹ A(2,3) =A(1,A(2,2)) =A(1,7) =7+2=9

Los valores de A(2, n) se ven en la tabla 8.3:

Tabla 8.3. A(2,n)

n	A (2,n)
0	3
1	5

2	7
3	9
4	11

Se puede ver que: **A(2,n)=2n+3**

Desde un enfoque recursivo se generan estos datos:

A(2,n)=A(1,A(2,n-1))

=2+A(2,n-1)

=2+A(1,A(2,n-2))

=2+2+A(2,n-2)

=2+2+A(1,A(2,n-3))

=2+2+2+A(2,n-3)

=2+2+2+A(1,A(2,n-4))

=2+2+2+2+A(2,n-4)

Se puede observar que hasta este momento hay la suma de cuatro números dos, existiendo una relación con la resta A(2,n-x), donde x=4. Cuando se llega a A(2,n-n) =A(2,0) =3 se tienen en ese momento n veces la suma del 2. Por lo tanto:

A(2, n) =2n+A (2,0) =2n+3

Entendiendo la definición recursiva para A(3,n)

Se parte de estos tres valores:

A(0,n)=n+1

A(1,n)= n+2

A(2,n)=2n+3

Cálculo de A(3,n)

Recordando que: A(m, n) = A(m-1, A(m,n-1))

Ahora calculemos algunos valores:

1. A(3,0) ⟹ A(3,0) =A(2,1) =2(1)+3=5

2. A(3,1) ⟹ A(3,1) =A(2,A(3,0)) =A(2,5) =2(5)+3=13

3. A(3,2) ⟹ A(3,2) =A(2,A(3,1)) =A(2,13) =2(13)+3=29

4. A(3,3) ⟹ A(3,3) =A(2,A(3,2)) =A(2,29)+3=61

4. A(3,4) ⟹ A(3,4) =A(2,A(3,3)) =A(2,61) =2(61)+3=125

Los valores para obtener A(3, n) crecen de la siguiente forma (Tabla 8.4):

Tabla 8.4. A(3,n)

n	A(3, n)
0	5
1	13
2	29
3	61

4	125
5	253
6	509
7	1021

Se intuye un crecimiento exponencial aproximado a 2^n. Para localizar el valor exacto se realiza una incursión recursiva en profundidad para encontrar la expresión adecuada, se tiene:

A(3,n)=A(2, A(3, n-1))

= 3+2A(3,n-1)

= 3+2A(2, A(3, n-2))

=3+2(3+2A(3,n-2))

=3+2(3+2(A(2,A(3,n-3))))

=3+2(3+2(3+2(A(2,A(3,n-4))))) //La profundidad termina cuando n-n=0

=3+2(3+2(3+2(3+... 2A(3,0)))…) //A(3,0)=A(2,1)=2(1)+3=5

=3+2(3+2(3+2(3+....2(5)))))…)

Ejemplo:

Cálculo de $A(3,4)$

Partimos de la definición de la función de Ackermann para $m = 3$:

$$A(3,n) = A(2,A(3,n-1)).$$

Además, se utiliza la identidad conocida:

$$A(2,n) = 2n + 3.$$

Entonces:

$$\begin{aligned} A(3,4) &= A(2, A(3,3)) \\ &= 2\,A(3,3) + 3. \end{aligned}$$

Aplicando recursivamente:

$$\begin{aligned} A(3,4) &= 2(2A(3,2) + 3) + 3 \\ &= 2^2 A(3,2) + 2 \cdot 3 + 3 \\ &= 2^2(2A(3,1) + 3) + 2 \cdot 3 + 3 \\ &= 2^3 A(3,1) + 2^2 \cdot 3 + 2 \cdot 3 + 3 \\ &= 2^3(2A(3,0) + 3) + 2^2 \cdot 3 + 2 \cdot 3 + 3 \\ &= 2^4 A(3,0) + 2^3 \cdot 3 + 2^2 \cdot 3 + 2 \cdot 3 + 3. \end{aligned}$$

Sabemos que:

$$A(3,0) = 5.$$

Sustituyendo:

$$A(3,4) = 2^4 \cdot 5 + 3(2^3 + 2^2 + 2 + 1).$$

Factorizando:

$$\begin{aligned} A(3,4) &= 2^4 \cdot 5 + 3\sum_{i=0}^{3} 2^i \\ &= 2^4 \cdot 5 + 3(2^4 - 1) \\ &= 2^4(5 + 3) - 3 \\ &= 2^4 \cdot 8 - 3 \\ &= 2^7 - 3 \\ &= 128 - 3 = 125. \end{aligned}$$

Generalización

Siguiendo el mismo patrón, se obtiene:

$$A(3,n) = 2^n \cdot 5 + 3\sum_{i=0}^{n-1} 2^i.$$

Utilizando la suma geométrica:

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1,$$

se tiene:

$$\begin{aligned} A(3,n) &= 2^n \cdot 5 + 3(2^n - 1) \\ &= 5 \cdot 2^n + 3 \cdot 2^n - 3 \\ &= 8 \cdot 2^n - 3 \\ &= 2^{n+3} - 3. \end{aligned}$$

Resultado final

$$\boxed{A(3,n) = 2^{n+3} - 3}$$

Interpretación

El término 2^n refleja el crecimiento exponencial de la función, mientras que el exponente $n + 3$ puede interpretarse como la **profundidad efectiva de la recursión** en este nivel de la jerarquía de Ackermann. Este resultado ilustra cómo, incluso para valores pequeños de m, la función crece extremadamente rápido, superando a la mayoría de las funciones elementales.

Entendiendo la definición recursiva para A(4,n)

Se usará la definición de A(3,n) para calcular A(4,n)

A(4,n) =A(3, A(4,n-1))

A(4,n) =A(3, A(4,n-1))

$=2^{A(4,\ n-1)+3}$ -3

$=2^{2^{A(4,A(n-2))+3}-3}$-3

Obteniéndose como resultado n potencias sucesivas de 2.

Como ejemplo se calcula A(4,2)

A(4,2) $=2^{A(4,1)+3}-3$

$=2^{A(3,A(4,0)+3}-3$

$=2^{2^{A(4,0)+3}-3}-3$

$=2^{2^{2^4}-3}-3=2^{2^{16}-3}-3=2^{65533}-3 \equiv 65566\log_{10}(2)\ digitos \approx 19728\ digitos.$

Otra forma de explicar A(4,n) en adelante en forma aproximada es por medio de las flechas Knuth.

8.5.1. Flechas de Knuth

Las flechas Knuth es un método de notación para potencias muy grandes donde existen hiperoperaciones.

1. Exponenciación: a multiplicado por sí mismo b veces

$$a \uparrow^1 b = a \uparrow b = a^b = \underbrace{a * a * a \ldots . * a}_{\text{b veces}}$$

2. Tetración: a exponenciado por si mismo b veces

$$a \uparrow^2 b = a \uparrow\uparrow b = \underbrace{a^{(a^{(a^{(a\ldots)})})}}_{\text{b veces}} = \underbrace{a \uparrow (a \uparrow (a \uparrow (\ldots \uparrow a) \ldots)}_{\text{b veces}}$$

3. Pentación: Tetración de a, b veces

$$a \uparrow^3 b = a \uparrow\uparrow\uparrow b = \underbrace{a \uparrow\uparrow a \uparrow\uparrow a \uparrow\uparrow \cdots a}_{\text{b veces}}$$

4. Generalizando

$$a \uparrow^n b = a \uparrow\uparrow .. \uparrow b = a \uparrow^{n-1} (a \uparrow^{n-1} a \uparrow^{n-1} \ldots a) \ldots))$$

n veces

b veces

En forma recursiva se tiene:

$$a \uparrow^n b = \begin{cases} a^b & si\ n = 1 \\ a \uparrow^{n-1} (a \uparrow^n (b-1)) & si\ b > 0, n \geq 2 \\ 1 & si\ b = 0, n \geq 1 \end{cases}$$

Entonces:

$2\uparrow\uparrow 2=2^2=4$

$2\uparrow\uparrow 2$ //Una torre de 2 dos.

$2\uparrow\uparrow 4 = 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65536$

Observando el punto 4 de la definición de las flechas de Knuth:

$2\uparrow\uparrow\uparrow 2 = 2 \uparrow\uparrow 2=4$

Pero:

$2\uparrow\uparrow\uparrow 3 = 2 \uparrow\uparrow (2 \uparrow\uparrow 2)$

- $2\uparrow\uparrow 2=2^2=4$
- $2\uparrow\uparrow 4=2^{2^{2^2}} = 2^{2^4} = 2^{16}=65{,}536$

$2\uparrow\uparrow\uparrow 4 = 2 \uparrow\uparrow \big(2 \uparrow\uparrow (2 \uparrow\uparrow 2)\big)$

- $2\uparrow\uparrow 2=2^2=4$

- $2\uparrow\uparrow 4=65{,}536$
- $2 \uparrow\uparrow 65536 =$ Una torre de potencias de 2 de altura 65,536

En forma recursiva se evalúa:

$2\uparrow\uparrow\uparrow 4=2\uparrow\uparrow(2\uparrow\uparrow\uparrow 3) = 2\uparrow\uparrow(2\uparrow\uparrow (2 \uparrow\uparrow\uparrow 2))= 2\uparrow\uparrow(2\uparrow\uparrow (2 \uparrow\uparrow (2\uparrow\uparrow\uparrow 1)))$

$2\uparrow\uparrow(2\uparrow\uparrow (2 \uparrow\uparrow (2\uparrow\uparrow\uparrow 1))) = 2\uparrow\uparrow(2\uparrow\uparrow (2 \uparrow\uparrow (2\uparrow\uparrow (2 \uparrow\uparrow\uparrow 0)))$

$2\uparrow\uparrow(2\uparrow\uparrow (2 \uparrow\uparrow (2\uparrow\uparrow (2 \uparrow\uparrow\uparrow 0))) = 2\uparrow\uparrow(2\uparrow\uparrow (2 \uparrow\uparrow 2))$

$2\uparrow\uparrow(2\uparrow\uparrow (2 \uparrow\uparrow 2)) = 2\uparrow\uparrow 65{,}536$

Para dar una idea de lo que representa:

$2 \uparrow\uparrow 5 = 2 \uparrow (2 \uparrow\uparrow 4) = 2^{65{,}536} = 2.00359 * 10^{19728}$

$2 \uparrow\uparrow 6 = 2\uparrow (2 \uparrow\uparrow 5) = 2^{2.00359*10^{19728}}$.

$2 \uparrow\uparrow 65536 =$ Es un número imposible de calcular directamente. Es mucho más allá de lo que cualquier calculadora puede representar.

8.5.2. Relación entre la Notación de Knuth y la Función de Ackermann

La Función de Ackermann y las hiperoperaciones de Knuth están estrechamente relacionadas en cuanto a su capacidad de generar un crecimiento extremadamente rápido. Para valores pequeños de m, la función de Ackermann se relaciona con las hiperoperaciones de la siguiente manera:

- A(1, n) es equivalente a una forma de suma.
- A(2, n) es equivalente a una forma de multiplicación.
- A(3, n) crece de forma exponencial (relacionado con $a\uparrow b$).
- A(4, n) crece de forma tetracional (relacionado con $a\uparrow\uparrow b$).

Entendiendo la definición recursiva para A(5, x), utilizando la notación Knuth se tiene:

- A(3, n) = 2↑(n+3)-3
- A(4, n) = $2\uparrow\uparrow (n+3)-3$
- A(5, n) = $2\uparrow\uparrow\uparrow (n+3)-3$

Se van a calcular algunos ejemplos de A (5, n):

1. Calcular A(5,0) =$2\uparrow\uparrow\uparrow 3-3$

$2\uparrow\uparrow\uparrow 3 = 2\uparrow\uparrow(2\uparrow\uparrow 2) = 2\uparrow\uparrow 4 = 2^{2^{2^2}} = 2^{2^4} = 2^{16}$=65536

Por lo tanto, A (5,0) = 65536-3

2. A (5,1) = $2\uparrow\uparrow\uparrow 4-3$

$2\uparrow\uparrow\uparrow 4 = 2\uparrow\uparrow(2\uparrow\uparrow\uparrow 3)$

Ya se vio que $2\uparrow\uparrow\uparrow 3 = 65536$

Entonces: $2\uparrow\uparrow\uparrow 4 = 2\uparrow\uparrow 65536$

Esto es: $2^{2^{2^{\cdots}}}$ con 65536 capas de potencia.

A(5,1) =$2\uparrow\uparrow 65536-3$

Resumiendo, los cálculos obtenidos para A(5,n) crecen exponencialmente y se vuelven inalcanzables (Tabla 8.5):

Tabla 8.5. A(5,n)

n	A (5, n)
0	655336553365533
1	2↑↑ 655336..3

2	2↑↑↑ 65539..3
3	No cabe en el universo
4	Fuera del entendimiento humano

Si se continua, por definición recursiva para A(6,n) se tiene:

A(6,n)=$2\uparrow^{4}(n+3)-3$,

Y para A(7,n) la definición recursiva queda así:

A(7,n) =A (6, A (7, A(7, n-1)) con A(7,0) =A(6,1)

Y como: A(6, n) = $2\uparrow\uparrow\uparrow\uparrow(n+3)-3$

Entonces: A(7,0) = A(6,1) =$2\uparrow\uparrow\uparrow\uparrow 4-3$

A(7,1) =A(6, A(7,0)) =A(6, $2\uparrow\uparrow\uparrow\uparrow 4-3)=2\uparrow\uparrow\uparrow\uparrow(2\uparrow\uparrow\uparrow\uparrow 4)-3$

Por lo tanto: A(7,n)=$2\uparrow^{5}(n+3)-3$.

Esta correspondencia destaca porqué la función de Ackermann es tan importante en la teoría de la computabilidad: demuestra que existen funciones que crecen más rápido que cualquier función que pueda definirse usando solo las operaciones recursivas primitivas (suma, multiplicación, exponenciación limitada), lo que subraya los límites del crecimiento de ciertas clases de funciones. Enseguida se muestra en la tabla comparativa 8.6 en la que se muestra cómo se van incrementando los resultados de la función.

Tabla 8.6 Tabla comparativa

x	A(0,x)	A(1,X)	A(2,X)	A(3,X)	A(4,X)	A(5,X)
0	1	2	3	5	13	655336

1	2	3	5	13	65533	A (4,65533)
2	3	4	7	29	$2^{65536} - 3$ (número con 19729 dígitos)	A(4,A(5,1))
3	4	5	9	61	A(4, A(4,2)) (imposible de calcular)	A(4,A(5,2))

En términos de crecimiento, los hiperoperadores de Knuth representan una jerarquía que queda por debajo de la función de Ackermann, por lo que se puede llegar a la siguiente conclusión:

$$\boldsymbol{Ackermann(n+2,b) \in \Theta(2 \uparrow^{n} (b+3)) \quad (para\ valores\ grandes)}$$

Siempre que n≥1.

La tabla 8.7 muestra la relación entre la complejidad temporal aproximada con los resultados junto con su complejidad temporal aproximada expresada en notación clásica y notación de flechas de Knuth.

m	A(m, n)	Complejidad Temporal Aproximada
0	n + 1	O (n)
1	n + 2	O (n)
2	2n + 3	O (n)
3	2^{n+3} - 3	O (2^n)

4	2 ↑↑ (n+3)-3	O (2 ↑↑ n)
5	2 ↑↑↑ (n+3)-3	O (2 ↑↑↑ n)
6	2 ↑↑↑↑ (n+3)-3	O (2 ↑↑↑↑ n)

Tabla 8.7: Complejidad temporal aproximada de la función de Ackermann

8.3 Ackermann en compiladores, interpretes, sistemas operativos y arquitecturas

La complejidad espacial de una función recursiva viene dada por la profundidad máxima de la pila de llamadas (*call stack*) durante su ejecución. En el caso de la función de Ackermann, esta profundidad puede estimarse de la siguiente manera:

- **Para A(3,n):** La profundidad es O(n), ya que el número de llamadas anidadas crece linealmente con el parámetro n.
- **Para A(4,n):** La profundidad de la pila crece en el orden de O($2\uparrow\uparrow n$), es decir, de forma tetracional, alcanzando valores extremadamente grandes incluso para $n \geq 3$.

Por tanto, la memoria utilizada puede modelarse como:

$$S(n) \approx profundidad(n)\ x\ tamaño\ del\ frame$$

El tamaño de cada *frame* de activación depende directamente del compilador, del sistema operativo y de las variables locales almacenadas en cada llamada.

Implementación en un compilador C

En arquitecturas de 64 bits, un compilador típico como GCC o Clang genera código optimizado donde cada *stack frame* incluye, como mínimo, la dirección de retorno, las

variables locales y los registros temporales o preservados (según la Interfaz Binaria de Aplicaciones o ABI).

El tamaño de un *frame* de activación suele oscilar entre 32 y 128 bytes, dependiendo del código generado. Por ejemplo, si la evaluación de A(4,1) produce aproximadamente 16,000 niveles de recursión, el consumo de memoria de la pila puede estimarse como:

$$S \approx 16{,}000 \; x \; 64 \; bytes \; \approx 1 \; MB$$

Esta cifra ilustra cómo la función de Ackermann puede sobrepasar rápidamente los límites físicos de pila de un entorno de ejecución convencional.

Implementación en un intérprete (Python)

En Python, el escenario es aún más restrictivo. Cada llamada recursiva genera no solo un marco en la pila nativa (C stack), sino también un objeto *frame* del intérprete que incluye metadatos (nombre de función, línea actual), un diccionario de variables locales y referencias a objetos dinámicos.

Debido a esta naturaleza dinámica, donde cada *frame* es un objeto completo con estructuras internas, el consumo de memoria no es de decenas de bytes como en C, sino del orden de kilobytes por llamada. Por ello, Python impone un límite de recursión estricto (1000 llamadas por defecto). Esta restricción no obedece a la teoría matemática de la función, sino a un mecanismo de seguridad del intérprete para evitar un desbordamiento catastrófico de la pila nativa del sistema. Así, incluso cálculos matemáticamente tratables como A(3,100) pueden saturar al intérprete, evidenciando la ineficiencia espacial inherente al modelo interpretado.

Papel del sistema operativo

Cada hilo de ejecución dispone de un tamaño de pila máximo impuesto por el sistema operativo. Por ejemplo, Windows asigna alrededor de 1 MB por defecto para procesos compilados con Visual Studio, mientras que Linux y macOS asignan típicamente 8 MB en entornos de terminal.

Aunque estos valores pueden modificarse mediante parámetros del enlazador o políticas del entorno de ejecución, el límite físico determina de manera inexorable si una invocación de Ackermann se completará o provocará un error fatal (*Stack Overflow*). Esto demuestra que la validez matemática de un algoritmo no garantiza su viabilidad de ejecución.

La Función de Ackermann como espejo de las Arquitecturas Computacionales

La función de Ackermann, A(m,n), actúa como un "cristal intensificador" que expone las limitaciones fundamentales de cualquier arquitectura computacional. Su crecimiento superexponencial, característico de las funciones recursivas generales no primitivas, provoca que los recursos físicos se tornen insuficientes a distintas velocidades según la plataforma empleada.

- **Escenario 1: El muro de la pila única (PC)**

  En una arquitectura clásica tipo PC (modelo de Von Neumann), el problema se asemeja a "un corredor de alta velocidad en un callejón estrecho". Estas máquinas utilizan una única pila de ejecución por hilo, diseñada para minimizar la latencia en tareas secuenciales, no para soportar profundidades extremas. Para A(4,1), se requieren cerca de 65,000 marcos, rozando el límite de memoria. Para A(4,2), la profundidad explota, causando un desbordamiento inmediato. Esto revela que la arquitectura de propósito general es ineficiente para asignar todos sus recursos a un único problema de profundidad extrema.

- **Escenario 2: La paradoja del Mainframe**

  El *mainframe*, diseñado como una "fortaleza" de alta disponibilidad y memoria masiva, prioriza la concurrencia y la seguridad. Aunque puede ejecutar miles de instancias de Ackermann en paralelo (aisladas entre sí), cada hilo individual sigue sometido a límites de pila similares a los de un PC. La arquitectura no permite "sumar" la memoria de otros procesos para resolver una recursión profunda individual. Funciona como una flota de botes salvavidas seguros: excelentes para rescatar a muchas personas simultáneamente, pero incapaces de realizar una inmersión profunda unitaria.

- **Escenario 3: La barrera del paralelismo (Supercomputadoras)**

En el cómputo de alto rendimiento (HPC), basado en paralelismo masivo y memoria distribuida, la implementación de Ackermann requiere replantear la ejecución mediante memorización distribuida y cálculo cooperativo. Sin embargo, la estructura intrínseca de la función impone un obstáculo significativo: cada paso depende estrictamente del resultado anterior. Incluso con miles de nodos, el costo de comunicación y sincronización para mantener la coherencia de los resultados parciales en niveles profundos anula cualquier beneficio obtenido por el paralelismo.

- **Escenario 4: Reescribiendo Ackermann en el Espacio de Hilbert**

  El traslado de la función de Ackermann al contexto de la computación cuántica plantea un problema teórico particularmente complejo. En una computadora clásica, el cálculo sigue una trayectoria secuencial en la que el tiempo de ejecución crece de acuerdo con la expansión del árbol recursivo (véase Figura 8.6, izquierda). En un modelo cuántico, el enfoque conceptual cambia significativamente: un estado inicial $|m,n\rangle$ podría evolucionar mediante transformaciones unitarias hacia una superposición de configuraciones asociadas a distintas ramas del proceso recursivo (véase Figura 8.6, derecha). Desde una perspectiva teórica, esto sugiere la posibilidad de explorar múltiples trayectorias computacionales de manera simultánea. Sin embargo, la extracción del resultado correcto requeriría mecanismos de interferencia cuántica capaces de amplificar las configuraciones válidas y suprimir aquellas que no contribuyen al resultado buscado.

Implicaciones y límites estructurales en cómputo cuántico

Esta implementación plantea un escenario fascinante: reducir la exploración secuencial mediante la superposición cuántica. Sin embargo, surgen dos barreras fundamentales:

1. **Costo de Reversibilidad:** Las operaciones cuánticas deben ser estrictamente reversibles (unitarias). Adaptar una función clásica destructiva como Ackermann requiere técnicas de computación reversible (como el uso de bits ancilla) que incrementan el consumo de recursos de manera inmensa.
2. **Límite Estructural (La limitación estructural):** Más allá de la reversibilidad, el obstáculo real es la profunda dependencia de datos. Cada llamada de Ackermann

depende del resultado concreto de una llamada anterior (evaluar A(m,n) requiere resolver primero A(m, n-1) antes de proceder). Esto impide aprovechar la ventaja cuántica, la cual florece en problemas con simetrías o estructuras algebraicas ocultas (como en la factorización de enteros), no en funciones hiper-recursivas y estrictamente secuenciales.

Por ello, la computación cuántica no representa una solución universal para este problema. La dificultad no es únicamente cuantitativa, sino estructural: el crecimiento recursivo del algoritmo introduce restricciones que no desaparecen simplemente mediante paralelismo cuántico.

Nota: La computación cuántica es un campo en desarrollo y la implementación directa de funciones recursivas complejas como Ackermann representa un desafío computacional teórico más que una aplicación práctica

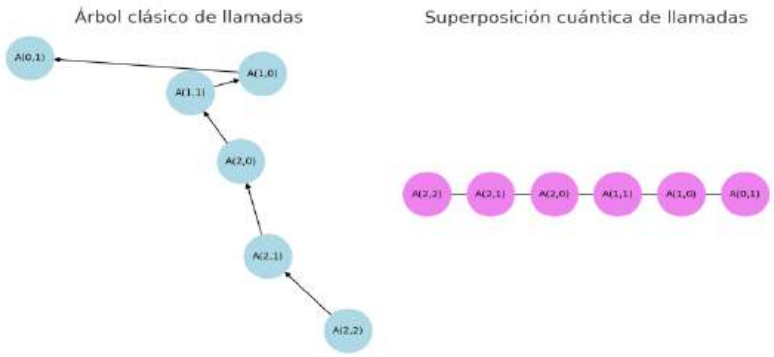

Figura 8.6 Comparación entre el árbol clásico de llamadas de Ackermann y la idea de su superposición cuántica

Reflexión Final: La lección fundamental

La Paradoja de la Memoria

El estudio de la función de Ackermann muestra que, en ciertos casos extremos, la complejidad espacial puede superar ampliamente cualquier capacidad física concebible de

almacenamiento. La expansión directa de $A(4,2)$ produciría estructuras cuyo tamaño excede por órdenes de magnitud las escalas materiales asociadas al universo observable.

Importancia práctica y *Benchmarking*

A pesar de su apariencia de curiosidad puramente teórica, la función de Ackermann es una herramienta sumamente valiosa en la práctica informática y se emplea frecuentemente como un *benchmark* riguroso para evaluar el rendimiento de sistemas. Las pruebas de estrés basadas en esta función permiten observar métricas cruciales (ver tabla 8.8)

Tabla 11 ***Aspectos evaluados en un benchmark con la función de Ackermann.***

Recurso Evaluado	Qué se observa en el Benchmark
Profundidad de pila	Límite máximo de los parámetros m y n que el sistema puede procesar antes de sufrir un *stack overflow*.
Optimizaciones del compilador	Detección de la capacidad del lenguaje para aplicar *Tail Call Optimization* (TCO), un reto enorme dado el anidamiento de Ackermann.
Costo de recursión	Medición de la sobrecarga real (*overhead*) de memoria y CPU que genera cada llamada a función en el sistema.
Eficiencia del *Runtime*	Velocidad comparativa del entorno de ejecución al procesar código intensivo (ej. C frente a Python o Java).

El recorrido a través de estos cinco algoritmos demuestra de manera contundente que la complejidad computacional no es un mero detalle de implementación, sino una propiedad matemática intrínseca de los problemas. Al transitar desde
algoritmos altamente eficientes hasta funciones matemáticamente inalcanzables en la práctica, se hace evidente que la escritura de código está gobernada por leyes estructurales inquebrantables.

Los primeros casos estudiados ilustran la elegancia de la tratabilidad. El algoritmo de Euclides y el ordenamiento Merge Sort muestran cómo la recursividad, cuando se diseña bajo principios como divide y vencerás o mediante reducciones proporcionales, produce soluciones con complejidades logarítmicas (O(log n)) y linealítmicas (O(n log n)). Estos constituyen los cimientos sobre los que se sostiene la viabilidad del software moderno.

Sin embargo, la frontera de lo tratable se cruza rápidamente. Las Torres de Hanói y el cálculo del determinante por cofactores revelan cómo un conjunto mínimo de reglas o una definición matemática aparentemente inocua pueden desencadenar un crecimiento exponencial (2^n) y factorial (O(n!)). Estos algoritmos funcionan como advertencias estructurales: la corrección lógica de un procedimiento no garantiza en absoluto su viabilidad física. Frente a estas complejidades, ningún incremento en la capacidad de cómputo logra compensar la explosión combinatoria inherente al problema.

Finalmente, la función de Ackermann lleva esta exploración a sus últimas consecuencias. Al romper las fronteras de la recursión primitiva, nos enfrenta a un crecimiento superexponencial que desafía tanto la intuición como los límites del hardware. Su valor no reside en el cálculo de sus resultados, sino en su papel como referencia extrema: un punto en el que la teoría de la computación se encuentra con las restricciones físicas de la ejecución.

En conjunto, este catálogo de algoritmos no solo ilustra distintos órdenes de crecimiento, sino que revela una estructura más profunda: la computación no se organiza únicamente entre lo posible y lo imposible, sino entre lo computable, lo eficiente y lo intratable. Resolver un problema no implica poder hacerlo en la práctica, y poder hacerlo en la práctica no implica que siempre sea posible hacerlo de manera general.

Así, el recorrido del libro encuentra aquí su punto de convergencia. Desde los primeros intentos por formalizar el cálculo hasta las teorías modernas de la complejidad, se ha trazado una misma línea: comprender qué significa calcular es, en última instancia, comprender sus límites. La informática aparece entonces no solo como una disciplina orientada a resolver problemas, sino como una ciencia que delimita el alcance mismo de la resolución.

La computación no se define solo por lo que puede hacer, sino por aquello que nunca podrá calcular; comprenderla es comprender sus límites: allí donde el algoritmo termina, comienza el verdadero alcance del pensamiento.

Bibliografía

Agüero Macken, E. (2004). *Giuseppe Peano y la utopía del lenguaje.* Madríd, España: Universidad Nacional de Educacion a Distancia (UNED).

Aho, A. V. (1974). *The Design and Analysis of Computer Algorithms.* EEUU: Addison-Wesley.

Andew S. Tanenbaum, H. B. (2015). *Modern Operating Systems.* New Jersey: Pearson Prentice Hall.

Báez, L. (s. f.). *Algoritmos y Estructura de Datos*. Obtenido de https://www.luchonet.com.ar/aed/?page_id=241

Cairó/Gardati. (2000). *Estructura de Datos.* México: Mc Graw hill.

Calculo de un determinante mediante recursividad en MATLAB. (20 de 06 de 2025). Obtenido de MATLAB TYP: https://matlab-typ.blogspot.com/2014/08/calculo-de-un-determinante-mediante.html

CHUANG, M. A. (2010). *Quantum Computation and Quantum Information.* Cambridge,: CAMBRIDGE UNIVERSITY PRESS.

Churh, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics,*, 345-363.

Científicas, V. (05 de 06 de 2025). *Rózsa Péter (1905-1977): jugando con el infinito*. Obtenido de Mujeres con ciencia: https://mujeresconciencia.com/2017/12/21/rozsa-peter-1905-1977-jugando-con-el-infinito/

Comte, A. (2014). *Curso de filosofía positiva.* Buenos Aires, Argentina: Claridad.

Dedekind, R. (2014). *¿Qué son y para qué sirven los números?* España: Alianza editorial. Ediciones de la Universidad Autónoma de Madrid.

Dewdney, A. (1993). *The New Turing Omnibus.* New York: Computer Science Press.

Dewdney, A. K. (1989). *The Turing Omnibus. 61 Excursions in Computer Science.* Estados Unidos: Computer Science Press.

Dijkstra, E. W. (1976). *A Discipline of Programming.* EEUU: Prentice Hall.

Documentation Archive. (21 de 09 de 2025). *Thread Management*. Obtenido de https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/CreatingThreads/CreatingThreads.html?utm_source=chatgpt.com

Ellis Horowitz, S. S. (1978). *Fundamentals of Computer Algorithms.* United States of America: Computer Science Press.

Gabriel, R. P. (1985). *Performance and Evaluation of Lisp Systems.* California: The MIT Press in their Computer Systems Series.

Gabriel, R. P. (1998). *Patterns of Software: Tales from the Software Community.* EEUU: Oxford University Press.

Giuliano Benenti, G. C. (2007). *Principles of Quantum Computation and Information.* Singapore.: World Scientific Publishing Co. Pte. Ltd.

Gonzales, Á. L. (2012). *METAFÍSICA MODAL EN G. W. LEIBNIZ.* España: Cuadernos de Anuario Filosófico. Departamento de Filosofía. Universidad de Navarra.

Harold Abelson, G. J. (1996). *Structure and Interpretation of Computer Ptograms.* EEUU: The Massachuse□s Institute of Technology.

Hilbert, D. (2011). *Fundamentos de las Matemáticas.* México: UNAM.

Hofstadter, D. R. (2015). *GÖDEL, ESCHER, BACH: UN ETERNO Y GRACIL BUCLE.* Barcelona: Booket.

Hull, L. W. (2013). *Historia Y Filosofía De La Ciencia.* Barcelona, España: Critica.

IBM Redbooks. (2020). *IBM z15 (8562) Technical Guide.* EEUU: IBM.

Ignite, M. (24 de 09 de 2025). *Microsoft Learn. Cree posibilidades.* Obtenido de https://learn.microsoft.com/es-mx/

John L. Hennessy, D. A. (2019). *Computer Architecture A Quantitative Approach.* United States: MORGAN KAUFHAN.

Jon Keinberg, E. T. (2006). *Algorithm Design.* Boston: Pearson Education Inc.

KerrisK, M. (2010). *The Linux Programming inTerface A Linux and UNIX. System Programming Handbook.* San Francisco: no starch press.

Kleen., S. C. (05 de 06 de 2025). *Reflexiones sobre la tesis de Churc*. Obtenido de bibliotecadigital.ilce.: bibliotecadigital.ilce.edu.mx/Colecciones/ReinaCiencias/_docs/ReflexionessobrelatesisdeChurch.pdf

kleene, S. C. (1967). *Mathematical Logic.* New York: John Wiley & Sons, Inc.

Kneale, W. (1972). *El desarrollo de la lógic.* España: Tecnos.

KOZEN, D. C. (1997). *Automata and Computability. UNDERGRADUATE TEXTS IN COMPUTER SCIENCE.* the United States of Americal: springer.

Langhlz, G., Francioni, J., & Kandel, A. (1989). *ELEMENTS OF COMPUTER ORGANIZATION.* New York: Prentice Hall.

López, A. S. (24 de 09 de 2025). *Introducción al análisis de algoritmos* . Obtenido de FCC/BUAP Grupo MOVIS : chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cs.buap.mx/~asanchez/Ada/introduccionADA.pdf

Matiyasevich, Y. V. (05 de 06 de 2025). *SCHOLARPEDIA*. Obtenido de http://www.scholarpedia.org/article/Matiyasevich_theorem

Murcia, A. d. (20 de 02 de 2025). *Pensandolo bien*. Obtenido de ARS MAGNA : https://portales.um.es/web/acc/-/sin-t%C3%ADtulo-contenido-web-b%C3%A1sico-2/1.4

Penrose, R. (1989). *The emperor's new mind: Concerning computers, minds, and the laws of physics.* Inglaterra: Oxford University Press.

Penrose, R. (1994). *Shadows of the mind: A search for the missing science of consciousness.* Inglaterra: Oxford University Press.

Post, E. L. (1936). *Finite Combinatory Processes-Formulation 1.* EEUU: Journal of Symbolic Logic.

Post, E. L. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Matematics*, 197-215.

Preskill, J. (30 de 07 de 2018). *Institute for Quantum Information and Matter and Walter Burke Institute for Theoretical Physics, California Institute of Technology, Pasadena CA 91125, USA*. Obtenido de chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://arxiv.org/pdf/1801.00862

Python. (24 de 09 de 2025). *Python 3.13.7 documentation*. Obtenido de https://docs.python.org/3/

Recursión y teorema maestro. (19 de 06 de 2025). Obtenido de Matemáticas discretas: https://madi.nekomath.com/P4/TeoremaMaestro.html

ScienceNews. (s.f.). *Kurt Gödel*. Obtenido de ScienceNews: https://scimyst.com/kurt-godel-great-mathematician-on-par-with-albert-einstein-92193/

Skiena, S. S. (2020). *The Algorithm Design Manual.* Switzerland: Springer Nature Switzerland.

Stack Exchange. (s. f.). *Code Review*. Obtenido de Ackermann Function of (4,2): https://codereview.stackexchange.com/questions/107130/ackermann-function-of-4-2

Stanford University. (03 de 09 de 2025). *Stanford Encyclopedia of Philosophy*. Obtenido de Recursive Functions: https://plato.stanford.edu/entries/recursive-functions/#EarlHistRecuDefi

TechTarget. (24 de 09 de 2025). *Definition x86-64*. Obtenido de https://www.techtarget.com/whatis/definition/x86-64?utm_source=chatgpt.com

Turing, A. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 230-266.

Universidad de Granada. (27 de 08 de 2025). *Dto Algebra, Facultad de Ciencias.* Obtenido de Wilhelm Ackermann: https://www.ugr.es/~eaznar/ackerman.htm

Wikipedia. (s.f). *NP-hardness.* Obtenido de https://en.wikipedia.org/wiki/NP-hardness

Wolfram mathWord. (23 de 07 de 2025). *Knuth up-arrow notation*. Obtenido de Wolfram MathWord: https://mathworld.wolfram.com/KnuthUp-ArrowNotation.html