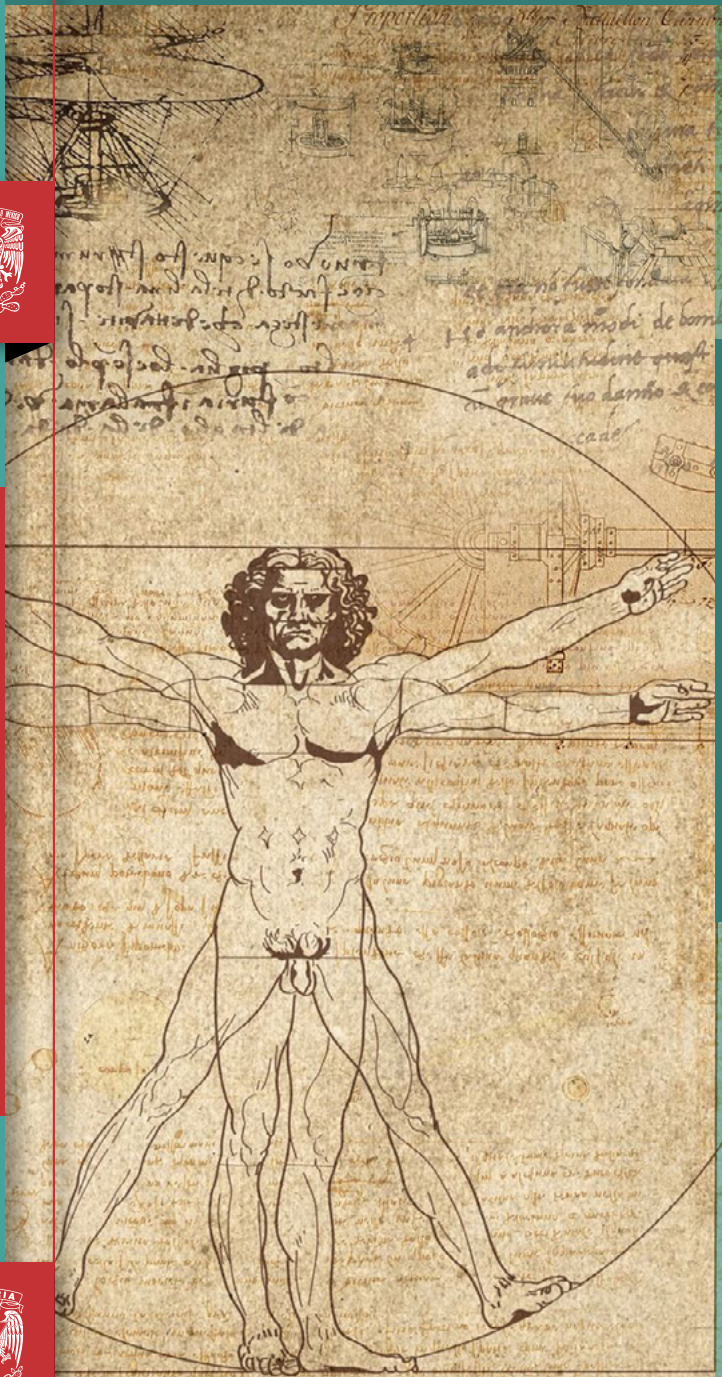
UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

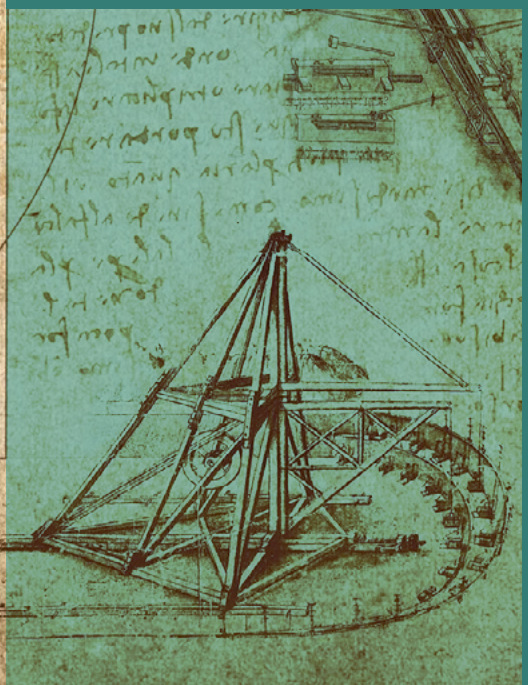
FACULTAD DE INGENIERÍA

ANÁLISIS DE ALGORITMOS Y OPTIMIZACIÓN

Idalia Flores de la Mota

DIVISIÓN DE INGENIERÍA MECÁNICA E INDUSTRIAL





CUADERNILLO DE DIVULGACIÓN 18

ANÁLISIS DE ALGORITMOS Y OPTIMIZACIÓN

Idalia Flores de la Mota

CUADERNILLO
DE DIVULGACIÓN
18

Flores, De la Mota, Idalia
Análisis de algoritmos y optimización
Universidad Nacional Autónoma de México,
Facultad de Ingeniería, 2022, 139 p.

Análisis de algoritmos y optimización

Primera edición electrónica
de un ejemplar (5 MB) Formato PDF
Publicado en línea el 29 de agosto de 2022

FACULTAD DE INGENIERÍA
http://www.ingenieria.unam.mx/

Hecho en México.

Unidad de Apoyo Editorial
Cuidado de la edición: María Cuairán Ruidíaz
Formación y diseño editorial: Nismet Díaz Ferro

PRÓLOGO

Como parte de las actividades del Departamento de Investigación de Operaciones e Ingeniería Industrial, de la División de Ingeniería Mecánica e Industrial de la Facultad de Ingeniería, UNAM, nos hemos propuesto el desarrollo de una serie de cuadernillos de divulgación que sirvan de apoyo a los diferentes cursos que se imparten y, desde luego, como material de referencia o de lectura para quienes así lo requieran.

En dichos cuadernillos se busca mantener un alto nivel, de manera que contribuyan a una sólida formación teórica del alumno, pero al mismo tiempo se tiene como objetivo desarrollar los temas con sencillez para un buen entendimiento de los temas y finalmente acercar al lector a las aplicaciones de tales conocimientos, como es en síntesis el objetivo central del posgrado de ingeniería.

Estos cuadernillos tienen como objetivo aportar conocimientos básicos a los alumnos de las maestrías en Sistemas, así como a los alumnos de licenciatura interesados en el tema del Análisis de algoritmos y optimización.

Fue tarea agradable la redacción de este material y la autora espera que los estudiantes de ingeniería lo encuentren grato e informativo, cuando traten de aprender cómo aplicar análisis de algoritmos en sus campos de interés.

Agradezco a la Unidad de Apoyo Editorial de la Facultad, y a su jefa la Lic. Patricia Eugenia García Naranjo, y en especial a la Maestra María Cuairán Ruidíaz por la revisión de este material, a la LDG Nismet Díaz Ferro por la dedicación y disposición para el diseño de la portada, al Ing. Juan Pablo Cisneros Castañeda y al Act. Alejandro Felipe Zárate Pérez por su colaboración en las gráficas, figuras y en la revisión teórica de los conceptos desarrollados. Y finalmente al proyecto PE107421: Una visión holística de la ingeniería y las matemáticas: Retos actuales en la enseñanza.

1
2
3
4
5
6
7
8
9
10
11
12
13

SEMBLANZA DE LA AUTORA

La Dra. Idalia Flores es profesora titular B de tiempo completo con nivel de PRIDE C, y cuenta con la siguiente formación académica: Hizo sus estudios de doctorado en Investigación de Operaciones en la División de Estudios de Posgrado de la Facultad de Ingeniería, UNAM, se graduó en 1998. Los estudios de maestría también los realizó en Investigación de Operaciones en la misma división y facultad, se graduó con mención honorífica en 1990. Es matemática por la Facultad de Ciencias de la UNAM desde 1985.

Sus líneas de investigación son: **Optimización y simulación aplicadas a procesos industriales y de transporte.**

En cuanto a docencia ha impartido más de 15 asignaturas diferentes en posgrado y licenciatura de la Facultad de Ingeniería desde 1990.

Ha publicado dos series de cuadernillos, 6 entre 2004 y 2006 y otros 10 entre 2012-2014, así como coeditora de otros 7 cuadernillos con las maestras Ann Wellens y Francisca Irene Soler.

Ha publicado desde 1990 una serie de 6 apuntes para apoyo de los cursos de posgrado.

Programación dinámica, Facultad de Ingeniería, UNAM. (2015).
https://www.ingenieria.unam.mx/publicaciones/libros/libro_58.php

Álgebra lineal y sus aplicaciones (2021)
https://www.ingenieria.unam.mx/publicaciones/libros/libro_49.php

Coeditora y autora del libro *Applied simulation and optimization*. Editorial Springer (2014-2015). http://www.springer.com/la/book/9783319150321

Coeditora y autora del libro *Applied simulation and optimization 2*. Editorial Springer (2017).
https://link.springer.com/book/10.1007/978-3-319-55810-3

Coeditora y autora del libro *Robust modelling and simulation.* Editorial Springer (2017).
https://link.springer.com/book/10.1007/978-3-319-53321-6

Coeditora y autora del libro *Modelos de simulación usando SIMIO y redes de Petri.* Edición Kindle.
https://www.amazon.com.mx/Modelos-Simulaci%C3%B2n-usando-SIMIO-Redes-ebook/dp/B01M70WZFT/ref=sr_1_1?ie=UTF8&qid=1496711921&sr=8-1&keywords=idalia+flores+de+la+mota

ÍNDICE

Prólogo 4
Semblanza de la autora 6
Introducción 9
Glosario 11

1. Introducción 16
2. Complejidad de algoritmos: tiempo y espacio 21
3. Peor caso y caso probabilístico 40
4. Análisis asintótico de funciones 45
5. Velocidad de crecimiento y cálculo de tiempo de ejecución de un algoritmo 55
6. Tiempo de ejecución de un programa 59
7. Representación de los algoritmos 64
8. Complejidad de los problemas. 67
9. Problemas de optimización 97
10. Conclusiones 117
11. Notas históricas 120
12. Bibliografía 127
13. Anexo 130

INTRODUCCIÓN

1
2
3
4
5
6
7
8
9
10
11
12
13

En ciencias de la computación, el diseño de algoritmos es un método específico para poder crear un modelo matemático ajustado a un problema concreto para resolverlo. El diseño de algoritmos es un área central de las ciencias de la computación, también es muy importante para la Investigación de Operaciones, en ingeniería del software y en otras disciplinas afines.

La Investigación de Operaciones es una disciplina que se ocupa de la aplicación de métodos analíticos avanzados para ayudar a tomar mejores decisiones. A menudo se considera que es un subcampo de las matemáticas aplicadas, ya que se hace uso de modelos matemáticos y de algoritmos para resolver problemas complejos. En este sentido, el presente documento expone de manera concisa la teoría y ejemplos del diseño y análisis de algoritmos para la selección de métodos más eficientes en la solución de problemas.

1
2
3
4
5
6
7
8
9
10
11
12
13

Aunque en la Investigación de Operaciones se resuelven problemas estocásticos que involucran modelos de simulación, aquí nos enfocaremos al área de la optimización, si bien la estructura para resolver problemas de optimización y simulación es muy similar y de hecho se complementan, no es objetivo de este documento el análisis de algoritmos para resolver modelos de simulación.

La optimización se ocupa de determinar los valores extremos de algún objetivo del mundo real: los máximos (de ganancia, rendimiento o rentabilidad) o mínimos (de pérdida, riesgo o costo), asimismo considera el manejo de estructuras de datos que actualmente son muy grandes y para lo que se requiere el uso eficiente de algoritmos de búsqueda, patrones (minería de datos) y análisis (*big data*) de los datos.

La optimización como la Investigación de Operaciones se originó en los esfuerzos militares de la Segunda Guerra Mundial para la administración eficiente de recursos. Posteriormente, y con países deprimidos económicamente por la guerra se hizo necesario que sus técnicas se desarrollaran a la par de la computación para tratar problemas en distintas industrias.

Aun con el acelerado desarrollo de los algoritmos y de las computadoras veremos en este escrito que hay problemas para los cuales no es sencillo encontrar buenas soluciones y que para arribar a ellas se requiere de conocimientos en algoritmos y creatividad.

La complejidad computacional de los algoritmos es un tema sumamente interesante y a veces choca con nuestra manera de «ver el mundo», por lo cual se discute ampliamente, sobre todo lo relativo a la pregunta P vs. NP. En los casos en donde ha sido posible se recomiendan videos que ayuden a la comprensión de los temas.

GLOSARIO

Algoritmo. Para que una computadora lleve a cabo una tarea es preciso decirle qué operaciones debe realizar, es decir, debemos describir cómo debe realizar la tarea. Dicha descripción se llama algoritmo.

Procesador es el agente que interpreta y realiza las instrucciones de un algoritmo.

Tiempo de ejecución es número de operaciones elementales realizadas por un algoritmo.

Un **problema** es un conjunto de instancias al cual corresponde un conjunto de soluciones a través de una relación que asocia para cada instancia del problema un subconjunto de soluciones (posiblemente vacío).

Una **instancia de un problema** se obtiene cuando se especifican valores particulares para todos los parámetros del problema.

1
2
3
4
5
6
7
8
9
10
11
12
13

Un **problema computacional** constituye una pregunta para ser respondida, que tiene generalmente varios parámetros, o variables libres, cuyos valores no se han especificado.

Si f y g son dos funciones de N en R, se dice que a f **g domina asintóticamente** (o simplemente que g domina a f) si existen enteros $k \geq 0$ y $m \geq 0$ tales que se verifica la desigualdad: $|f(n)| \leq k|g(n)|$ para todo entero $n \geq m$.

Un **problema de optimización** tiene como objetivo encontrar una solución que sea óptima entre todas las soluciones factibles que se tengan. Estas soluciones factibles se pueden considerar como decisiones factibles, y están sujetas a una serie de restricciones que deben cumplir.

Un **problema de decisión** es un tipo especial de problema computacional cuya respuesta es solamente "sí" o "no" (o, de manera más formal, "1" o "0").

Problemas indecidibles. Son aquellos problemas para los cuales no se puede escribir un algoritmo. Un problema indecidible es aquel que debería dar una respuesta de "sí" o "no", pero para el cual todavía no existe un algoritmo que dé la respuesta correcta para todas las entradas posibles.

Problemas de búsqueda y decisión. El primero se refiere a encontrar soluciones a instancias dadas, mientras que el segundo se refiere a determinar si la instancia dada tiene una propiedad predeterminada.

Máquinas de Turing. El modelo de las máquinas de Turing ofrece una formulación relativamente simple de la noción de algoritmo.

Teorema 1. El problema de la parada es indecidible.

Problemas intratables (problemas que se demuestran son difíciles). Son aquellos problemas para los cuales no se pueden desarrollar algoritmos polinomiales. En otras palabras, solo se pueden resolver con algoritmos exponenciales.

Algoritmos universales. Una máquina universal calcula la función parcial u que se define en pares $(<M>, x)$ tal que M se detiene en la entrada x, en cuyo caso se cumple que $u(<M>, x) = M(x)$.

Eficiente e ineficiente. La eficiencia está asociada con los cálculos de tiempo polinomial, mientras que los cálculos que requieren más tiempo se consideran ineficientes o intratables (o inviables, no factibles).

La clase P. La clase de problemas de decisión que se resuelven eficientemente.

La clase NP. La clase de problemas de decisión que tienen sistemas de prueba eficientemente verificables.

Teorema 2 $PC \subseteq PF$ si y solo si $P = NP$.

La pregunta P-vs-NP. Se cree ampliamente que P es diferente de NP. Esta creencia está respaldada por consideraciones tanto filosóficas como empíricas.

La **definición tradicional de NP.** Tradicionalmente, NP se define como la clase de conjuntos que pueden decidirse mediante un dispositivo ficticio llamado máquina de tiempo polinomial no determinista (que explica el origen de la notación NP).

Cook-reducible. Un problema Π es Cook-reducible a un problema Π' si Π se puede resolver de manera eficiente cuando se le da acceso a cualquier procedimiento (u oráculo) que resuelva el problema Π'.

Karp-reducible. Un problema de decisión S es Karp-reducible a un problema de decisión S' si existe una función computable en tiempo polinomial f tal que, para cada x, se cumple que $x \in S$ si y solo si $f(x) \in S'$.

Tesis de Cobham-Edmonds. Afirma que los problemas computacionales se pueden calcular de manera factible en algún dispositivo computacional solo si se pueden calcular en tiempo polinómico; es decir, si se encuentran en la clase de complejidad *P*.

Teoremas 3 y 4 (también conocido como **teorema de Cook-Levin**). La satisfactibilidad del circuito (CSAT) y la satisfactibilidad de la fórmula (SAT) son NP-completos.

“

Ver un mundo en un grano de arena
Y un cielo en una flor silvestre
Prender el infinito en la palma de tu mano
Y la eternidad en solo una hora”

William Blake

1 Introducción

Hace unos años, antes de la llegada de los videojuegos y las computadoras domésticas, los juegos de computadora eran un regalo relativamente raro que se encontraba en algún museo de los niños. Uno de los más difundidos fue un simple «Adivina el número» que se jugaba de la siguiente manera:

—¡Hola! ¡Bienvenido a Adivina el número! Estoy pensando en un número en el rango de 1 a 100. Intenta adivinarlo.
—¿Cuál es tu conjetura? 20
—Eso es demasiado pequeño, inténtalo de nuevo
—¿Cuál es tu conjetura? 83
—Eso es demasiado grande, inténtalo de nuevo.

El juego continuaba de esta manera, aceptando nuevas conjeturas del jugador, hasta que se descubría el número secreto.

—¿Cuál es tu conjetura? 37

—¡Felicidades! ¡Lo conseguiste en 12 intentos!

Para los niños que estaban felices de pasar un poco más de tiempo con uno de estos juegos, doce conjeturas no parecían excesiva s. Eventualmente, sin embargo, incluso jugadores relativamente jóvenes descubrirían que podían hacerlo un poco mejor que esto mediante la explotación de una estrategia más sistemática.

Para desarrollar tal estrategia, la idea central es que cada conjetura debe reducir el rango de búsqueda lo más rápido posible. Esto se logra eligiendo el valor más cercano a la mitad del rango disponible. El problema original se puede expresar como:

—Adivina un número en el rango de 1 a 100.

Si adivinamos 50 y descubrimos que es muy grande entonces se reduce el problema a:

—Adivina un número en el rango de 1 a 49.

Esto tiene el efecto de reducir el problema original a un subproblema idéntico en el que el número se limita a un rango más restringido. Eventualmente, debemos adivinar el número correcto, ya que el rango se hará más y más pequeño hasta que solo queda una única posibilidad.

En el lenguaje de la informática, este algoritmo se llama búsqueda binaria y es un ejemplo de la estrategia recursiva «divide y vencerás». Para este problema, la búsqueda binaria parece funcionar razonablemente bien.

Por otro lado, ciertamente no es el único enfoque posible. Por ejemplo, cuando se le pide que encuentre un número en el rango de 1 a 100, ciertamente podríamos simplemente formular una serie de preguntas de la forma:

—¿Es el 1?, ¿el 2?, ¿el 3?...

Estamos obligados a darle con el tiempo, después de no más de 100 conjeturas. Este algoritmo se llama búsqueda lineal y se usa con bastante frecuencia en informática para encontrar un valor en una lista desordenada.

Intuitivamente, tenemos la sensación de que el mecanismo de búsqueda binaria es un mejor acercamiento al juego «Adivina el número», pero no estamos seguros de cuán mejor puede ser. Para tener algún estándar de comparación, debemos encontrar una manera de medir la eficiencia de cada algoritmo. En informática, esto se logra más a menudo mediante el cálculo de la complejidad computacional del algoritmo, que expresa el número de operaciones necesarias para resolver un problema en función del tamaño de ese problema.[1]

Para que una computadora lleve a cabo una tarea es preciso decirle qué operaciones debe realizar, es decir, debemos describir cómo debe realizar la tarea. Dicha descripción se llama **algoritmo**.

Un algoritmo describe el método mediante el cual se realiza una tarea. Un algoritmo consiste en una secuencia de instrucciones, las cuales, realizadas adecuadamente, dan lugar al resultado deseado.

1. Tomado de *Thinking recursively*. Eric Roberts, 1986.

La noción de algoritmo no es exclusiva de la computación o de las matemáticas. Existen algoritmos que describen toda clase de procesos de la vida real, por ejemplo, las recetas de cocina, las partituras, etc.

En todos los casos anteriores el ejecutor o procesador de las instrucciones que realiza la tarea correspondiente es el hombre. Sin embargo, el agente que interpreta y realiza las instrucciones de un algoritmo se llama **procesador**.

Un procesador puede ser una persona, una computadora, o cualquier otro sistema electrónico o mecánico. Un procesador realiza un proceso siguiendo, o ejecutando, el algoritmo correspondiente. La ejecución de un algoritmo requiere la ejecución de cada uno de los pasos o instrucciones que lo constituyen. De aquí se desprende que una computadora no es más que un tipo particular de procesador.

Como se ha dicho ya, para que un procesador lleve a cabo un proceso debe estar previamente provisto de un algoritmo adecuado. Por ejemplo, el cocinero debe conocer la receta, el pianista, la partitura, etc.

Si el procesador de un algoritmo es una computadora, el algoritmo se debe expresar en forma de programa. Un programa se escribe en un lenguaje de programación, y la actividad que consiste en expresar un algoritmo en un lenguaje de programación se llama **programar.**

Cada paso del algoritmo se expresa mediante una instrucción o sentencia en el programa. Por tanto, un programa consiste en una secuencia de instrucciones, cada una de las cuales especifica ciertas operaciones a realizar por la computadora.

Podría por tanto afirmarse que son más importantes los algoritmos que los lenguajes de programación e incluso las mismas computadoras, considerando que un lenguaje de programación es simplemente un medio conveniente para expresar un algoritmo y una computadora es simplemente un procesador para ejecutarlo; es decir, tanto los lenguajes de programación como las computadoras son medios para lograr un fin: ejecutar un algoritmo.

Con esto no se pretende restar importancia a las computadoras y a los lenguajes de programación. Los avances en la tecnología informática permiten día a día ejecutar algoritmos más rápidamente y con más precisión y a mejor precio.

Un aspecto importante es el **diseño de algoritmos**. ¿Cómo diseñar buenos algoritmos? El diseño de buenos algoritmos requiere creatividad e ingenio y no existen, en general, reglas para diseñar algoritmos. En otras palabras, no existe un algoritmo para diseñar algoritmos.

Si existen varios algoritmos para resolver un problema, ¿cuál de ellos es el «mejor», en el sentido de que necesita menos recursos informáticos? ¿Cuáles son los mínimos recursos informáticos necesarios para llevar a cabo una tarea determinada? (Es decir, ¿qué recursos utilizará el mejor algoritmo posible para realizar dicha tarea?). ¿Se puede averiguar cuál es el mejor algoritmo? ¿Existen problemas para los cuales el mejor algoritmo posible requerirá tantos recursos que hará inviable su ejecución incluso con la computadora más grande y rápida existente?

Todas estas cuestiones se engloban para su estudio bajo el título:
Complejidad de algoritmos.

2 Complejidad de algoritmos: tiempo y espacio

Como sabemos, un programa es una representación de un algoritmo en un lenguaje de programación que puede interpretar y ejecutar una computadora.

La manera de representar un algoritmo en forma de programa no es en general única. Asimismo, para resolver un problema para el cual existe un algoritmo, no es único el algoritmo que lo resuelve. Cabe por tanto preguntarse cuál es el mejor algoritmo de entre dos algoritmos dados que resuelven el mismo problema.

Una posible alternativa para contestar dicha cuestión consiste en representar los dos algoritmos mediante un lenguaje de programación, a continuación, ejecutarlos en una computadora y medir el tiempo requerido por cada uno de ellos para obtener la solución de un mismo problema particular.

El **tiempo** de ejecución requerido por un algoritmo para resolver un problema es uno de los parámetros importantes en la práctica para medir la bondad de un algoritmo pues, entre otros factores, el tiempo de ejecución equivale a tiempo de utilización de la computadora y, en consecuencia, costo económico.

Es más, si el tiempo de ejecución es demasiado grande, puede suceder que el algoritmo sea en la práctica inútil, pues el tiempo necesario para su ejecución puede sobrepasar el tiempo disponible de la computadora.

Resulta evidente que el tiempo real requerido por una computadora para ejecutar un algoritmo es directamente proporcional al número de operaciones básicas elementales que la misma debe realizar en su ejecución, medir por tanto el tiempo real de ejecución equivale a medir el número de operaciones elementales realizadas. (Nosotros supondremos desde ahora que todas las operaciones básicas se ejecutan en una unidad de tiempo. Para una mayor precisión habría que distinguir los tiempos de ejecución de cada una de las distintas operaciones elementales). Por esta razón se suele llamar **tiempo** de ejecución no al tiempo real físico, sino al número de operaciones elementales realizadas.

2.1 Clasificación y tipos de algoritmos

Para entender una clasificación de algoritmos recuerde que en la primera sección de este escrito se mencionaron métodos de búsqueda, y el diseño de algoritmos.

1. Métodos o algoritmos de búsqueda

Cuando se manipulan conjuntos de datos, la búsqueda de valores se convierte en una operación de vital importancia, cuya resolución, en ocasiones, no es trivial. Los métodos de búsqueda tienen como objetivo la localización de un elemento con ciertas propiedades dentro de la estructura de datos, por ejemplo, ubicar el registro correspondiente a cierta persona en una base de datos, o el mejor movimiento en una partida de ajedrez.

Los ya mencionados son búsqueda secuencial y búsqueda binaria

a. Búsqueda secuencial: Consiste en ir comparando el elemento que se busca con cada elemento del arreglo hasta que se encuentra.
b. Una búsqueda más eficiente puede hacerse sobre un arreglo ordenado. Una de estas es la búsqueda binaria. La búsqueda binaria compara si el valor buscado está en la mitad superior o inferior. En la que esté, subdivido nuevamente, y así sucesivamente hasta encontrar el valor.

La búsqueda binaria es mejor opción con respecto a la búsqueda secuencial cuando el conjunto de datos de entrada *n* crece. Para valores pequeños de *n* la búsqueda secuencial es atractiva dada la sencillez de su implementación.

2. Métodos de ordenamiento

Los algoritmos de ordenamiento nos permiten, como su nombre lo dice, ordenar información de una manera especial basándonos en un criterio de ordenamiento.

En la computación el ordenamiento de datos cumple un rol muy importante, ya sea como un fin en sí o como parte de otros procedimientos más complejos. Se han desarrollado muchas técnicas en este ámbito, cada una con características específicas, y con ventajas y desventajas sobre las demás.

a. El ordenamiento por inserción es muy natural para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria. La idea de este algoritmo de ordenamiento consiste en ir insertando un elemento de la lista o un arreglo en la parte ordenada de ella misma, asumiendo que el primer elemento es la parte ordenada, el algoritmo irá comparando un elemento de la parte desordenada de la lista con los elementos de la parte ordenada, insertando el elemento en la posición correcta dentro de la parte ordenada, y así sucesivamente hasta obtener la lista ordenada.

   En la página consultada y que está al pie de página se puede ver una representación animada de este método.[2]

b. El ordenamiento por burbuja funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Y su representación animada.[3]

c. El ordenamiento de burbuja bidireccional (también llamado «método de la sacudida» o *coctail sort* o *shaker sort*) es un algoritmo de ordenamiento que surge como una mejora del algoritmo ordenamiento de burbuja.

2 http://lwh.free.fr/pages/algo/tri/tri_insertion_es.html
3 http://lwh.free.fr/pages/algo/tri/tri_bulle_es.html

Ya vimos cómo funciona el algoritmo de ordenación por burbuja, entonces se observa que los números grandes se están moviendo rápidamente hasta al final de la lista (estas son las «liebres»), pero que los números pequeños (las «tortugas») se mueven solo muy lentamente al inicio de la lista.

Una solución es ordenar con el método de burbuja y cuando llegamos al final de la primera iteración, no volver a realizar el cálculo desde el principio, sino que empezaremos desde el final hasta al inicio. De esta manera siempre se consigue que tanto los números pequeños como los números grandes se desplacen a los extremos de la lista lo más rápido posible. Para ver animación de este método.[4]

d. Ordenación «gnome». El algoritmo de ordenación conocido como *gnome_sort* fue inventada por Hamid Sarbazi-Azad (profesor de la universidad de Sharif, una de las mayores universidades de Irán), quien lo desarrolló en el año 2000 y al que llamó *stupid sort* (ordenamiento estúpido).

   Cuando Dick Grune lo reinventó y documentó, no halló evidencias de que existiera y en palabras suyas, dijo de él: «*the simplest sort algorithm*» (es el algoritmo más simple) y quizás tenga razón, pues lo describió en solo cuatro líneas de código. Dick Grune se basó en los gnomos de jardín holandés, en cómo se colocan en los maceteros y de ahí también el nombre que le dio.

4 http://lwh.free.fr/pages/algo/tri/tri_shaker_es.html

El algoritmo es similar a la ordenación por inserción, excepto que, en lugar de insertar directamente el elemento en su lugar apropiado, el algoritmo realiza una serie de permutaciones, como en el ordenamiento de burbuja. Para ver animación.[5]

e. Mezcla de arreglos consiste en generar un arreglo ordenado a partir de otros ordenados con anterioridad. En lugar de concatenar los dos subarreglos en otro mayor y ordenar el conjunto, se aprovecha que ya existe un orden parcial en las entradas.

   El mecanismo de mezcla consiste en comparar sendos elementos de cada uno de los arreglos de entrada y se coloca el más pequeño en el arreglo destino, tomando el elemento siguiente del arreglo correspondiente, hasta agotar todos los elementos de uno de ellos. Por último, se copiarán los elementos no procesados del otro arreglo.

Nota: Para más videos acceder a:
https://www.youtube.com/watch?v=aXXWXz5rF64

3. Técnicas para el diseño de algoritmos[6]

Aunque en la solución de problemas sencillos parezca evidente la codificación en un lenguaje de programación concreto, es aconsejable realizar el diseño del algoritmo, a partir del cual se codifique el programa. Las soluciones a problemas más complejos pueden requerir muchos más pasos.

5 http://lwh.free.fr/pages/algo/tri/tri_gnome_es.html
6 https://programas.cuaed.unam.mx/repositorio/moodle/pluginfile.php/1196/mod_resource/content/1/contenido/index.html

Las estrategias seguidas usualmente a la hora de encontrar algoritmos para problemas complejos son:

a. **Partición o divide y vencerás:** consiste en dividir un problema grande en unidades más pequeñas que puedan ser resueltas individualmente. Las rutinas en las cuales el texto contiene al menos dos llamadas recursivas se denominan algoritmos de divide y vencerás; no así aquellas cuyo texto solo comprende una.

   La idea de la técnica divide y vencerás es dividir un problema en subproblemas del mismo tipo y, aproximadamente, del mismo tamaño; resolver los subproblemas recursivamente y combinar la solución de los subproblemas para dar una solución al problema original.

   La recursión finaliza cuando el problema es pequeño y la solución fácil de construir directamente.

   Ejemplo: Podemos dividir el problema de limpiar una casa en labores más simples correspondientes a limpiar cada habitación.

b. **Algoritmos voraces o glotones.** Suelen utilizarse en la solución de problemas de optimización y se distinguen porque son: sencillos en cuanto a su diseño y codificación. Miopes, toman decisiones con la información disponible de forma inmediata, sin tener en cuenta sus efectos futuros y eficientes, ya que dan una solución rápida al problema (aunque esta no sea siempre la mejor).

   Tienen las siguientes propiedades:
   - Tratan de resolver problemas de forma óptima.
   - Disponen de un conjunto o lista de candidatos.

A medida que avanza el algoritmo se acumulan dos conjuntos:

» Candidatos considerados y seleccionados.
» Candidatos considerados y rechazados.

Los algoritmos voraces suelen ser bastante simples. Se emplean sobre todo para resolver problemas de optimización; por ejemplo, encontrar la secuencia óptima para procesar un conjunto de tareas por una computadora; hallar el camino mínimo de un grafo, etcétera. Por lo regular, intervienen estos elementos:

Un conjunto o lista de candidatos (tareas a procesar, nodos de una red).

Una función que determina si un conjunto de candidatos es una solución al problema (aunque no tiene que ser la óptima).

Una función que determina si un conjunto es completable, es decir, si añadiendo a este conjunto nuevos candidatos es posible alcanzar una solución al problema, suponiendo que esta exista.

Una función de selección que escoge al candidato aún no seleccionado que es más prometedor.

Una función objetivo que da el valor/costo de una solución (tiempo total del proceso, longitud de una ruta) y es la que se busca maximizar o minimizar.

c. **Programación dinámica.** Inventada por el matemático Richard Bellman en 1953, es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas. Una subestructura óptima significa que soluciones óptimas de subproblemas pueden ser usadas para encontrar las soluciones óptimas del problema en su conjunto.

   En general, se pueden resolver problemas con subestructuras óptimas siguiendo estos pasos:

   1. Dividir el problema en subproblemas más pequeños.
   2. Resolver estos problemas de la mejor manera, usando este proceso de tres pasos recursivamente.
   3. Aplicar estas soluciones óptimas para construir una solución óptima al problema original.

   Los subproblemas se resuelven, a su vez, dividiéndolos en subproblemas más pequeños, hasta alcanzar el caso fácil, donde la solución al problema es trivial.

d. **Resolución por analogía.** Dado un problema, se trata de recordar algún problema similar que ya esté resuelto. Los dos problemas análogos pueden incluso pertenecer a áreas de conocimiento totalmente distintas.[7]

   Ejemplo: El cálculo de la media de las temperaturas de los estados del Bajío y la media de las notas de los alumnos en una clase se realiza del mismo modo.

7 https://plataforma.josedomingo.org/pledin/cursos/programacion/curso/u03/

e. Todas estas estrategias para resolver problemas se usan en Investigación de Operaciones y mencionaremos algunas más como las siguientes:

- » Algoritmos probabilísticos: Presentan soluciones aproximadas del problema.
- » Algoritmos heurísticos: Son utilizados cuando no existe una solución por las vías tradicionales.
- » Algoritmo de escalada: Comienza con una solución insatisfactoria (que no cumple con la entrada / salida), y se va a acercar a lo que se busca.
- » Algoritmos paralelos: Permiten la división de un problema en subproblemas de forma que se puedan ejecutar simultáneamente en varios procesadores.
- » Algoritmos probabilísticos: Algunos de los pasos de este tipo de algoritmos están en función de valores pseudoaleatorios.
- » Algoritmos determinísticos: El comportamiento del algoritmo es lineal: cada paso del algoritmo tiene únicamente un paso sucesor y otro antecesor.
- » Algoritmos no determinísticos: El comportamiento del algoritmo tiene forma de árbol y a cada paso del algoritmo puede bifurcarse a cualquier número de pasos inmediatamente posteriores, además todas las ramas se ejecutan simultáneamente.
- » Metaheurísticas: Encuentran soluciones aproximadas (no óptimas) a problemas basándose en un conocimiento anterior (a veces llamado experiencia) de los mismos.
- » Programación dinámica: Intenta resolver problemas disminuyendo su coste computacional aumentando el coste espacial.
- » Ramificación y acotación: Se basa en la construcción de las soluciones al problema mediante un árbol implícito que se recorre de forma controlada para encontrar las mejores soluciones.

- Vuelta atrás (*backtracking*): Se construye el espacio de soluciones del problema en un árbol que se examina completamente, almacenando las soluciones menos costosas.
- Algoritmo determinista: Es completamente lineal (cada paso tiene un paso sucesor y un paso predecesor).

En cuanto al análisis de los algoritmos, que es lo que se presenta en el resto de este texto, podemos decir que son independientes tanto del lenguaje de programación en que se expresan como de la computadora que los ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecución de otra manera, pero el algoritmo es siempre el mismo.

Otro de los factores importantes, en ocasiones el decisivo, para comparar algoritmos es la cantidad de memoria de máquina requerida para almacenar los datos durante el proceso.

La cantidad de memoria utilizada por un algoritmo durante el proceso se suele llamar **espacio** requerido por el algoritmo.

Para entender la diferencia del espacio requerido por dos algoritmos que resuelven el mismo problema veamos un ejemplo.

Ejemplo 1

Dados *n* números naturales en forma secuencial, es decir, dados de uno en uno con un intervalo de tiempo entre uno y otro, encuentre cuál es el máximo de todos ellos.

Veamos entonces, dos algoritmos que resuelven el mismo problema:

ALGORITMO 1

Paso 1 Almacenar los n números utilizando n variables $a_1, \cdots, a_n$

Paso 2 Asignar $m = a_1$, $i = 2$

Paso 3 Si $m > a_i$ entonces $i = i + 1$ en caso contrario $m = a_i$, $i = i + 1$

Paso 4 Si $i > n$ el máximo es m; FIN. En caso contrario regresar al paso 3.

ALGORITMO 2

Paso 1 Asignar el primer elemento a la variable m.

Paso 2 Asignar el siguiente elemento a la variable a.

Paso 3 Si $a > m$ entonces $m = a$.

Paso 4 Mientras queden elementos volver al paso 2.

Paso 5 El máximo es m FIN.

Los dos algoritmos anteriores resuelven el problema del máximo. Sin embargo, el espacio requerido por el primero de ellos depende del valor de n. Cuanto mayor sea n mayor es el número de variables a las que hay que asignar valores en el paso 1 y, en consecuencia, mayor será la cantidad de espacio necesario.

El algoritmo 2 por el contrario utiliza solo las variables m y a independientemente de la cantidad de números, pues este segundo algoritmo antes de recibir un nuevo número calcula el máximo de los números anteriores manteniéndolo en la variable m.

Está claro que para que el segundo algoritmo pueda ejecutarse, el intervalo de tiempo que transcurre entre la entrada de dos números debe ser mayor o a lo sumo igual al tiempo requerido para efectuar las operaciones del paso 3.

En adelante nos ocuparemos solamente del **tiempo** requerido por un algoritmo para su ejecución.

Si observamos la fórmula propuesta para medir el tiempo requerido por un algoritmo, observamos que, al no ser única la manera de representarlo mediante un programa, y al no ser única tampoco la computadora en donde se ejecuta resulta que dicha medida será variable dependiendo fundamentalmente de los siguientes factores:

1. El lenguaje de programación elegido.
2. El programa que representa el algoritmo.
3. La computadora que lo ejecuta.

En definitiva, al existir gran cantidad de lenguajes, técnicas de programación y computadoras diferentes, resulta que la forma propuesta de medir el tiempo, y en consecuencia la bondad de un algoritmo no llega a ser adecuada.

Por esta razón surge la necesidad de medir el tiempo requerido mediante un algoritmo independientemente de su representación y del procesador que lo ejecute.

Por otra parte, si la cantidad de datos del problema por resolver es pequeña, prácticamente cualquier algoritmo que lo resuelva lo hará en un espacio corto de tiempo, siendo, en este caso, prácticamente indiferente el elegir uno u otro algoritmo para resolverlo.

Cuando aparece la dificultad sobre el tiempo requerido por un algoritmo es cuando el volumen de datos del problema por resolver aumenta.

Cuando el volumen de datos es suficientemente grande es cuando un algoritmo puede realmente aventajar a otro para resolver el mismo problema.

Veamos un ejemplo:

Ejemplo 2

Dada una lista $\{a_1, a_2, \cdots, a_n\}$ de números ordenados de menor a mayor, verifique si hay algún número repetido.

Los siguientes algoritmos resuelven el problema:

ALGORITMO 3

Paso 1 $i = 1, j = 2$
Paso 2 Si $a_i = a_j$ entonces la respuesta es SI. FIN
Paso 3 Si $j < n$ entonces $i = i + 1$; $j = j + 1$ y volver al paso 2.
Paso 4 La respuesta es NO. FIN

ALGORITMO 4

Paso 1 $i = 1$; $j = 2$
Paso 2 Si $a_i = a_j$ entonces la respuesta es SI. FIN
Paso 3 Si $j < n$ entonces $j = j + 1$ y volver al paso 2
Paso 4 Si $i < n-1$ entonces $i = i + 1$; $j = i + 1$ y volver al paso 2.
Paso 5 La respuesta es NO. FIN.

Si contamos solo las comparaciones que se efectúan en el paso 2 de ambos algoritmos, observamos que el primero compara cada número de la lista con el inmediatamente posterior, por lo que efectúa $n-1$ comparaciones como máximo (el peor caso será cuando no haya repeticiones o

bien cuando estén repetidos los dos últimos números solamente); mientras que en el segundo algoritmo puede suceder el caso en que compare cada número con todos los que le siguen, por lo que el número de comparaciones puede llegar a ser:

$$(n-1)+(n-2)+(n-3)+\cdots+2+1=\frac{n^2}{2}-\frac{n}{2}$$

Teniendo en cuenta la tabla siguiente en la que se observan los valores que toman n y $n^2/2 - n/2$ para distintos valores de n se deduce inmediatamente que el algoritmo 3 es claramente más ventajoso que el 4 especialmente cuando el valor n es grande.

Tabla 1. Valores de crecimiento de n

n	n-1	$n^2/2 - n/2$
2	1	1
10	9	45
100	99	4950
1000	999	499500
100000	99999	49995000
1000000	999999	4999950000

Ejemplo 3 *El método de reducción de Gauss*

Para resolver sistemas de ecuaciones lineales cuadrados, es decir: con el mismo número de ecuaciones que de incógnitas, podemos usar este método.

Sea el sistema:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{1}$$

Este sistema está completamente determinado por su matriz de coeficientes $A = (a_{ij})$ y el vector columna b con i-ésimo elemento b_i. La matriz aumentada se denota $(A|b)$ y se escribe:

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ a_{n1} & an_2 & \cdots & a_{nn} & b_n \end{array}\right] \tag{2}$$

Y el vector columna

$$x = \begin{bmatrix} x_1 \\ \cdots \\ x_n \end{bmatrix}$$

Es fácil ver que las soluciones del sistema (1) son las mismas que las de cualquier sistema obtenido del mismo por los siguientes procedimientos.

1. Intercambio de dos ecuaciones.
2. Multiplicación de una ecuación del sistema (1) por una constante $\neq 0$.
3. Sustitución de una ecuación por la suma de sí misma con un múltiplo de una ecuación diferente del sistema.

Aplicados al sistema (1), estos procedimientos corresponden a las **operaciones elementales en las filas** que se aplican a toda la matriz aumentada (2).

Para resolver el sistema (1) mediante el método de Gauss, tratamos de usar operaciones elementales en las filas para reducir la matriz (2) a una matriz de la forma:

$$\left[\begin{array}{cccc|c} u_{11} & u_{12} & u_{1n} & & c_1 \\ & u_{22} & u_{2n} & & c_2 \\ & & \cdots & \cdots & \cdots \\ & & & u_{nn} & c_n \end{array}\right] \tag{3}$$

Donde la parte cuadrada U a la izquierda de la partición tiene entradas igual a cero debajo de la diagonal principal. La matriz de coeficientes *A* original del sistema (1) se transforma en una ***matriz triangular superior U*** con ceros debajo de la diagonal principal. El sistema se convierte en:

$$Ux = c$$

Si una matriz *B* se puede obtener de una matriz *A* a través de operaciones elementales en las filas, entonces *B* es equivalente por filas a *A*. Por lo cual las matrices *A* y *U* descritas antes son equivalentes por filas.

ALGORITMO 5

Reducción de Gauss a la forma triangular superior

Paso 1 Si la entrada superior de la columna 1 es cero, entonces efectuar una operación de intercambio de filas para obtener un elemento distinto de cero en la parte superior de la columna. Siempre es posible esto en el caso de solución única. Llamamos pivote al elemento elegido distinto de cero.

Paso 2 Efectuar operaciones elementales en las filas de tal forma que las filas inferiores resultantes tengan cero como primera entrada.

Paso 3 Después del paso 2, la matriz tiene la forma:

$$\left[\begin{array}{cccc|c} u_{11} & u_{12} & \cdots & u_{1n} & c_1 \\ \hline 0 & x & \cdots & x & x \\ 0 & \cdots & \cdots & \cdots & \cdots \\ 0 & x & \cdots & x & x \end{array}\right] \tag{4}$$

Y la primera columna está en la forma que queremos. Se elimina (mentalmente) la fila superior y la primera columna de la matriz (4), dejando la parte enmarcada de esta. Se vuelve al paso 1 con esta matriz más pequeña y se repite el procedimiento para componer la siguiente columna. Se continúa hasta obtener la forma triangular superior.

Resuelva el sistema lineal a través del método de Gauss con sustitución regresiva:

$$\begin{aligned} x_2 - 3x_3 &= -5 \\ 2x_1 + 3x_2 - x_3 &= 7 \\ 4x_1 + 5x_2 - 2x_3 &= 10 \end{aligned}$$

Solución

Se reduce la matriz aumentada por medio de operaciones elementales en las filas. Los pivotes se encierran en un círculo.

Se intercambian las filas 1 y 2 (una operación)

$$\left[\begin{array}{ccc|c} 0 & 1 & -3 & -5 \\ 2 & 3 & -1 & 7 \\ 4 & 5 & -2 & 10 \end{array}\right] \approx \left[\begin{array}{ccc|c} 2 & 3 & -1 & 7 \\ 0 & 1 & -3 & -5 \\ 4 & 5 & -2 & 10 \end{array}\right]$$

Se suma la fila 1 multiplicada por -2 a la fila 3 (segunda operación)

$$\left[\begin{array}{ccc|c} 2 & 3 & -1 & 7 \\ 0 & 1 & -3 & -5 \\ 4 & 5 & -2 & 10 \end{array}\right] \approx \left[\begin{array}{ccc|c} 2 & 3 & -1 & 7 \\ 0 & 1 & -3 & -5 \\ 0 & -1 & 0 & -4 \end{array}\right]$$

Se suma la fila 2 a la 3: (tercera operación)

$$\left[\begin{array}{ccc|c} 2 & 3 & -1 & 7 \\ 0 & 1 & -3 & -5 \\ 0 & -1 & 0 & -4 \end{array}\right] \approx \left[\begin{array}{ccc|c} 2 & 3 & -1 & 7 \\ 0 & 1 & -3 & -5 \\ 0 & 0 & -3 & -9 \end{array}\right]$$

La suma de estas operaciones, considerando a las operaciones elementales como las que se enumeran, es en general

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}$$

De la última matriz obtenemos, por sustitución regresiva:

$$x_3 = 3 \quad x_2 = 4 \quad x_1 = -1$$

Para la sustitución regresiva se suman las operaciones y se obtiene la siguiente serie:

$$1 + 2 + 3 + \cdots + n = \frac{n^2}{2} + \frac{n}{2}$$

3 Peor caso y caso probabilístico

Para muchos algoritmos, el número de operaciones realizadas depende en gran medida de los datos involucrados y puede variar ampliamente de un caso a otro. Por ejemplo, en el juego «Adivina el número», siempre es posible «tener suerte» y seleccionar el número correcto en el primer intento. Por otro lado, este apenas es útil para estimar el comportamiento general del algoritmo. Generalmente, estamos más interesados en estimar el comportamiento en (1) el caso promedio, que proporciona una idea del comportamiento típico del algoritmo, y (2) el peor caso posible, que proporciona un límite superior en el tiempo requerido.

En el caso del algoritmo de búsqueda lineal, cada una de estas medidas es relativamente fácil de analizar. En el peor de los casos, adivinar un número en el rango de 1 a *N* puede requerir *N* conjeturas completas. En el ejemplo específico que implica el rango de 1 a 100, esto ocurre si el número fuera exactamente 100. Para calcular el caso promedio, debemos sumar el número de conjeturas requeridas para cada posibilidad y

dividir ese total entre N. El número 1 se encuentra en el primer intento, 2 requiere dos conjeturas, y así sucesivamente, hasta N, que requiere N conjeturas, la suma de estas posibilidades es entonces:

$$1+2+3+\cdots+N=\frac{N^2}{2}+\frac{N}{2}$$

Dividiendo entre N se obtiene el número promedio de conjeturas que es:

$$\frac{N+1}{2}$$

El caso de la búsqueda binaria requiere un poco más de reflexión, pero sigue siendo razonablemente simple. En general, cada conjetura que hacemos nos permite reducir el tamaño del problema por un factor de dos. Por ejemplo, si acertamos 50 en el ejemplo de 1 al 100 y descubrimos que nuestra conjetura es baja, podemos eliminar inmediatamente valores en el rango de 1 a 50 para mayor consideración. Por lo tanto, la primera conjetura reduce el número de posibilidades a $N/2$, el segundo a $N/4$, y así sucesivamente. Aunque en algunos casos podríamos tener suerte y adivinar el valor exacto en algún momento en el proceso. El peor de los casos requerirá continuar este proceso hasta que solo quede una posibilidad. El número de pasos requeridos para lograr esto es como sigue:

$$\underbrace{\frac{N}{2}\cdots\frac{N}{2}}_{k\text{ veces}}=1$$

donde k indica el número de conjeturas requeridas. Simplificando esto se tiene:

$$\frac{N}{2^k} = 1$$
$$N = 2^k$$

Como queremos una expresión para k en términos del valor de N, debemos usar la definición de logaritmo para cambiar esto.

$$k = \log_2 N$$

Por lo tanto, en el peor de los casos el número de pasos necesarios para adivinar un número usando la búsqueda binaria es el logaritmo en base 2 del número de valores. En el caso promedio podemos esperar encontrar el valor correcto usando una suposición menos.

Las estimaciones de la complejidad computacional se utilizan con mayor frecuencia para proporcionar información sobre el comportamiento de un algoritmo a medida que crece el tamaño del problema, como ya lo vimos en el caso del algoritmo 4. Aquí, por ejemplo, podemos usar la fórmula del peor de los casos para crear una tabla que muestre el número de conjeturas requeridas para los algoritmos de búsqueda lineal y binaria, respectivamente:

Tabla 2. Comparación de búsqueda lineal y búsqueda binaria para *N*

N	Búsqueda lineal	Búsqueda binaria
10	10	4
100	100	7
1,000	1,000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20

Esta tabla demuestra de manera concluyente el valor de la búsqueda binaria. La diferencia entre los algoritmos se vuelve cada vez más pronunciada a medida que *N* adquiere mayores valores. Para diez valores, la búsqueda binaria arrojará el resultado en no más de cuatro conjeturas. Dado que la búsqueda lineal requiere diez, el método de búsqueda binaria representa un factor de 2.5 aumento de la eficiencia. Para 1,000,000 de valores, por otro lado, este factor ha aumentado a uno de 50,000. Lo que representa una enorme mejora.

En el estudio anterior del tiempo de ejecución de los algoritmos, hemos analizado el número de comparaciones que podrían llegar a realizarse en algún caso extremo. A estos casos en que el tiempo es el mayor posible de entre todos los casos que se pueden presentar se les llama **el peor caso.**

En ocasiones el peor caso se presenta a menudo al resolver un problema y en otras se presenta con poca frecuencia. Por esta razón tiene interés el estudio del tiempo de ejecución de un algoritmo en el caso medio o caso probabilístico. Este estudio entra dentro del campo del cálculo de probabilidades y de la estadística y, siendo de gran interés en la evaluación de los algoritmos, resulta en ocasiones sumamente complejo.

En adelante el análisis que hagamos de los algoritmos será siempre estudiando el comportamiento de este en el peor caso, que es también conocido como caso promedio.

A pesar de la ventaja que esto supone, en ocasiones un algoritmo cuyo comportamiento en el peor caso es desastroso puede ser útil en una gran cantidad de casos si se presenta el peor caso con una frecuencia muy pequeña.

El análisis de algoritmos se encarga del estudio del **tiempo y espacio** requerido por un algoritmo para su ejecución. Ambos parámetros, como hemos visto, pueden ser estudiados respecto del **peor caso** (o caso general) o respecto del caso probabilístico (o caso esperado). En la práctica, casi siempre es más difícil determinar el tiempo de ejecución esperado que el del peor caso, pues el análisis se hace intratable en matemáticas. Así pues, se utilizará el tiempo de ejecución del peor caso como medida principal de la complejidad del tiempo, aunque se mencionará la complejidad del caso promedio cuando pueda hacerse en forma significativa.

Tanto el tiempo como el espacio de un algoritmo son parámetros que, en general, dependen del tamaño n de los datos de entrada del algoritmo. En consecuencia, son funciones $T(n)$ y $E(n)$ enteras de variable entera.

En general, no resulta sencillo determinar el valor exacto de la función tiempo $T(n)$ de un algoritmo. Para poder hacerlo es preciso conocer con exactitud cuáles y cuántas veces se realizan las operaciones básicas cuya ejecución requiere un tiempo constante conocido.

En definitiva, lo que aparece al analizar un algoritmo es un problema combinatorio. No obstante, como hemos dicho, no siempre resulta fácil hacer el cálculo exacto del número de operaciones requeridas por un algoritmo. Por esta razón es por la que en muchas ocasiones el análisis de algoritmos se reduce a estudiar el **comportamiento** de la función tiempo $T(n)$ y no cuál es su valor exacto.

4 Análisis asintótico de funciones

1
2
3
4
5
6
7
8
9
10
11
12
13

En esta sección el objeto de estudio serán las funciones reales de variable natural $f : N \to R$. La idea fundamental consiste en comparar funciones de este tipo para poder decir cuál tiene mejor comportamiento asintótico, es decir, cuál es menor cuando la variable independiente es suficientemente grande.

Si sabemos hacer esto con dos funciones podremos utilizar las funciones de tiempo de dos algoritmos y así determinar cuál de ellos tiene mejor comportamiento asintótico.

El problema que en ocasiones resulta complicado es el de hallar explícitamente la función tiempo de un algoritmo. El análisis asintótico que haremos permitirá, en ocasiones, conocer cómo se comporta una función aun sin conocerla.

Definición 1

Si f y g son dos funciones de N en R, se dice que g domina asintóticamente a f (o simplemente que g domina a f) si existen enteros $k \geq 0$ y $m \geq 0$ tales que se verifica la desigualdad: $|f(n)| \leq k\,|g(n)|$ para todo entero $n \geq m$.

Observe que si g domina a f y $g(n) \neq 0$, entonces $|f(n)/g(n)| \leq k$ para casi todos los enteros n (es decir, para todos salvo una cantidad finita). En concreto, para todos los valores n $\geq m$ se verifica dicha desigualdad.

En esta situación, si f y g son funciones de tiempo de dos algoritmos F y G respectivamente, resulta que el algoritmo F nunca tardará más de k veces el tiempo que tarda el algoritmo G en resolver un problema del mismo tamaño.

Ejemplo 4

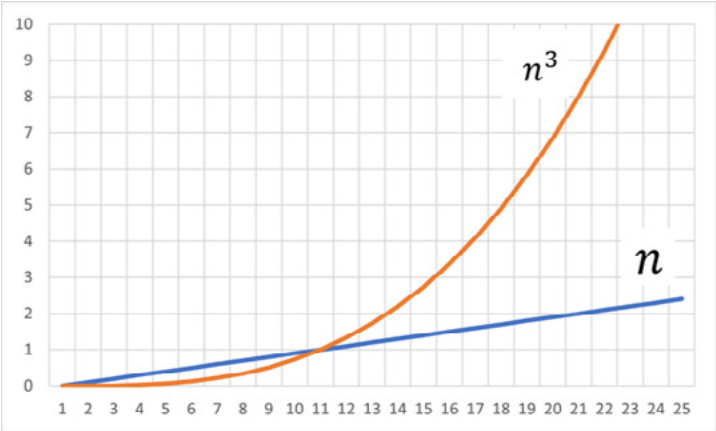
Si $f(n) = n$ y $g(n) = n^3$, se verifica que g domina a f: En efecto, si $m = 0$ y $k = 1$ se verifica que para todo $n \geq m$ se cumple la desigualdad

$$|n| \leq k|n^3|$$

En este caso, $|n| \leq |n^3|$ ya que $k = 1$ pues el cubo de un número natural es siempre mayor o igual que dicho número.

Y se comprueba gráficamente:

Figura 1. Comportamiento de $f(n)=n$ y $g(n)=n^3$ con $k=1$



La definición de dominación establece una relación binaria en el conjunto de las funciones de *N* en *R*. El siguiente teorema da propiedades de dicha relación.

TEOREMA 1

La relación de dominación $<$ definida por $f < g \leftrightarrow g$ domina a f, es una relación reflexiva y transitiva. La demostración de este teorema queda de ejercicio, asimismo proponemos como ejercicio el comprobar que dicha relación no es simétrica, es decir, si *g* domina a *f*, no se verifica necesariamente que *f* domina a *g*. En consecuencia, la relación de dominación no es una relación de equivalencia.

Por otra parte, esta relación tampoco es relación de orden pues no verifica la propiedad antisimétrica (compruébelo también como ejercicio).

Si *f* es una función de *N* en *R* denotaremos por $O(f)$ al conjunto de todas las funciones dominadas por *f*. Con notación matemática:

$$O(f)=\{g \mid g: N \to R \text{ es una aplicación y } g<f\}$$

Si una función g pertenece al conjunto $O(f)$ se suele decir que g es una o mayúscula de f o que ***g es de orden f***.

Por ejemplo, decir que el tiempo de ejecución de un programa $T(n)$ es $O(n^2)$, significa que existen constantes enteras positivas m y k tales que para $n \geq m$ se tiene $T(n) \leq kn^2$

Ejemplo 5

Suponga que $T(0)=1$, $T(1)=4$ y en general $T(n)=(n+1)^2$. Entonces se observa que $T(n)$ es $O(n^2)$ cuando $m=1$ y $k=4$, es decir para $n \geq 1$, se tiene que $(n+1)^2 \leq 4n^2$ que es fácil de demostrar. Observe que no se puede hacer $m=0$ pues $T(0)=1$ no es menor que $k0^2=0$ para ninguna constante k.

Se dice que $T(n)$ es $O(f(n))$ si existen m y k tales que $T(n) \leq k\, f(n)$ cuando $n \geq m$. Cuando el tiempo de ejecución de un programa es $O(f(n))$ se dice que tiene velocidad de crecimiento $f(n)$.

TEOREMA 2

Se verifican las siguientes propiedades de los conjuntos $O(f)$:

1. $f \in O(f)$
2. Si $f \in O(g)$ entonces $cf \in O(g)$ para todo $c \in R^+$
3. Si $f \in O(g)$ y $h \in O(g)$ entonces $f+g \in O(g)$
4. $f \in O(g)$ si y sólo si $O(f) \subset O(g)$

5. Si $f \in O(g)$ y $g \in O(f)$ entonces $O(f) = O(g)$
6. Si $f \in O(g)$ y $g \in O(h)$ entonces $f \in O(h)$

Algunos conjuntos $O(f)$ que aparecen con frecuencia al analizar algoritmos son:

$$O(1),\ O(\log n),\ O(n),\ O(n \log n),\ O(n^2),\ \cdots,\ O(n^p),\cdots,O(c^n),\ O(n!)$$

Si la función tiempo de un algoritmo es de orden 1 se dice que dicho algoritmo tiene complejidad constante, si es de orden $\log n$ se dice que es **logarítmica**, si es de orden n se dice que es **lineal**, si es de orden n^p siendo p un número natural, se dice que es **polinómica**, si es de orden c^n con $c > 1$ se dice que es **exponencial** y si es de orden $n!$ se dice que es **factorial**.

Observación 1

Mientras no se diga lo contrario, se entenderá que los logaritmos que aparecen son en base 2.

Ejemplo 6

La función $T(n) = 3n^3 + 2n^2$ es $O(n^3)$. Para comprobar esto, sean $m = 0$ y $k = 5$ entonces para $n \geq 0$, $3n^3 + 2n^2 \leq 5n^3$. También se podría decir que $T(n)$ es $O(n^4)$ pero sería una afirmación más débil que decir que $T(n)$ es $O(n^3)$

Ejemplo 7

La función 3^n no es $O(2^n)$

Demostración:

Suponga que existen m y k tales que para toda $n \geq m$, se tiene $3^n \leq k2^n$, entonces $k \geq (3/2)^n$ para cualquier $n \geq m$, pero $(3/2)^n$ se hace arbitrariamente grande conforme n crece y por lo tanto, ninguna constante k puede ser mayor que $(3/2)^n$ para toda n.

Cuando se dice que $T(n)$ es $O(f(n))$ se sabe que $f(n)$ es una cota superior para la velocidad de crecimiento de $T(n)$. Para especificar una cota inferior para la velocidad de crecimiento de $T(n)$ se usa la notación $W(g(n))$ que se lee «$T(n)$ es omega de $g(n)$» lo cual significa que existe una constante c tal que $T(n) \geq cg(n)$ para un número infinito de valores de n.

Ejemplo 8

Para verificar que la función $T(n) = n^3 + 2n^2$ es $\Omega(n^3)$, sea $c = 1$, entonces $T(n) \geq cn^3$ para $n = 0, 1$.

TEOREMA 3

Dadas las funciones de tiempo, se verifican las siguientes contenciones:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(c^n) \subset O(n!)$$

siendo $c > 1$.

Además, dichas contenciones son propias, en ninguna de ellas se da la igualdad.

Demostración:

a. $O(1) \subset O(\log n)$

En efecto si $m = 2$ y $k = 1$ se verifica que para todo $n > 2$ es $1 \leq \log n$ por lo que la función constante 1 pertenece a $O(\log n)$, por lo tanto, por el apartado 4 del teorema 2, $O(1) \subset O(\log n)$.

Además, la contención es propia: si $O(\log n)$ estuviera contenido en $O(1)$ se verificaría que $\log n$ pertenecería a $O(1)$ y, por lo tanto, $\log n$ estaría dominada por la función constante 1. Esto significa que existen $m \geq 0$ y $k \geq 0$ tales que para todo $n > m$ sería

$$|\log n| \leq k\,|1|$$

es decir:

$$\log n \leq k$$

lo cual es una contradicción, pues como

$$\lim_{n \to \infty} \log n = \infty$$

la función $\log n$ no puede estar acotada por una constante k.

b. $O(\log n) \subset O(n)$

Considerando $m = 0$ y $k = 1$ se verifica que para todo $n > m$ se cumple:

$$|\log n| \leq k\,|n|$$

ya que para todo $n>0$ es $\log n<n$. Por lo tanto, $\log n \subset O(n)$ y por la misma razón de antes $O(\log n)\subset O(n)$.

c. $O(n) \subset O(n \log n)$

Si $n>2$ se verifica que $\log n>1$ y, por tanto, $n \log n>1(n)=n$; por lo que tomando $m=2$ y $k=1$ se verifica que para todo $n>m$ se tiene:

$$| n | \leq k | n \log n |$$

Luego $n \in O(n \log n)$ y por lo tanto $O(n) \subset O(n \log n)$.

d. $O(n \log n) \subset O(n^2)$

Como $\log n<n$ si $n>0$ resulta que $n \log n<n(n)=n^2$. Por lo tanto, considerando $m=0$ y $k=1$ se verifica que para todo $n>m$ es

$$| n \log n | < k | n^2 |$$

luego $n \log n \in O(n^2)$ y, por lo tanto, $O(n \log n)\subset O(n^2)$.

e. $O(n^2) \subset O(c^n)$ (si $c>1$)

Como en casos anteriores, es suficiente probar que para una n suficientemente grande se verifica la desigualdad

$$n^2 < c^n$$

lo que equivale, tomando logaritmos (tenga en cuenta que como $c>1$ el $\log c>0$), a la desigualdad:

$\log n^2 < \log c^n$

o bien

$$\frac{\log n}{n} < \frac{\log c}{2}$$

Como:

$$\lim_{n \to \infty} \frac{\log n}{n} = 0$$

y $(\log c)/2$ es una constante positiva, resulta que existe un valor m tal que para todo valor $n > m$ es

$$\frac{\log n}{n} < \frac{\log c}{2}$$

como queríamos probar.

f. $O(c^n) \subset O(n!)$

Sean $k = [c]$ y $m =$ máx. $\{[c^k], k\}$ (donde $[c]$ representa al menor entero mayor que c)

Entonces, si $n > m$ se verifica

$n! = n(n-1)(n-2)\cdots(k+1)\,k(k-1)\cdots 2\,1$

y como $k \geq [c]$ y $[c] \geq c$, se verifica

$(n-1)(n-2)\cdots(k+1)\,k \geq c^n - k$

Por otra parte, como $n > c^k$, resulta

$n! > c^k\, c^{n-k} = c^n$

En consecuencia

$n! > c^n$ si $n > m$ y por lo tanto $O(c^n) \subset O(n!)$ como antes.

TEOREMA 4

Si c es una constante, se verifica:

a. $\Sigma c \in O(n)$
b. $\Sigma i \in O(n^2)$
c. $\Sigma i^2 \in O(n^3)$

La demostración se deja como ejercicio.

5 Velocidad de crecimiento y cálculo de tiempo de ejecución de un algoritmo

Partiremos del supuesto de que es posible evaluar algoritmos comparando sus funciones de tiempo de ejecución sin considerar las constantes de proporcionalidad. Es decir, un algoritmo con tiempo de ejecución $O(n^2)$ es mejor que uno con tiempo de ejecución $O(n^3)$, sin embargo, además de los factores constantes debidos al compilador y la máquina, existe un factor constante debido a la naturaleza del algoritmo mismo. Es posible que, con una combinación determinada de compilador y máquina, el primer algoritmo tarde $100n^2$ milisegundos, mientras el segundo tarde $5n^3$ milisegundos. En este caso ¿no es preferible el segundo algoritmo al primero?

La respuesta a esto depende del tamaño de las entradas que se espera que procesen los algoritmos. Para entradas de tamaño $n < 20$, el programa con tiempo de ejecución $5n^3$ será más rápido que el de tiempo de ejecución $100n^2$. Así pues, si el algoritmo se va a ejecutar con entradas pequeñas, será preferible el algoritmo cuyo tiempo de ejecución es $O(n^3)$. No obstante, conforme n crece, la razón de los tiempos de ejecución que

es $5n^3/100n^2 = n/20$, se hace arbitrariamente grande. Así, a medida que crece el tamaño de la entrada, el algoritmo $O(n^3)$ requiere un tiempo significativamente mayor que $O(n^2)$. Pero si hay algunas entradas grandes en los problemas para cuya solución se están diseñando estos dos algoritmos, será mejor optar por el algoritmo cuyo tiempo de ejecución tiene la menor velocidad de crecimiento.

Ejemplo 9

En la figura 2 se muestran gráficamente los tiempos de ejecución de cuatro algoritmos de distintas complejidades de tiempo, medidas en segundos, para una combinación determinada de compilador y máquina.

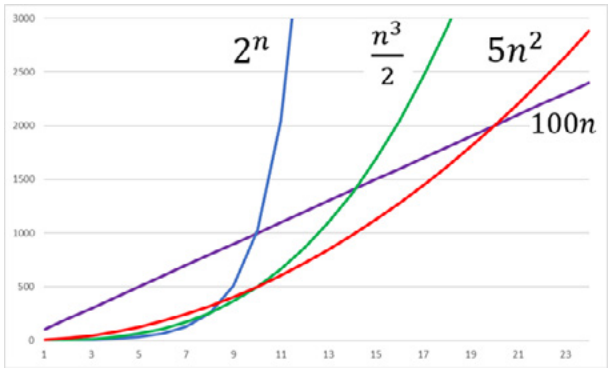


Figura 2. Tiempos de ejecución de diferentes algoritmos

Suponga que se dispone de 1000 segundos o alrededor de 17 minutos para resolver un problema determinado. ¿Qué tamaño de problema se puede resolver? Considerando estos tiempos, en la segunda columna de la siguiente tabla se muestra que los cuatro algoritmos pueden resolver problemas de un tamaño similar en 10^3 segundos. Si se adquiere una máquina que funciona diez veces más rápido sin costo adicional, entonces es posible dedicar 10^4 segundos a la solución de problemas que

antes requerían 10^3 segundos. El tamaño máximo de problema que es posible resolver ahora con los cuatro algoritmos se muestra en la tercera columna de la tabla, y la razón entre los valores de la segunda y tercera columnas se muestran en la cuarta.

Se observa que un aumento de 1000% en la velocidad de la computadora origina apenas un incremento del 30% en el tamaño del problema que se puede resolver con el programa $O(2^n)$. Los aumentos adicionales de un factor de diez en la rapidez de la computadora a partir de este punto originan aumentos porcentuales aun menores en el tamaño de los problemas. De hecho, el programa $O(2^n)$ solo puede resolver problemas pequeños, independientemente de la rapidez de la computadora.

Tabla 3. Comparación de tiempos de ejecución

Tiempo de ejecución $T(n)$	Tamaño máximo del problema para 10^3 segundos	Tamaño máximo del problema para 10^4 segundos	Incremento en el tamaño máximo del problema
$100\,n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

En la tercera columna de la tabla se puede apreciar una superioridad evidente del programa $O(n)$, este permite un aumento de 1000% en la rapidez de la computadora. Y se observa que los programas $O(n^3)$ y $O(n^2)$ permiten aumentos de 230% y 320% respectivamente en el tamaño del problema, para un incremento del 1000% en la rapidez de la computadora. Estas razones se mantendrán vigentes para incrementos adicionales en la rapidez de la computadora.

A medida que las computadoras aumenten su rapidez y disminuyan su precio, también el deseo de resolver problemas más grandes y complejos seguirá creciendo. Así la importancia del descubrimiento y el empleo de algoritmos eficientes (cuyas velocidades de crecimiento sean pequeñas) irá en aumento.

Observaciones 2

a. La velocidad de crecimiento del tiempo de ejecución del peor caso no es el único criterio, ni necesariamente el más importante para evaluar un algoritmo.
b. De acuerdo con el punto anterior, si un programa solo se va a usar algunas veces, el costo de su escritura y depuración es el dominante, de manera que el tiempo de ejecución raramente influirá en el costo total. En tal caso debe elegirse el algoritmo que sea más fácil de aplicar correctamente.
c. Un algoritmo eficiente pero complicado puede no ser apropiado porque posteriormente puede suceder que tenga que darle mantenimiento otra persona distinta del escritor. Se espera que, al difundir el conocimiento de las principales técnicas de diseño de algoritmos eficientes, se podrán utilizar libremente algoritmos más complejos, pero debe considerarse la posibilidad de que un programa resulte inútil debido a que nadie entiende sus sutiles y eficientes algoritmos.
d. Existen ejemplos de algoritmos eficientes que ocupan mucho espacio para poder aplicarlos sin un almacenamiento secundario lento, lo cual puede anular la eficiencia.
e. En los algoritmos numéricos, la precisión y la estabilidad son tan importantes como la eficiencia.

6 Tiempo de ejecución de un algoritmo: reglas de operación

Comenzaremos con algunas reglas básicas de suma y producto en notación asintótica:

Sean $T_1(n)$ y $T_2(n)$ los tiempos de ejecución de dos fragmentos P_1 y P_2 de un algoritmo y que $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$. Entonces

$$T_1(n) + T_2(n)$$

El tiempo de ejecución de P_1 seguido de P_2 es $O(\text{máx.}[f(n), g(n)])$. Esto se puede verificar observando que, para algunas constantes, c_1, c_2, n_1, n_2 si $n \geq n_1$, entonces

$$T_1(n) \leq c_1 f(n)$$

y si $n \geq n_2$ entonces

$$T_2(n) \leq c_2 g(n)$$

Sea $m = \text{máx.}(n_1, n_2)$, si $n \geq m$, entonces:

$$T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$$

De aquí se concluye si $n \geq m$ entonces

$$T_1(n) + T_2(n) \leq (c_1 + c_2) \max. [f(n), g(n)]$$

Por lo tanto: $T_1(n) + T_2(n)$ es $O(\text{máx.}[f(n), g(n)])$.

Ejemplo 10

La regla de la suma anterior se puede usar para calcular el tiempo de ejecución de una secuencia de pasos de programa, donde cada paso puede ser fragmento de un programa arbitrario con ciclos y ramificaciones. Suponga que se tienen tres pasos cuyos tiempos de ejecución son respectivamente, $O(n^2)$, $O(n^3)$ y $O(n \log n)$. Entonces el tiempo de ejecución de los dos primeros pasos ejecutados en secuencia es $O(\text{máx.}(n^2, n^3))$ que es $O(n^3)$, el tiempo de ejecución de los tres juntos es $O(\text{máx.}(n^3, n \log n))$ que es $O(n^3)$.

En general el tiempo de ejecución de una secuencia fija de pasos, dentro de un factor constante, es igual al tiempo de ejecución del paso con mayor tiempo de ejecución. En raras ocasiones dos pasos pueden tener tiempos de ejecución inconmensurables (ninguno es mayor que el otro, ni son iguales). Por ejemplo, puede haber pasos con tiempo de ejecución $O(f(n))$ y $O(g(n))$, donde:

$$f(n) = \begin{cases} n^4 \text{ si } n \text{ es par} \\ n^2 \text{ si } n \text{ es par} \end{cases} \qquad g(n) = \begin{cases} n^2 \text{ si } n \text{ es par} \\ n^3 \text{ si } n \text{ es par} \end{cases}$$

En tales casos, la regla de la suma debe aplicarse directamente, en el ejemplo, el tiempo de ejecución es $O(\text{máx.}[f(n), g(n)])$, esto es n^4 si n es par y n^3 si n es impar.

Otra observación útil sobre la regla de la suma es que si $g(n) \leq f(n)$ para toda $n \geq m$ entonces $O(f(n) + g(n))$ es lo mismo que $O(f(n))$. Por ejemplo, $O(n^2 + n)$ es lo mismo que $O(n^2)$. Lo mismo sucede con algoritmos como el de eliminación gaussiana que es de orden $O(n^3)$.

La regla del producto es la siguiente: Si $T_1(n)$ y $T_2(n)$ son $O(f(n))$ y $O(g(n))$, respectivamente, entonces $T_1(n)T_2(n)$ es $O(f(n)g(n))$. Su demostración se deja como ejercicio. Según la regla del producto, $O(cf(n))$ significa lo mismo que $O(f(n))$ si c es una constante positiva cualquiera. Por ejemplo, $O(n^2/2)$ es lo mismo que $O(n^2)$.

Una vez que se han estudiado las herramientas necesarias para analizar el comportamiento de la función tiempo de un algoritmo, se hará el análisis de algunos algoritmos.

Ejemplo 11

Encuentre el máximo de un conjunto de n números naturales $\{a_1, a_2, \cdots, a_n\}$

ALGORITMO 6

Algoritmo máximo secuencial

Entrada:	La lista $L = \{a_1, \cdots, a_n\}$
Paso 1	$m = a_1 \quad i = 2$
Paso 2	Si $m < a_i$ entonces $m = a_i$
Paso 3	$i = i + 1$

Paso 4	Si $i > n$ FIN; en caso contrario volver al paso 2
Salida:	el máximo es m

Análisis del algoritmo:

En el paso 1 se realizan dos operaciones, en el paso 2, en el peor caso, otras dos, en el paso 3 una y en el paso 4, en el peor caso 2. Por lo tanto, cada vez que del paso 4 volvemos al paso 2 serán necesarias 5 operaciones; como tras aplicar el paso 1, el ciclo 2-4 se ejecuta $n-1$ veces (hasta que i toma el valor $n+1$), el número total de operaciones que requiere este algoritmo en el peor caso será:

$$T(n) = 2 + 5(n-1)$$

y el comportamiento asintótico queda dado por la expresión:

$$T(n) \in O(n)$$

Comparamos este algoritmo con el siguiente algoritmo recursivo

ALGORITMO 7

Algoritmo máximo binario

Entrada:	La lista $L = \{a_1, \cdots, a_{i'}, a_i + 1, \cdots, a_j\}$ con n elementos, $i = 1$
Paso 0	Si $i = j$ entonces $m = a_1$ e ir al paso 5
Paso 1	$k = [i + j/2]$ (donde [ ] indica parte entera de $i + j/2$)
Paso 2	$L_1 = \{a_{i'}, \cdots, a_k\}$, $L_2 = \{a_k + 1, \cdots, a_j\}$
Paso 3	$m_1 =$ MAXIMO BINARIO (L_1), $m_2 =$ MAXIMO BINARIO (L_2)
Paso 4	Si $m_1 > m_2$ entonces $m = m_1$. En caso contrario $m = m_2$
Paso 5	FIN
Salida:	el máximo es m

Análisis del algoritmo:

Si llamamos T(n) al tiempo que requiere el algoritmo para hallar el máximo de una lista de *n* números, se verifica que el tiempo requerido para ejecutar el paso 3 será 2 T(n/2)(cuando *n* sea potencia de 2). Como el número de operaciones que requieren los pasos 0, 1, 2, 4 y 5 es constante se verificará que

$$T(n) \leq 2T(n/2) + c$$

donde *c* es una constante. Como $T(1)$ es constante, podemos suponer que $T(1) \leq c$ por lo que se verifica que $T(n) \in O(n)$.

De esto último se concluye que el comportamiento asintótico de ambos algoritmos es el mismo. Observe que en el análisis asintótico de este segundo algoritmo no fue necesario obtener la función de tiempo de este.

En varias áreas de optimización se presentan problemas de tipo combinatorio como se verá en la siguiente sección en donde la complejidad computacional de estos problemas tiene una clasificación especial, de hecho, hay un famoso problema abierto sobre ellos.

Tabla 4. Características comunes de complejidad

Complejidad del algoritmo	Tiempo de ejecución cuando *n* se dobla	Nombre convencional
$O(1)$	No cambia	Constante
$O(\log n)$	Se incrementa por una pequeña constante	Logarítmica
$O(n)$	Se dobla	Lineal
$O(n \log n)$	Un poco más que el doble	$n \log n$
$O(n^2)$	Se incrementa por un factor de 4	Cuadrática
$O(n^k)$	Se incrementa por un factor de 2^k	Polinomial
$O(\alpha^n)$, $\alpha>1$	Depende de α, pero crece muy rápido	Exponencial

7 Representación de los algoritmos[8]

Los algoritmos se representan básicamente de dos maneras: con diagramas de flujo o con pseudocódigo.

Diagramas de flujo

Los diagramas de flujo son la representación gráfica de los algoritmos. Elaborarlos implica diseñar un diagrama de bloque que contenga un bosquejo general del algoritmo, y con base en este proceder a su ejecución con todos los detalles necesarios. Las reglas para construir diagramas de flujo se dan a continuación:

8 https://programas.cuaed.unam.mx/repositorio/moodle/pluginfile.php/1196/mod_resource/content/1/contenido/index.html#contenido

1. Debe diagramarse de arriba hacia abajo y de izquierda a derecha. Es una buena costumbre en la diagramación que el conjunto de gráficos tenga un orden.
2. El diagrama solo tendrá un punto de inicio y uno final. Aunque en el flujo lógico se tomen varios caminos, siempre debe existir una sola salida.
3. Usar notaciones sencillas dentro de los gráficos, y si se requieren notas adicionales, colocarlas en el gráfico de anotaciones a su lado.
4. Se deben inicializar todas las variables al principio del diagrama. Esto es muy recomendable, ya que ayuda a recordar todas las variables, constantes y arreglos que van a ser utilizados en la ejecución del programa; además, nunca sabemos cuándo otra persona modificará el diagrama y necesitará saber de estos datos.
5. Procurar no cargar demasiado una página con gráficos; si es necesario, utilizar más hojas, emplear conectores. Cuando los algoritmos son muy grandes se pueden utilizar varias hojas para su graficación, con conectores de hoja para cada punto en donde se bifurque a otra hoja.
6. Todos los gráficos estarán conectados con flechas de flujo. Jamás debe dejarse un gráfico sin que tenga alguna salida, a excepción del que marque el final del diagrama.

Terminado el diagrama de flujo se realiza la prueba de escritorio; es decir, se le da un seguimiento manual al algoritmo, llevando el control de variables y resultados de impresión de forma tabular.

Ventajas:

1. Programas bien documentados.
2. Cada gráfico se codificará como una instrucción de un programa, realizando una conversión sencilla y eficaz.

3. Facilita la depuración lógica de errores.
4. Se simplifica su análisis al facilitar la comprensión de las interrelaciones.

Desventajas:

1. Su elaboración demanda varias pruebas en borrador.
2. Los programas muy grandes requieren diagramas laboriosos y complejos.
3. Falta de normatividad en su elaboración, lo que complica su desarrollo.

Pseudocódigo

Representa de forma descriptiva las operaciones que debe realizar un algoritmo.

```
function fact(n: integer): integer;
        {fact(n) calcula n!}
        begin
                if n<=1 then
                  fact := 1
                else
                  fact := n + fact(n-1)
        end {fact}
```

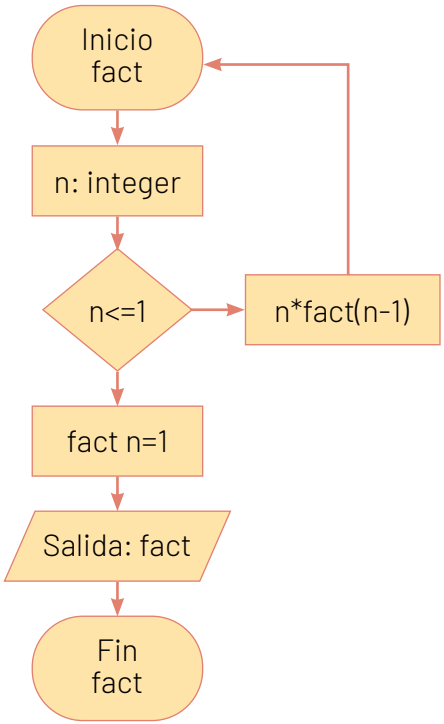


Figura 3. Pseudocódigo y diagrama de flujo

8 Complejidad de los problemas

Algunas definiciones importantes sobre la complejidad de los problemas son las siguientes:

Definición 2[9]

Un problema es un conjunto de instancias al cual corresponde un conjunto de soluciones a través de una relación que asocia para cada instancia del problema un subconjunto de soluciones (posiblemente vacío).

9 https://www.u-cursos.cl/ingenieria/2008/2/IN34A/3/material_docente/bajar?id=185136

Definición 3

Una instancia de un problema se obtiene cuando se especifican valores particulares para todos los parámetros del problema. Por ejemplo, consideremos el problema de la prueba de primalidad. La instancia es un número (por ejemplo 15) y la solución es «sí» si el número es primo, y «no» en caso contrario. Visto de otra manera, la instancia es una entrada particular del problema, y la solución es la salida correspondiente para la entrada dada.

Consideraremos dos tipos de problemas, los problemas de decisión y los problemas de optimización.

Definición 4

Problema computacional[10]

Un problema computacional constituye una pregunta que debe ser respondida, teniendo generalmente varios parámetros o variables libres, cuyos valores no se han especificado. Un problema se describe mediante: Una descripción general de todos sus parámetros (pueden ser de entrada o de salida).

Una sentencia que describa las propiedades que la respuesta, o la solución, debe cumplir.

10 https://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_complejidad_computacional

Definición 5

Problema de optimización

Un problema de optimización tiene como objetivo encontrar una solución que sea óptima entre todas las soluciones factibles que se tengan, estas soluciones factibles se pueden considerar como decisiones factibles, y están sujetas a una serie de restricciones que deben cumplir.

Definición 6

Problema de decisión

Un problema de decisión es un tipo especial de problema computacional cuya respuesta es solamente «sí» o «no» (o, de manera más formal, «1» o «0»).

Un problema de decisión pudiera verse como un lenguaje formal, donde los elementos que pertenecen al lenguaje son las instancias del problema cuya respuesta es «sí», los que no pertenecen al lenguaje son aquellas instancias cuya respuesta es «no». El objetivo es decidir, con la ayuda de un algoritmo, si una determinada entrada es un elemento del lenguaje formal considerado. Si el algoritmo devuelve como respuesta «sí», se dice que el algoritmo acepta la entrada, de lo contrario se dice que la rechaza.

Los problemas de decisión constituyen uno de los principales objetos de estudio de la teoría de la complejidad computacional, pues la NP-completitud se aplica directamente a este tipo de problemas en vez de a problemas de optimización. Estos problemas tienen gran importancia porque casi todo problema puede transformarse en un problema de decisión.

En general, un problema de decisión puede reescribirse como un problema de optimización y viceversa, algunos ejemplos son los siguientes.

Ejemplo 12

Consideremos el problema del agente viajero.

Problema de optimización: dados el conjunto de ciudades, el conjunto de caminos entre ciudades y los costos de utilizar cada camino, se desea determinar un viaje de costo mínimo.

Problema de decisión: dados el conjunto de ciudades, el conjunto de caminos entre ciudades, los costos de utilizar cada camino y un entero no negativo k, ¿existe un viaje de costo menor o igual que k?

Ejemplo 13

Consideremos el problema de programación de trabajos. Se tiene un conjunto de n trabajos que deben ser procesados por m máquinas en forma secuencial de modo tal que el tiempo de procesamiento sea mínimo. Es decir, cada trabajo debe ser procesado primero por la máquina 1, luego por la 2 y así sucesivamente hasta m.

Problema de optimización: dados los tiempos de procesamiento de cada uno de los n trabajos en cada una de las *m* máquinas, determinar una secuencia para procesarlos que minimice el tiempo total de procesamiento.

Problema de decisión: dados los tiempos de procesamiento de cada uno de los n trabajos en cada una de las *m* máquinas y un entero no negativo *k*, ¿existe una secuencia tal que el tiempo total de procesamiento sea menor o igual que *k*?

Como se puede observar, en un problema de decisión las instancias se dividen en dos clases: sí y no. Mientras que, en un problema de optimización, se desea obtener las mejores soluciones de cada instancia, cada instancia incluye una función objetivo de su conjunto de soluciones y se busca el óptimo de dicha función.

Podemos clasificar los problemas en cuatro clases de acuerdo con su grado de dificultad:

Definición 7

Problemas indecidibles

Son aquellos problemas para los cuales no se puede escribir un algoritmo. Un problema indecidible es aquel que debería dar una respuesta de «sí» o «no», pero para el cual todavía no existe un algoritmo que dé la respuesta

correcta para todas las entradas posibles. Por ejemplo, se ha probado que un programa que se detendrá en una Máquina de Turing (1937) es de esta clase. El problema de Turing de parar, al igual que el décimo problema de Hilbert, es decir no solo no existe un algoritmo polinomial para su solución, sino que no existe algoritmo.

Observación 3

Una máquina de Turing es un dispositivo que manipula símbolos sobre una tira de cinta de acuerdo con una tabla de reglas. A pesar de su simplicidad, una máquina de Turing puede ser adaptada para simular la lógica de cualquier algoritmo de computadora y es particularmente útil en la explicación de las funciones de una CPU dentro de una computadora.

Originalmente fue definida por el matemático inglés Alan Turing como una «máquina automática» en 1936 en la revista *Proceedings of the London Mathematical Society*. La máquina de Turing no está diseñada como una tecnología de computación práctica, sino como un dispositivo hipotético que representa una máquina de computación. Las máquinas de Turing ayudan a los científicos a entender los límites del cálculo mecánico.[11]

11 Hodges, Andrew (1983). Alan Turing: The Enigma. Reino Unido: Burnett Books/ Hutchinson. ISBN 0-671-49207-1. Minsky, Marvin (1967). «Unsolvability of the Halting Problem». Computation: Finite and Infinite Machines. NJ: Prentice-Hall, Turing, A.M. (1936). «On Computable Numbers, with an Application to the Entscheidungsproblem». Proceedings of the London Mathematical Society. 2 (1937) 42: 230-265. doi:10.1112/plms/s2-42.1.230.Turing, A.M. (1938). «On Computable Numbers, with an Application to the Entscheidungsproblem: A correction». Proceedings of the London Mathematical Society. 2 (1937) 43 (6): 544-6. doi:10.1112/plms/s2-43.6.544.

Turing dio una definición sucinta del experimento en su ensayo de 1948 «Máquinas inteligentes». Refiriéndose a su publicación de 1936, Turing escribió que la máquina de Turing, aquí llamada una máquina de computación lógica, consistía en:

> ...«*una ilimitada capacidad de memoria obtenida en la forma de una cinta infinita marcada con cuadrados, en cada uno de los cuales podría imprimirse un símbolo. En cualquier momento hay un símbolo en la máquina; llamado el símbolo leído. La máquina puede alterar el símbolo leído y su comportamiento está en parte determinado por ese símbolo, pero los símbolos en otros lugares de la cinta no afectan el comportamiento de la máquina. Sin embargo, la cinta se puede mover hacia adelante y hacia atrás a través de la máquina, siendo esto una de las operaciones elementales de la máquina. Por lo tanto, cualquier símbolo en la cinta puede tener finalmente una oportunidad*»

Turing (1948, p.61)

El problema de Turing de parar se puede enunciar de la siguiente manera: Comenzamos por imaginar que si existe un algoritmo que determina si un programa para o se detiene.[12]

Basado en su código y una entrada, ¿terminará de ejecutar un programa en particular?

12 https://es.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/solving-hard-problems/a/undecidable-problems#:~:text=Un%20problema%20indecidible%20es%20aquel,para%20todas%20las%20entradas%20posibles.

Por ejemplo, considera este programa que cuenta hacia abajo:

```
num ← 10
REPEAT UNTIL (num = 0){
DISPLAY(num)
num  ← num - 1
}
```

Ese programa parará puesto que num finalmente llega a 0.

Se compara ahora con el siguiente programa que cuenta hacia arriba:

```
num ←1
REPEAT UNTIL (num = 0){
DISPLAY(num)
num  ←num + 1
}
```

Cuenta hacia arriba por siempre, puesto que num nunca será igual a 0.

Hay algoritmos que pueden predecir correctamente que el primer programa para y el segundo no lo hace. Estos son programas sencillos que no cambian con base en entradas diferentes. Sin embargo, no existe un algoritmo que pueda analizar el código de cualquier programa y determinar si para o no.

Para un mayor entendimiento de este problema que en sí resulta complicado, se recomienda el siguiente video animado: https://www.youtube.com/watch?v=92WHN-pAFCs

Definición 8

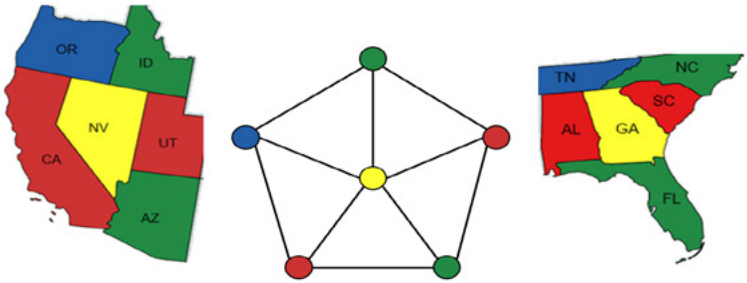
Problemas intratables

(problemas que se demuestran son difíciles)

Son aquellos problemas para los cuales no se pueden desarrollar algoritmos polinomiales. En otras palabras, solo se pueden resolver con algoritmos exponenciales.

Ejemplos de problemas intratables son: el problema de las torres de Hanoi, el problema del agente viajero, el problema del circuito hamiltoniano o el problema de coloración de grafos, los mismos pueden resolverse solamente para instancias pequeñas.

Figura 4. Coloreado de grafos[13]



Ejemplo 14

El caso del ciclo hamiltoniano se puede escribir como sigue:[14]
Entrada: Un grafo G (con vértices V) y $k \in N$.

13 https://www.kaggle.com/c/coloreado-de-grafos
14 http://webdiis.unizar.es/asignaturas/APD/wp/wp-content/uploads/2013/09/200918Intratablesl1.pdf

Salida: ¿Existe un conjunto U de k vértices de G tal que cada arista (i, j) de G cumple que $i \in U$ o $j \in U$?

Reducción de circuito hamiltoniano a B.

Algoritmo para CicHam.

$x \rightarrow$ reducción $\rightarrow$ y algoritmo para B $\rightarrow$ Cic Ham (x)

- Transformamos la entrada de ciclo hamiltoniano en entrada de B en tiempo polinómico.
- La solución de B nos da la solución para ciclo hamiltoniano.
- Como ciclo hamiltoniano es intratable entonces B es intratable.

Observación 4

- Las reducciones en tiempo polinómico son transitivas.
- Si A es reducible a B y B es reducible a C entonces A es reducible a C.
- Si A es intratable podemos demostrar que C es intratable demostrando primero que B es intratable (con $A \leq B$) y después que C es intratable (con $B \leq C$).

Ejemplo 15

El Problema del Agente Viajero (PAV) es intratable.
Este problema se puede enunciar de la siguiente manera:
Entrada: n el número de ciudades, la matriz de distancias $n \times n$ y cota superior k.

Salida: ¿Existe un recorrido por las n ciudades, sin repeticiones y volviendo al punto de partida con distancia total $\leq k$?
Vamos a utilizar una reducción de ciclo hamiltoniano a PAV

- Los dos problemas tratan de encontrar ciclos cortos, PAV en un grafo con pesos.
- Para reducir ciclo hamiltoniano a PAV tenemos que convertir cada entrada de ciclo hamiltoniano (un grafo) en una entrada de PAV (ciudades y distancias).

Ciclo hamiltoniano (G)

```
n = número de vértices de G
        for i = 1 to n
                for j = 1 to n
                        if (i, j) es arista de G
                                d (i, j) = 1
                                else d (i, j) = 2
```

Resultado PAV (n, d, n).

De esta reducción podemos observar lo siguiente:

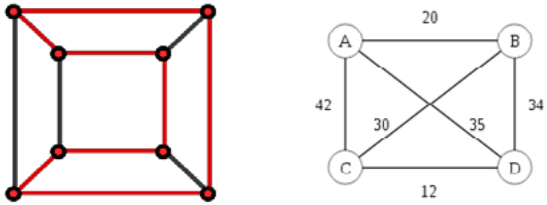
1. La reducción anterior tiene un tiempo de ejecución de $O(n^2)$ (los dos **for** anidados), podemos resolver ciclo hamiltoniano en $O(n^2)$ + el tiempo de PAV.
2. La reducción funciona porque la respuesta de ciclo hamiltoniano con entrada G es la misma que PAV con entrada (n, d, n):

» Si G tiene un ciclo hamiltoniano $(v_1, \cdots, v_n)$ entonces el mismo recorrido para (n, d, n) tiene distancia total $\sum_{i=1}^{n-1} d(v_1, v_{1+1}) + d(v_n, v_1) = n$

» Si G no tiene un ciclo hamiltoniano entonces (n, d, n) no tiene recorrido con distancia total $\leq n$ porque un recorrido así solo puede pasar por n distancias 1, luego pasa por n aristas del grafo y sería un ciclo hamiltoniano de G.

3. Tenemos una reducción eficiente de ciclo hamiltoniano a PAV. Como ciclo hamiltoniano es intratable entonces PAV es intratable.

Figura 5. Un ciclo hamiltoniano y PAV



Para los problemas **P** y **NP** haremos una serie de consideraciones previas a su definición puesto que existe una pregunta abierta en la teoría de la complejidad computacional que se refiere a ¿**P = NP**?

Primero es necesario hacer una distinción entre problemas de búsqueda y problemas de decisión, hasta ahora en este escrito nos hemos enfocado a problemas de decisión ya que es la forma en que la teoría de la complejidad clasifica a los algoritmos para definir su complejidad.

Sobre la formulación de problemas de búsqueda.[15] Los teóricos de la complejidad están tan acostumbrados a centrarse en los problemas de decisión que parecen olvidar que los problemas de búsqueda son al menos tan naturales como los problemas de decisión. Además, a muchos no expertos, los problemas de búsqueda pueden parecer incluso más naturales que los problemas de decisión: Por lo general, las personas buscan soluciones con más frecuencia de lo que se detienen para preguntarse si existen o no soluciones. Por lo tanto, recomendamos comenzar con una formulación de la pregunta **P-vs-NP** en términos de problemas de búsqueda.

Es cierto que el costo representa formulaciones más incomodas, pero vale más que la pena. Con el fin de reflejar la importancia de la versión de búsqueda, así como para facilitar las formulaciones menos engorrosas, decidimos introducir notaciones concisas para las dos clases de problemas de búsqueda que corresponden a **P** y **NP**: Estas clases se denotan PF y PC (que significa Tiempo-polinomial encontrar y Tiempo-polinomial comprobar, respectivamente). Que resulta más intuitivo que polinomial no determinístico.

Sobre la formulación de problemas de decisión. Al presentar el **P-vs-NP** en términos de problemas de decisión, definimos **NP** como una clase de conjuntos que tienen pruebas de afiliación verificables de manera eficiente. Esta definición aclara la naturaleza fundamental de la clase **NP**, pero es cierto que es más complicada que la definición más tradicional de **NP** en términos de ficticios «máquinas no deterministas».

15 A partir de esta parte hasta el final de la sección se traduce parte de lo expuesto en: P, NP, and NP-Completeness: The Basics of Computational Complexity, Oded Goldreich, 2010.

De todo lo discutido hasta ahora se deduce que gran parte de la Teoría de la Complejidad se refiere a encontrar algoritmos eficientes. Estos últimos se definen como algoritmos en tiempo polinomial (es decir, algoritmos que tienen una complejidad de tiempo acotado superiormente por un polinomio en la longitud de la entrada). Por la tesis de Cobham-Edmonds, la definición de esta clase es invariable bajo la elección de un modelo de cálculo «razonable y general». La asociación de algoritmos eficientes con cálculo de tiempo polinomial se basa en las siguientes dos consideraciones:

La tesis de Cobham, también conocida como tesis de *Cobham-Edmonds* (llamada así por Alan Cobham y Jack Edmonds), afirma que los problemas computacionales se pueden calcular de manera factible en algún dispositivo computacional solo si se pueden calcular en tiempo polinómico; es decir, si se encuentran en la clase de complejidad **P**. En términos modernos, identifica problemas tratables con la clase de complejidad **P**.[16]

1. *Consideración filosófica:* Intuitivamente, los algoritmos eficientes son aquellos que se puede implementar dentro de un número de pasos que corresponden a una función de entrada con un crecimiento moderado. Para permitir la lectura de toda la entrada, al menos se debe permitir un tiempo lineal. Por otro lado, los algoritmos aparentemente lentos y en particular los algoritmos de «búsqueda exhaustiva», que toman tiempo exponencial, deben evitarse. Además, una buena definición de la clase de eficiente de los algoritmos es que deben estar cerrados bajo la composición natural de los algoritmos (así como ser robustos con respecto a modelos razonables de cálculo y con respecto a cambios simples en la codificación de instancias de problemas). Elegir polinomios como

16 https://en.wikipedia.org/wiki/Cobham%27s_thesis

el conjunto de límites de tiempo para algoritmos eficientes satisface todos los requisitos anteriores: Los polinomios constituyen un sistema de funciones «cerrado» de crecimiento moderado, donde «cierre» significa cierre bajo suma, multiplicación y composición de funciones. Estas propiedades de cierre garantizan la cerradura de la clase de algoritmos eficientes bajo condiciones naturales de composición de los algoritmos, así como su robustez. Además, los algoritmos de tiempo polinomial P pueden realizar cálculos aparentemente simples (aunque no necesariamente triviales), y por otro lado no incluyen algoritmos que son aparentemente ineficientes (como la búsqueda exhaustiva).

2. *Consideración empírica:* Está claro que los algoritmos que se consideran eficientes en la práctica tienen un tiempo de ejecución acotado por un pequeño polinomio (al menos sobre las entradas que se dan en la práctica). La pregunta es si un algoritmo de tiempo polinomial puede considerarse eficiente en un sentido intuitivo. La creencia, apoyada por experiencias pasadas, es que todo problema natural que se puede resolver en tiempo polinomial también tiene un algoritmo «razonablemente eficiente».

Además del papel dominante de los problemas de búsqueda en las ciencias de la computación, resolver problemas de búsqueda corresponde a la noción cotidiana de «resolver problemas». Por lo tanto, los problemas de búsqueda son de interés general natural. Aquí vamos a considerar la cuestión de qué problemas de búsqueda se pueden resolver de manera eficiente. De hecho, los problemas de búsqueda que se pueden resolver de manera eficiente son el tema de la mayoría de los cursos básicos de diseño algorítmico. Los ejemplos incluyen clasificar, encontrar patrones en cadenas, encontrar soluciones (racionales) a sistemas lineales de ecuaciones (racionales).

Una condición necesaria para una solución eficiente. Un tipo de problemas de búsqueda que no se pueden resolver eficientemente consiste en aquellos para los cuales las soluciones son demasiado largas en términos de la longitud de las instancias del problema, en cuyo caso, simplemente escribir la solución equivale a una actividad que se considera ineficiente, y entonces este caso no es realmente interesante (desde un punto de vista computacional).

Por lo tanto, consideramos solo problemas de búsqueda en los que la longitud de la solución es acotada por un polinomio en la longitud de la instancia. Enfocamos nuestra atención a las relaciones acotadas polinomialmente, recordando que estos problemas de búsqueda están asociados con relaciones binarias.

Definición 9

Relaciones polinomialmente acotadas

Decimos que $R \subseteq \{0, 1\}* \times \{0, 1\}*$ está polinomialmente acotado si existe un polinomio p tal que para cada $(x, y) \in R$ se cumple que $|y| \leq p(|x|)$.

Recuerde que $(x, y) \in R$ significa que y es una solución al problema de instancia x, donde R representa el problema en sí. Por ejemplo, en el caso de encontrar un factor primo de un entero dado, nos referimos a una relación R tal que $(x, y) \in R$ si el entero y es un factor primo del entero x. Asimismo, en el caso de encontrarse un árbol de expansión en una gráfica dada, nos referimos a una relación R tal que $(x, y) \in R$ si y es un árbol de expansión de la gráfica x.

Para una relación polinomialmente acotada R, tiene sentido preguntar si dada una instancia de un problema x, se puede encontrar eficientemente una solución adecuada y (es decir, encontrar y tal que $(x, y) \in R$). El polinomio ligado a la longitud de la solución (es decir, y) garantiza que una respuesta negativa no se deba simplemente a la longitud de la solución requerida. Una definición corta de los problemas **P** es la siguiente:

Definición 10

Problemas P (donde P se entiende por polinomial)

Esta clase incluye todos los problemas que tienen algoritmos de tiempo polinomial. Varios autores consideran a esta clase como una subclase propia de la clase NP.

Un problema se dice polinomial si existe un algoritmo para el cual el tiempo requerido para su solución, está acotado por una función polinomial del tamaño del problema (donde entendemos el tamaño del problema como la longitud de un código, por ejemplo, binario de los datos del problema). Se tiene así por ejemplo que en una gráfica $G = [X, A]$ con $N = X$ nodos y $M = A$ arcos, una ruta más corta se encuentra a lo más en un tiempo $O(m\,n)$, un flujo máximo en un tiempo $O(n^3)$, un árbol de peso mínimo en un tiempo $O(n^2)$. Sin embargo, no todos los problemas combinatorios son polinomiales.

La clase P como una clase natural de problemas de búsqueda

Recuerde que estamos interesados en la clase de problemas de búsqueda que pueden resolverse eficientemente, es decir, problemas para

los cuales las soluciones (siempre que existan) pueden encontrarse de manera eficiente. Restringiendo nuestra atención a las relaciones acotadas polinomialmente, identificamos la clase fundamental correspondiente de problemas de búsqueda (o relaciones binarias), denotado *PF* (que significa «búsqueda de tiempo polinomial»).

Definición 11

Problemas de búsqueda solucionables eficientemente

- El problema de búsqueda de una relación polinomialmente acotada $R \subseteq \{0, 1\}* \times \{0, 1\}*$ es soluble eficientemente si existe un algoritmo de tiempo polinomial A tal que, para todo x, se cumple que si $R(x) =^{def} \{y \mid (x, y) \in R\}$ no es vacío, entonces $A(x) \in R(x)$, y en caso contrario $A(x) = \perp$ (lo que indica que x no tiene solución).
- Denotamos por *PF* la clase de problemas de búsqueda (acotados polinomialmente) que son eficientemente resolubles. Es decir, $R \in PF$ si R está polinomialmente acotado y existe un algoritmo de tiempo polinomial que resuelve R.

Tenga en cuenta que $R(x)$ denota el conjunto de soluciones válidas para la instancia del problema x.

Por lo tanto, se requiere que el algoritmo A encuentre una solución válida (es decir, que satisfaga $A(x) \in R(x)$) siempre que exista tal solución (es decir, $R(x)$ no está vacío). Por otro lado, si la instancia x no tiene solución (es decir, $R(x) = \varnothing$), entonces claramente $A(x) \in R(x)$.

La condición extra requiere que en este caso $A(x) = \perp$. Por lo tanto, el algoritmo A siempre genera una respuesta correcta, que es una solución válida en el caso de que tal solución exista (y proporciona una indicación de que de lo contrario no existe solución).

Hemos definido una clase fundamental de problemas, y conocemos muchos problemas naturales en esta clase (por ejemplo, resolver ecuaciones lineales sobre los racionales, encontrar caminos más cortos en gráficos, encontrar patrones en cadenas, encontrar un perfecto y una variedad de otros problemas de búsqueda). Sin embargo, estos hechos *per se* no significan que somos capaces de caracterizar los problemas naturales con respecto a la pertenencia a esta clase. Por ejemplo, no sabemos si el problema de encontrar los factores primos de un entero dado está en esta clase (es decir, en *PF*).

De hecho, actualmente, no tenemos una buena comprensión con respecto a los contenidos reales de la clase *PF*; es decir, somos incapaces de caracterizar muchos problemas naturales con respecto a la pertenencia a esta clase. Esta situación es bastante común en la teoría de la complejidad, y parece ser una consecuencia del hecho que las clases de complejidad se definen en términos del «comportamiento externo» (de algoritmos potenciales), en lugar de en términos de la «estructura interna» (del problema).

Volviendo a *PF*, observamos que, si bien contiene muchos problemas naturales de búsqueda, también hay muchos problemas de búsqueda natural que no son conocidos por estar en *PF*. A continuación, se presenta una clase natural que contiene una gran cantidad de tales problemas.

1
2
3
4
5
6
7
8
9
10
11
12
13

La clase NP como otra clase natural de problemas de búsqueda

Los problemas de búsqueda natural tienen la propiedad de que las soluciones válidas (para ellos) pueden ser reconocidas eficientemente. Es decir, dada una instancia x del problema R y una solución candidata y, se puede determinar eficientemente si y es o no una solución válida para x (con respecto al problema R, es decir, sea o no $y \in R(x)$). Por ejemplo, soluciones candidatas para un sistema de ecuaciones lineales (o polinomiales) se puede verificar fácilmente su validez mediante la creación de instancias y la manipulación aritmética. Asimismo, es fácil verificar si una determinada secuencia de vértices constituye una trayectoria hamiltoniana en un grafo dado.

La clase de todos los problemas de búsqueda que permiten soluciones eficientes reconocibles (válidas) es una clase natural *per se*, porque no está claro por qué uno debería preocuparse por una solución a menos que uno pueda reconocer una solución válida una vez dada. Además, esta clase es un dominio natural de los candidatos a *PF,* porque la capacidad de reconocer eficientemente una solución válida parece ser un requisito previo natural (aunque no absolutamente necesario) para una discusión sobre la complejidad de encontrar tales soluciones.

Restringimos nuestra atención nuevamente a las relaciones acotadas polinomialmente, y consideramos la clase de relaciones para las cuales la pertenencia de pares en la relación puede decirse eficientemente. Hacemos hincapié en que consideramos decidir la pertenencia de determinados pares de la forma (x, y) en una relación fija R, y no decidir la pertenencia de x en el conjunto $S_R = ^{\text{def}} \{x \mid R(x) \neq \phi\}$.

Definición 12

Problemas de búsqueda con soluciones comprobables de manera eficiente

- El problema de búsqueda de una relación polinomialmente acotada $R \subseteq \{0, 1\}* \times \{0, 1\}*$ tiene soluciones comprobables eficientemente si existe un algoritmo de tiempo polinomial A tal que, para cada x e y, se cumple que $A(x, y) = 1$ si y solo si $(x, y) \in R$.
- Denotamos por PC (que significa «comprobación de tiempo polinomial») la clase de problemas de búsqueda que corresponden a relaciones binarias acotadas polinomialmente que tienen soluciones comprobables eficientemente. Es decir, $R \in PC$ si se cumplen las dos condiciones siguientes:

  i. Para algún polinomio p, si $(x, y) \in R$ entonces $|y| \leq p(|x|)$.
  ii. Existe un algoritmo de tiempo polinomial que dado (x, y) se determina si $(x, y) \in R$.

Note que el algoritmo postulado en el punto ii) también debe manejar las entradas de la forma (x, y) tal que $|y| > p(|x|)$. Tales insumos, que evidentemente no están en R (por el inciso i)), son fáciles de manejar simplemente determinando $|x|, |y|$ y $p(|x|)$.

Por lo tanto, el quid del punto 2 suele estar en el caso de que la entrada (x, y) satisfaga $|y| \leq p(|x|)$.

La clase *PC* contiene miles de problemas naturales (por ejemplo, encontrar un recorrido del agente viajero de una duración que no exceda un umbral dado, encontrar la factorización prima de un compuesto dado,

encontrando una asignación de verdad que satisface una fórmula booleana dada, etc.). En cada uno de estos problemas naturales, la corrección de las soluciones se puede verificar de manera eficiente (por ejemplo, dado un recorrido del agente viajero es fácil calcular su longitud y comprobar si excede o no el umbral dado).

P y NP como problemas de decisión

Sin embargo, deseamos enfatizar que los problemas de decisión son interesantes y naturales per se (es decir, más allá de su papel en el estudio de los problemas de búsqueda). Después de todo, la gente se preocupa por la verdad, por lo que determinar si ciertas afirmaciones son ciertas es un problema computacional natural.

¿P-vs-NP?

La pregunta se refiere a la complejidad de resolver tales problemas para una clase amplia y natural de propiedades asociadas a la clase NP. Esta última clase se refiere a propiedades que tienen «sistemas de prueba eficientes» que permiten la verificación de la afirmación de que un objeto dado tiene una propiedad predeterminada (es decir, es miembro de un conjunto predeterminado). Adelantándonos, mencionamos que la pregunta P-vs-NP se refiere a la pregunta de si las propiedades que tienen sistemas de demostración eficientes pueden decidirse también de manera eficiente (sin pruebas).

No hace falta decir que estamos interesados en la clase de problemas de decisión que son solubles eficientemente. Esta clase se denota

tradicionalmente como P (que significa tiempo polinomial). La siguiente definición se refiere a la formulación de la solución de problemas de decisión.

Definición 13

Problemas de decisión que se resuelven eficientemente

- Un problema de decisión $S \subseteq \{0, 1\}*$ es soluble eficientemente si existe un algoritmo de tiempo polinomial A tal que, para cada x, se cumple que $A(x) = 1$ si y solo si $x \in S$.
- Denotamos por **P** la clase de problemas de decisión que son resolubles eficientemente.

De hecho, hay muchos problemas naturales de decisión que no se sabe si pertenecen a **P**, y esa clase natural que contiene una serie de tales problemas se denota **NP**.

Cada vez que decidir por nuestra cuenta parece difícil, es natural buscar ayuda (por ejemplo, consejos) de otros. En el contexto de verificar que un objeto tiene una propiedad predeterminada (o pertenece a un conjunto predeterminado), la ayuda puede tomar la forma de una prueba, donde las pruebas deben ser pensadas como consejos que pueden ser evaluados con exactitud. De hecho, una clase natural de problemas de decisión que surge es la clase **NP**, de todos los conjuntos tal que la pertenencia (de cada instancia) en cada conjunto puede verificarse eficientemente con la ayuda de una prueba adecuada. Así, definimos **NP** como la clase de problemas de decisión que tienen sistemas de prueba eficientemente verificables. Este camino definitorio requiere aclarar la noción de un sistema de prueba.

En términos generales, decimos que un conjunto S tiene un sistema de prueba si las instancias en S tienen pruebas válidas de membresía (es decir, pruebas aceptadas como válidas por el sistema), mientras que las instancias que no están en S no tienen pruebas válidas. De hecho, las pruebas se definen como cadenas que (cuando acompañan a la instancia) son aceptadas por el (eficiente) procedimiento de verificación. Es decir, decimos que V es un procedimiento de verificación para pertenencia a S si cumple las dos condiciones siguientes:

1. *Completez:* las afirmaciones verdaderas tienen pruebas válidas (es decir, pruebas aceptadas como válidas por V). Teniendo en cuenta que las afirmaciones se refieren a la pertenencia a S, esto significa que para todo $x \in S$ existe una cadena y tal que $V(x, y) = 1$; es decir, V acepta y como prueba válida de la pertenencia de x a S.
2. *Solidez:* Las afirmaciones falsas no tienen pruebas válidas. Es decir, para todo $x \notin S$ y cada cadena y se tiene que $V(x, y) = 0$, lo que significa que V rechaza y como prueba de la pertenencia de x a S.

Notamos que la condición de solidez captura la «seguridad» del procedimiento de verificación, es decir, su capacidad de no dejarse engañar (por nada) para aceptar una afirmación incorrecta. La condición de completitud captura la «viabilidad» del procedimiento de verificación, es decir, su capacidad para ser convencido de cualquier afirmación válida (cuando se presenta con una prueba adecuada).

Consideremos un par de ejemplos antes de pasar a la definición formal (de sistemas de prueba eficientemente verificables). Comenzando con el conjunto de gráficas hamiltonianas, notamos que este conjunto tiene un procedimiento de verificación que, dado un par (G, π), se acepta si y solo si π es un camino hamiltoniano en el grafo G. En este caso, π sirve

como prueba de que *G* es hamiltoniano. Considere que tales pruebas son relativamente cortas (es decir, el camino es en realidad más corto que la descripción del gráfico) y son fáciles de verificar. No hace falta decir que este sistema de prueba satisface las condiciones de completez y solidez.

Pasando al caso de fórmulas booleanas satisfactibles, dada una fórmula φ y una asignación de verdad τ, el procedimiento de verificación de la instancia φ (de acuerdo con τ), se acepta si y solo si al simplificar la expresión booleana resultante se obtiene el valor verdadero. En este caso, τ sirve como prueba de que φ es satisfactible, y las presuntas pruebas son de hecho relativamente cortas y fáciles de verificar.

Definición 14

Sistemas de prueba eficientemente verificables

Un problema de decisión $S \subseteq \{0, 1\}*$ tiene un sistema de prueba eficientemente verificable si existe un polinomio p y un algoritmo de tiempo polinomial (verificación) *V* tal que se cumplen las dos condiciones siguientes:

- *Completez:* Para todo $x \in S$, existe y de longitud como máximo $p(|x|)$ tal que $V(x, y) = 1$. (Tal cadena y se llama NP-testigo para $x \in S$.)
- *Solidez:* Para todo $x \notin S$ y todo y, se cumple que $V(x, y) = 0$. Así, $x \in S$ si y solo si existe y de longitud como máximo $p(|x|)$ tal que $V(x, y) = 1$.

En tal caso, decimos que S tiene un sistema de prueba **NP** y nos referimos a V como su procedimiento de verificación (o como el propio sistema de prueba).

Denotamos por **NP** la clase de problemas de decisión que tienen sistemas de prueba eficientemente verificables. Note que el término **NP-testigo** se usa comúnmente. En algunos casos, *V* (o el conjunto de pares aceptados por *V*) se llama una relación testigo de *S*. Hacemos hincapié en que el mismo conjunto *S* puede tener muchos sistemas de prueba NP diferentes y que en algunos casos la diferencia es bastante diferente.

De todo lo anterior se puede inferir que esta definición de NP es más práctica y clara que la popular definición siguiente:

Definición 15

Problemas NP

(donde NP se entiende por polinomial no determinístico)

Esta clase incluye problemas que se pueden resolver en tiempo polinomial si podemos *adivinar* correctamente qué ruta computacional se puede seguir. El concepto de *adivinar* es extraño ya que todos los programas computacionales son determinísticos. En general, esta clase incluye todos los problemas que tienen algoritmos exponenciales pero que no se ha probado que no se puedan resolver con algoritmos de tiempo polinomial.

CONSIDERACIONES SOBRE P ≠ NP

«La mente intuitiva es un regalo sagrado y la mente racional es un fiel sirviente. Hemos creado una sociedad que rinde honores al sirviente y ha olvidado al regalo».

Albert Einstein

Einstein habla de la importancia de ambas consideraciones filosóficas (referidas como la mente racional) y consideraciones empíricas (referidas como la mente intuitiva) para la ciencia. De hecho, seguiremos su ejemplo. Se cree ampliamente que P es diferente de NP, es decir, que PC contiene problemas de búsqueda que no son eficientemente resolubles, y que hay sistemas de NP-prueba para conjuntos que no pueden decidirse eficientemente. Esta creencia es apoyada por consideraciones filosóficas y empíricas.

Consideraciones filosóficas. Ambas formulaciones de la pregunta **P-vs-NP** se refieren a cuestiones naturales sobre las que tenemos concepciones fuertes. La noción de resolver un problema (de búsqueda) parece suponer que, al menos en algunos casos (o en general), encontrar una solución es significativamente más difícil que verificar si una solución presentada es correcta. Esto se traduce en $PC \setminus PF \neq \emptyset$. Asimismo, la noción de prueba parece suponer que, al menos en algunos casos (o en general), la prueba es útil para determinar la validez de la afirmación, es decir, que la verificación de la validez de una afirmación puede ser significativamente más fácil cuando se ha proporcionado una prueba. Esto se traduce en **P ≠ NP**, lo que también implica que es significativamente más difícil encontrar pruebas que verificar su exactitud, lo que nuevamente coincide con la experiencia diaria de investigadores y estudiantes.

Consideraciones empíricas. La clase **NP** (o más bien *PC*) contiene miles de diferentes problemas para los que no se conoce un procedimiento de solución eficiente. Muchos de estos problemas han surgido en disciplinas muy diferentes, y fueron objeto de extensas investigaciones de numerosas comunidades diferentes de científicos e ingenieros Todos estos estudios esencialmente independientes no han proporcionado algoritmos eficientes para resolver estos problemas, una falla que es extremadamente difícil atribuir a pura coincidencia o a una racha de mala suerte.

Mencionamos que, para muchos de los problemas antes referidos, los algoritmos más conocidos no son significativamente más rápidos que una búsqueda exhaustiva (de una solución); es decir, la complejidad del algoritmo más conocido está polinomialmente relacionada con la complejidad de una búsqueda exhaustiva. De hecho, existe la creencia generalizada de que, para algunos problemas en **NP**, ningún algoritmo puede ser significativamente más rápido que una búsqueda exhaustiva.

La creencia (o conjetura) común de que **P ≠ NP** es realmente muy atractiva e intuitiva. El hecho de que esta conjetura natural no se haya resuelto parece ser una de las fuentes de frustración de la teoría de la complejidad. En nuestra opinión, sin embargo, es que este sentimiento de frustración está fuera de lugar (y solo refleja una ingenua subestimación de los problemas en cuestión). Por el contrario, el hecho de que la teoría de la complejidad evoluciona en torno a preguntas naturales y formuladas de forma sencilla que son tan difíciles para resolver hace que su estudio sea muy emocionante.

Meditaciones Filosóficas: Las limitaciones inherentes de nuestro conocimiento científico fueron articuladas por Kant, quien argumentó que nuestro conocimiento no puede trascender nuestra forma de entender.

Los «modos de entender» están predeterminados; preceden a toda adquisición de conocimiento y son la condición previa a tal adquisición. De alguna manera, Wittgenstein refinó el análisis, argumentando que el conocimiento debe formularse en un lenguaje, y este último debe estar sujeto a un mecanismo (sonoro) de asignación de sentido. Así, las limitaciones inherentes de cualquier posible «mecanismo de asignación de significado» impone limitaciones a lo que se puede decir (significativamente).

Ambos filósofos hablaron de la relación entre el mundo y nuestros pensamientos. Dieron por sentado (o más bien asumieron) que en el dominio de los pensamientos bien formulados (por ejemplo, la lógica), cada conclusión válida se puede alcanzar de manera efectiva (es decir, toda aseveración válida puede probarse efectivamente). De hecho, esta suposición ingenua fue refutada por Gödel. De manera similar, la obra de Turing afirma que *existen problemas bien definidos que no pueden ser resueltos por métodos bien definidos.*

Destacamos que la afirmación de Turing trasciende las consideraciones filosóficas del primer párrafo: Afirma que las limitaciones de nuestra capacidad se deben no solo a la brecha entre el «mundo tal como es» y nuestro modelo de él. En contraste, la afirmación de Turing se refiere a las limitaciones inherentes a cualquier proceso racional, incluso cuando este proceso se aplica a información bien formulada y tiene como objetivo una meta bien formulada. De hecho, en contraste con las presunciones ingenuas, no todos los problemas bien formulados pueden ser (efectivamente) resueltos.

La conjetura **P≠NP** va incluso más allá de la afirmación de Turing. Limita el dominio de la discusión a los problemas «justos», es decir, a los problemas para los cuales las soluciones válidas pueden ser reconocidas

eficientemente como tales. De hecho, hay algo fingido en problemas para los que no se pueden reconocer soluciones válidas de manera eficiente. Evitando tales problemas fingidos y/o injustos, **P ≠ NP** significa que (aun con esta limitación) existen problemas que son inherentemente irresolubles en el sentido de que no pueden ser resueltos eficientemente. Es decir, a diferencia de las presunciones ingenuas, no todos los problemas que se refieren a soluciones reconocibles de manera eficiente pueden resolverse eficientemente. De hecho, la brecha entre la complejidad de reconocer soluciones y la complejidad de encontrarlos avala el significado de la noción de un problema.

9 Problemas de optimización

1
2
3
4
5
6
7
8
9
10
11
12
13

Todos los días, los ingenieros y los tomadores de decisiones se enfrentan a problemas de creciente complejidad que surgen en diversos sectores técnicos, por ejemplo, en investigación de operaciones, diseño de sistemas mecánicos, procesamiento de imágenes y particularmente en electrónica. El problema por resolver se puede expresar a menudo como un problema de optimización. Aquí se puede definir una (o varias) función objetivo, o función de costo, que se busca minimizar o maximizar frente a todos los parámetros en cuestión. La definición del problema de optimización se complementa con la información de las restricciones. Todos los parámetros de las soluciones adoptadas deben satisfacer estas restricciones, o de lo contrario estas soluciones no se pueden encontrar.

Se pueden distinguir dos tipos de problemas de optimización: los problemas «discretos» y problemas con variables continuas. Para ser más precisos, citemos dos ejemplos: entre los problemas discretos, se puede discutir el famoso problema del agente viajero donde se trata de

minimizar la longitud de la ronda que debe visitar un determinado número de ciudades, antes de regresar al pueblo de partida. Un ejemplo tradicional de problema continuo es el problema de producción. En la práctica, también se pueden encontrar «problemas mixtos», que comprenden simultáneamente variables discretas y variables continuas.

Esta diferenciación es necesaria para determinar el dominio de los problemas de optimización difíciles. De hecho, dos tipos de problemas son referidos, en la literatura, como problemas de optimización difíciles (este nombre no está estrictamente definido, y de hecho está acotado al estado del arte para la optimización):

- Ciertos problemas de optimización discreta, para los cuales no hay conocimiento de un algoritmo polinómico exacto (es decir, cuyo tiempo de cálculo es proporcional a N^n, donde N es el número de parámetros desconocidos del problema, y n es una constante entera). Es el caso, en particular, de los problemas conocidos como «NP-difícil», por lo que se conjetura que no existe una n constante para el cual el tiempo de solución está limitado por un polinomio de grado n.
- Ciertos problemas de optimización de variables continuas, para los cuales no hay conocimiento de un algoritmo que permita localizar definitivamente un óptimo global (es decir, la mejor solución posible) y en un número completo de cálculos. Fueron muchos los esfuerzos realizados durante mucho tiempo, por separado, para solucionar estos dos tipos de problemas, a los que más adelante nos referiremos brevemente.

Un problema combinatorio es aquel que asigna valores numéricos discretos a algún conjunto finito de variables X, de tal forma que satisfaga un conjunto de restricciones y minimice o maximice alguna función objetivo.

Los problemas de optimización combinatoria abundan en la vida diaria. Una área importante y extensa de aplicaciones se refiere a la administración eficiente del uso de recursos escasos para incrementar la productividad. Estas aplicaciones incluyen problemas operacionales tales como distribución de bienes, planeación de la producción y secuenciación de máquinas. También incluyen problemas de planeación tales como inversión de capital, localización de medios y selección de cartera. También problema de diseño como diseño de redes de telecomunicación y transporte, diseño de circuitos y diseño de sistemas de producción automática. Los problemas de optimización discreta también se presentan en estadística (análisis de datos), física (determinación de estados de energía mínimos), criptografía (diseñando códigos infranqueables), política (seleccionando distritos electorales favorables) y en matemáticas (como una técnica poderosa para probar teoremas combinatorios). Como ya se ha mencionado el gran avance de las computadoras ha repercutido en el desarrollo acelerado de la optimización discreta.

Antes de definir los problemas NP-duros, daremos una definición para los problemas NP-completos.

Definición 16

Problemas NP-completos

En términos generales, un problema en **NP** se llama **NP-completo** si cualquier algoritmo eficiente que lo resuelva se puede convertir en un algoritmo eficiente para cualquier otro problema en **NP**. Por lo tanto, si **NP** es diferente de **P**, entonces no hay problema que pertenezca a **NP-completo** que pueda estar en **P.** La conversión antes mencionada de un algoritmo

eficiente para un problema **NP** en algoritmos eficientes para otros problemas **NP** se desarrolla en realidad por una reducción. Por lo tanto, un problema (en **NP**) es **NP-completo** si cualquier problema en **NP** es eficientemente reducible a él, lo que significa que *cada problema individual **NP** completo "codifica" todos los problemas en **NP**.*

Reflexiones sobre los problemas NP-completos

Sabemos que existen problemas NP-completos. La pregunta que nos hacemos aquí es qué aspectos en nuestro modelado de problemas permiten la existencia de problemas completos. Por supuesto, debemos tener en cuenta que la completitud se refiere a una clase de problemas; el problema completo debe «codificar» cada problema de la clase y estar en la clase. Desde este aspecto, en lo sucesivo denominado como *codificabilidad* de una clase, es lo suficientemente sorprendente (al menos para un profano), comenzamos preguntando qué es lo que lo habilita. Identificamos dos paradigmas fundamentales, con respecto al modelado de problemas, que parecen esenciales para la *codificabilidad* de cualquier clase (infinita) de problemas:

1. Cada problema se refiere a un conjunto infinito de instancias posibles.
2. La especificación de cada problema utiliza una descripción finita (por ejemplo, un algoritmo que enumera todas las soluciones posibles para cualquier instancia dada).

Estos dos paradigmas parecen algo conflictivos, pero juntos sugieren la definición de un problema universal. En concreto, este problema se refiere a instancias de la forma (*D, x*), donde *D* es una descripción de un problema y x es una instancia a ese problema, y una solución a la

instancia (D, x) es una solución a x con respecto al problema (descrito por) D. Intuitivamente, este problema universal puede codificar cualquier otro problema (siempre que los problemas se modelen de una manera que se ajuste a los paradigmas anteriores): Resolver el problema universal permite resolver cualquier otro problema.

Considere que el problema universal anterior es en realidad completo con respecto a la clase de todos los problemas, pero no es completo con respecto a ninguna clase que contiene solo (algorítmicamente) problemas solucionables (porque este problema universal no tiene solución). Dirigiendo nuestra atención a las clases de problemas solucionables, buscamos versiones del problema universal que sean completas para estas clases. Una dificultad arquetípica que surge es que, dada una descripción D (como parte de la instancia al problema universal), no podemos decir si D es o no una descripción de un problema en una clase C predeterminada (porque este problema de decisión es irresoluble). Este hecho es relevante porque si el problema universal requiere resolver instancias que se refieren a un problema que no está en C, entonces intuitivamente no puede estar él mismo en C.

Antes de volver a la resolución de la dificultad anterior, notamos que los paradigmas de modelado antes mencionados son fundamentales para la teoría de la computación en general. En particular, hasta ahora no hemos hecho referencia a ninguna consideración de complejidad. De hecho, una consideración de complejidad es la clave para resolver la dificultad anterior: la idea es modificar cualquier descripción D en una descripción D' tal que D' siempre está en C y D' concuerda con D en el caso de que D esté en C (es decir, en este caso describen exactamente el mismo problema). Hacemos hincapié en que en el caso de que D no esté en C, el problema correspondiente D' puede ser arbitrario (siempre y cuando esté en C). Tal modificación es posible con respecto a muchas clases teóricas de complejidad.

Consideramos dos tipos diferentes de clases, donde en ambos casos, la clase se define en términos de la complejidad temporal de los algoritmos que hacen algo relacionado con el problema (por ejemplo, reconocer soluciones válidas, como en la definición de **NP**).

1. *Clases definidas por una única función t limitada en el tiempo* (p. ej., $t(n) = n^3$). En este caso, cualquier algoritmo D se modifica al algoritmo D' de tal manera que en la entrada x, emula (hasta) $t(|x|)$ pasos de la ejecución de $D(x)$. La versión modificada del problema universal trata la instancia (D, x) como (D', x). Esta versión puede codificar cualquier problema en dicha clase C (correspondiente al tiempo de complejidad t).

Pero ¿estará este problema (versión del universal) en sí mismo en C? La respuesta depende tanto de la eficiencia de la emulación en el cálculo computacional del modelo correspondiente y en la tasa de crecimiento de t. Por ejemplo, para la triple exponencial de t, la respuesta definitivamente será sí, porque $t(|x|)$ pasos se puede emular en tiempo $poli(t(|x|))$ (en cualquier modelo razonable) mientras $t(|(D,x)|) > t(|x| + 1) > poli(t(|x|))$. Por otra parte, en los modelos más razonables, la emulación de $t(|x|)$ pasos requiere más de $O(t(|x|))$ tiempo, mientras que para cualquier polinomio t se cumple que $t(n + O(1))$ es menor que $2t(n)$.

2. *Las clases definidas por una familia infinita de funciones de diferente tasa de crecimiento (por ejemplo, polinomios).* Podemos, por supuesto, seleccionar una función t que crece más rápido que cualquier función en la familia y proceder como en el caso anterior, pero entonces el problema universal resultante definitivamente no estará en la clase.

Note que, en el caso actual, un problema completo será sorprendente porque, en particular, estará asociado con una función t_0 que crece más

moderadamente que algunas otras funciones en la familia (por ejemplo, un polinomio fijo crece de forma más moderada que otros polinomios). Aparentemente esto significa que el algoritmo que describe la máquina universal debe ser más rápido en términos del número real de pasos que algunos algoritmos que describen algunos otros problemas en la clase. Esta impresión supone que las instancias de ambos problemas son (aproximadamente) de la misma longitud, y entonces violamos intencionalmente esta presunción aumentando artificialmente la longitud de la descripción de las instancias al problema universal. Por ejemplo, si D está asociado con la cota de tiempo t_D, entonces la instancia (D, x) al problema universal se presenta como, $\left(D, x, 1t_0^{-1(tD(|x|)2)}\right)$, donde el cuadrado compensa la sobrecarga de la emulación (y en el caso de **NP** usamos $t_0(n) = n$).

Volviendo a la definición de NP-Duros podemos afirmar que una comprensión intuitiva de los problemas **NP-Duros** y **NP-Completos** se refiere a aquellos que son difíciles (y probablemente imposibles) de encontrar algoritmos de tiempo polinomial para su solución. Por lo tanto, las personas que generalmente son nuevas en algoritmos se harán esta pregunta:

¿Cuál es la diferencia entre NP-Duros y NP-Completos?

La respuesta simple es por definición: si todos los problemas **NP** pueden reducirse al problema A por polinomio, entonces el problema A es NP-Difícil; si el problema A es NP-Difícil y NP, entonces es NP-Completo.

De la definición, podemos ver fácilmente que la clase de problema NP-Duro incluye la clase NP-Completo. Pero además preguntaremos:

¿Hay algún problema que pertenezca a NP-Duro, pero no a NP-Completo?

La respuesta es sí.

Por ejemplo, el problema de parada, es decir, se dan un programa y una entrada para determinar si se terminará su operación. El problema de la detención es indecidible, por lo que, por supuesto, no es un problema de NP. Pero para problemas NP-Completos como SAT, se puede reducir al problema de detención por polinomio. Como podemos construir el programa *A*, el programa agota todas las asignaciones de sus variables a la fórmula de entrada, si hay una asignación para hacerla verdadera, se detiene, de lo contrario entra en un bucle infinito. De esta forma, juzgar si la fórmula puede satisfacerse se transforma en juzgar si el programa *A* que toma la fórmula como entrada está detenido. Por lo tanto, el problema de parada es NP-Duro en lugar de NP-Completo.[17]

Un problema NP-Duro es aquel que no es soluble en tiempo polinomial, pero se puede verificar en tiempo polinomial.

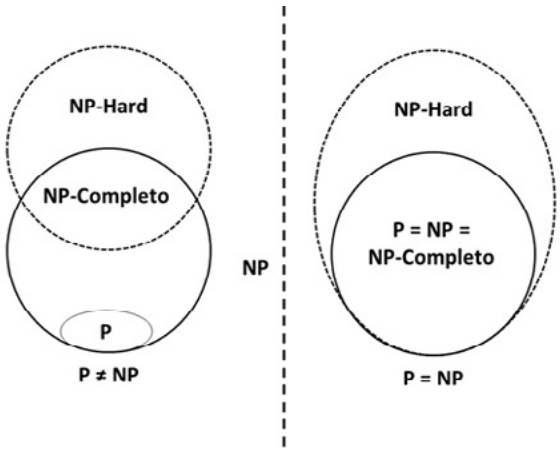
Un problema NP-Completo es aquel que es NP y también NP-Duro.

En términos más esquemáticos[18]:

NP-DURO	NP-COMPLETO
Los problemas NP-Difíciles (digamos X) se pueden resolver si y solo si hay un problema NP-Completo (digamos Y) que se pueda reducir a X en tiempo polinomial.	Los problemas NP-Completos pueden resolverse mediante un algoritmo / máquina de Turing no determinista en tiempo polinomial.

17 http://hi.baidu.com/nuclearspace/item/e0f8a1b7779149742 54b09f4
18 https://es.acervolima.com/diferencia-entre-problema-np-duro-y-np-completo/

Figura 6. Diagrama de los problemas P, NP, NP- completo y NP-duro



El problema de satisfactibilidad (SAT) fue el primer problema identificado como perteneciente a la clase de complejidad NP-completo por Stephen Cook en el año 1971, y se plantea de la siguiente manera:[19]

Comenzamos con una lista de variables booleanas $x_1, \cdots, x_n$. Un *literal* es una de las variables x_i (o la negación de una de las variables $\neg x_i$). Hay $2n$ literales posibles. Una *cláusula* es un conjunto de literales.

Las reglas del juego son las siguientes: Asignamos valores booleanos *Verdadero* (*V*) o *Falso* (*F*) a cada una de las variables. De este modo a cada uno de los literales se le asigna un valor booleano. Finalmente, una cláusula tiene valor *V* si y solo si al menos uno de los literales de la cláusula tiene un valor *V*, en otro caso tendrá un valor *F*.

19 https://www.xatakaciencia.com/computabilidad/problema-de-satisfacibilidad-sat

Un conjunto de cláusulas es satisfactible si existe una asignación de valores booleanos a las variables que *hagan* que todas las cláusulas sean ciertas. Consideramos *or* entre cada uno de los literales en una cláusula y *and* entre las cláusulas.

El problema de satisfactibilidad (SAT). *Dado un conjunto de cláusulas, ¿existe un conjunto de valores booleanos para una determinada expresión que la haga verdadera?*

Ejemplo 16

Consideramos el conjunto de variables x_1, x_2, x_3. Podemos construir la siguiente lista de cláusulas.

$$\{x_1, \neg x_2\}\ \{x_1, x_3\}\ \{x_2, \neg x_3\}\ \{\neg x_1, x_3\}$$

Si elegimos los valores (*V, V, F*) para las variables (x_1, x_2, x_3) respectivamente, entonces los valores de las cuatro cláusulas serán (*T, T, T, F*), así que no podría ser una asignación válida para satisfacer el conjunto de cláusulas.

Existen 8 posibles asignaciones ($2^{n=3}$). Al final obtenemos como asignación satisfactoria a (*T, T, T*).

El ejemplo nos deja la sensación de que SAT debe ser un problema computacional complicado, porque hay 2^n posibles conjuntos de valores que pueden resolver el problema.

Está absolutamente claro, sin embargo, que el problema pertenece a la clase de complejidad NP. Efectivamente, es un problema de decisión. Además, podemos asignar fácilmente un certificado a todos los conjuntos de cláusulas para cual la respuesta a SAT es: *Sí, las cláusulas son satisfactibles.*

El certificado contiene un conjunto de valores, uno por cada variable, que satisface todas las cláusulas. Una máquina de Turing que recibe un conjunto de cláusulas, apropiadamente codificadas, como entrada, acompañadas del certificado tendría que verificar solamente que si los valores son asignados a las variables como se muestra en el certificado, entonces efectivamente cada cláusula contiene al menos una literal de valor V. Esa verificación se realiza en tiempo polinómico.

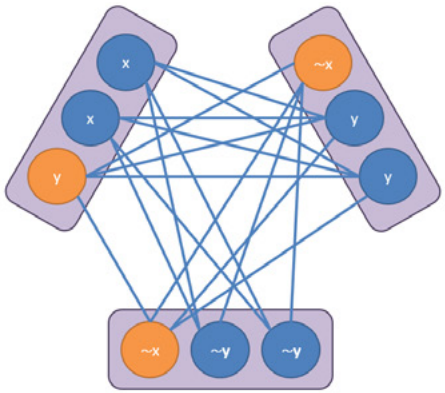


Figura 7. El problema SAT[20]

Ejemplo 17

Problemas como el del agente viajero o el de la mochila se conocen como NP completos. El problema del agente viajero se puede resolver con un algoritmo determinístico como el de recursividad, donde se especifica en cada paso qué arco se debe considerar desde un nodo dado.

20 https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

Para la pregunta: si existe un recorrido con una distancia total que sea menor que *B*, el algoritmo recursivo implícitamente prueba todos los recorridos posibles y da una respuesta afirmativa si tal recorrido existe y negativa en caso contrario. Ya que los recorridos se prueban uno por uno, y el último puede ser el que cumple con la propiedad deseada, el algoritmo recursivo para resolver el problema del agente viajero se considera $O(c^n)$.

Un algoritmo no determinístico puede adivinar correctamente qué arco se debe incluir en el recorrido. Si existe un recorrido con una distancia total menor que *B*, el algoritmo no determinístico requiere un tiempo de $O(n)$ para calcular la distancia total del recorrido y verificar si tal recorrido existe. Si el problema del agente viajero se puede resolver por un algoritmo no determinístico en tiempo polinomial se dice que el problema pertenece a la clase **NP**.

Figura 8. El juego del icosiano, origen del PAV

Ejemplo 18

Por otro lado, se tienen problemas en la categoría NP-Duros como el Problema de asignación cuadrática (QAP), que se enuncia de la siguiente manera: Asignar *n* instalaciones a *n* locaciones de manera que el costo sea mínimo (Koopmans Beckmann, 1957). Que tiene amplia aplicabilidad

en: distribución de instalaciones en hospitales, distribución de componentes circuitales, disminución de la fatiga ocular entre otros.[21]

Para demostrar que QAP es NP-Duro se hace uso de la técnica de reducción que ya vimos con anterioridad: SAT primer problema NP-Completo (Teorema de Cook-Levin 1971) todo problema NP se reduce a SAT.

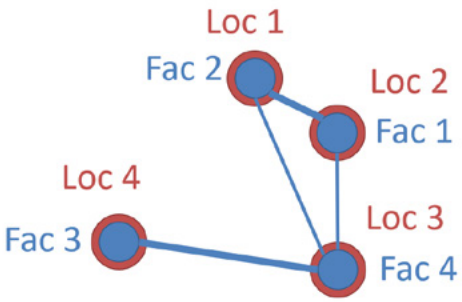


Figura 9. Problema de asignación cuadrática

Podemos considerar un problema con el enfoque de optimización y con el enfoque de decisión y tendremos dos clasificaciones de este, ya sea NP-Duro o NP-Completo, tal es el caso del PAV como veremos en el siguiente ejemplo, pero antes recordemos las definiciones de tales problemas dadas en la sección 8.

Problema de optimización

Un problema de optimización tiene como objetivo encontrar una solución que sea óptima entre todas las soluciones factibles que se tengan, estas soluciones factibles se pueden considerar como decisiones factibles, y están sujetas a una serie de restricciones que deben cumplir.

21 https://jcc.dcc.fceia.unr.edu.ar/2016/slides/2016-paredes-restrepo.pdf

Problema de decisión

Un problema de decisión es un tipo especial de problema computacional cuya respuesta es solamente «sí» o «no» (o, de manera más formal, «1» o «0»).

Ejemplo 19

En este ejemplo se muestra el problema del Agente Viajero (PAV) en tres versiones diferentes:

PAV problema de decisión
Entrada: Una gráfica *G*, un presupuesto *b*
Salida: ¿*G* admite un recorrido de a lo más *b*? (Sí/No)
Clasificación: NP- Completo

PAV problema de búsqueda
Entrada: Una gráfica *G*, un presupuesto *b*
Salida: Encontrar un recorrido en *G* con un peso de a lo más *b*, si es que existe.
Clasificación: NP-duro

PAV problema de optimización
Entrada: Una gráfica *G*
Salida: Encontrar un recorrido en *G* con un peso mínimo
Clasificación: NP-duro

Fuera de contexto, el problema del TSP podría referirse a cualquiera de estos. El problema aquí es que estos problemas son estrictamente distintos. La versión de decisión solo pide existencia, mientras que la versión de

búsqueda pide más, necesita un ejemplo de tal solución. En la práctica, a menudo queremos tener la solución real, por lo que los enfoques más prácticos pueden omitir mencionar los problemas de decisión.

Ahora, ¿qué pasa con eso? La definición de un problema NP-completo estaba pensada para problemas de decisión, por lo que técnicamente no se aplica directamente a problemas de búsqueda u optimización. Pero debido a que la teoría detrás de esto está bien definida y es útil, entonces es útil seguir aplicando el término NP-completo/NP-difícil de buscar/problema de optimización, para que tengan una idea de cuán difíciles son de resolver estos problemas. Entonces, cuando alguien dice que el problema del viajero es NP-completo, formalmente debería ser la versión del problema de decisión.

Evidentemente, muchas nociones se pueden ampliar para que abarquen también problemas de búsqueda, y así se presentan en algunos libros.

A continuación, se listan otros problemas representativos que abordan la optimización con su respectiva complejidad computacional y que se estudian en los diferentes cursos de Investigación de Operaciones, de acuerdo con la definición de problemas de optimización se tiene la siguiente clasificación:

- ***El problema del cartero chino:*** Para una gráfica dada (dirigida o no) con pesos específicos en los arcos, determine una trayectoria que incluya cada arco en la gráfica al menos una vez y que su peso total sea mínimo. NP-duro.

- ***El problema de la mochila:*** Determine un conjunto de valores enteros x_i con $i = 1, 2, \cdots, n$ que minimice $f(x_1, x_2, \cdots, x_n)$ sujeto a la restricción $g(x_1, x_2, \cdots, x_n) \geq b$ donde b es un parámetro. NP-duro.

- ***Planeación de máquinas en paralelo:*** Dado un conjunto T de tareas simples de operación, cada una con tiempo de procesamiento τ_j, $1 \leq j \leq |T|$, asignar cada tarea a exactamente m máquinas tal que el tiempo en que se realizan todas las tareas sea mínimo. NP-duro.

- ***Coloración de nodos:*** Dada una gráfica no dirigida, determine el número mínimo de colores necesarios para colorear cada nodo de la gráfica de tal forma que ningún par de nodos adyacentes (nodos conectados por un arco) tengan el mismo color. NP-completo.

- ***Árbol de expansión mínima:*** Para una gráfica dada con pesos específicos en los arcos, determinar un subconjunto de arcos de peso mínimo total que forme una gráfica acíclica conectada que tiene al menos un arco incidente a cada vértice. NP-completo.

- ***Ruta más corta:*** Para una gráfica dada (dirigida o no) con peso o longitudes específicas en los arcos, encontrar una sucesión de arcos no repetidos de longitud total mínima que conecte dos vértices específicos y que sea factible a la dirección de los arcos. NP- duro.

- ***Empacado de cajas:*** Para una lista de N pesos w_i, $1 \leq i \leq N$ y un conjunto de cajas, cada una de ellas con una capacidad fija, sea W, encontrar una asignación factible de pesos para las cajas que minimice el número total de cajas por usar. NP- duro.

- ***Apareamiento:*** Dada una lista de artículos $i = 1, 2, \cdots, n$ y pesos w_{ij} asociados con aparear un artículo i con un artículo j, encontrar un esquema de peso máximo total para aparear los artículos en la lista tal que cada artículo este apareado con, a lo más, otro artículo. NP-completo.

- ***Cubierta de conjuntos:*** Dado un conjunto finito S y una familia de subconjuntos $\{S_j \subseteq S | j \in J\}$ y costos asociados C_j a cada S_j, elegir una colección de subconjuntos de costo mínimo total que incluya a cada elemento de S al menos una vez. NP-completo.

- ***Flujo máximo:*** Dada una gráfica (dirigida o no) y capacidades específicas en los arcos, encontrar un flujo máximo entre dos vértices específicos que sea factible con respecto a las capacidades. NP-completo.

- ***Problema de cargo fijo:*** Dado un conjunto factible S de actividad o niveles de tráfico no negativos, $x = (x_1, x_2, \cdots, x_n)$, costos unitarios v_j para emplear x_j, y costos fijos f_j asignados siempre que x_j sea positiva, seleccionar un costo mínimo local $x \in S$. NP-completo.

De acuerdo con la definición de problemas de decisión los siguientes ejemplos se clasifican de la siguiente manera:

1. Problemas P:
   - » Ordenar un arreglo
   - » Ruta más corta entre 2 puntos
   - » Calcular el determinante de una matriz
   - » Programación lineal

2. Problemas NP-completo:
   - » Problema del agente viajero
   - » Programación lineal entera

3. Problemas intratables:
   - » Determinar todos los puntos enteros que satisfacen un sistema de desigualdades lineales
   - » Las torres de Hanoi

En la teoría de la complejidad computacional, los 21 problemas NP-completos de Karp son un conjunto de problemas computacionales. En su artículo de 1972, «Reductibilidad entre problemas combinatorios», Richard Karp usó el teorema de Stephen Cook de 1971 de que el problema de satisfactibilidad booleano es NP-completo (también llamado teorema de Cook-Levin para demostrar que hay una reducción de muchos-uno en el tiempo polinómico del problema de satisfactibilidad booleano a cada uno de los 21 problemas teóricos, combinatorios y gráficos, mostrando así que todos son NP-completos. Esta fue una de las primeras demostraciones de que muchos problemas computacionales naturales que ocurren en la ciencia de la computación son intratables desde el punto de vista computacional, y generó interés en el estudio de la completitud de **NP** y el problema **P vs. NP**.[22]

Los 21 problemas de Karp se muestran a continuación, muchos con sus nombres originales. El anidamiento indica la dirección de las reducciones utilizadas. Por ejemplo, se demostró que mochila es **NP-completo** al reducir la cobertura exacta a mochila.

22 https://hmong.es/wiki/Karp%27s_21_NP-complete_problems

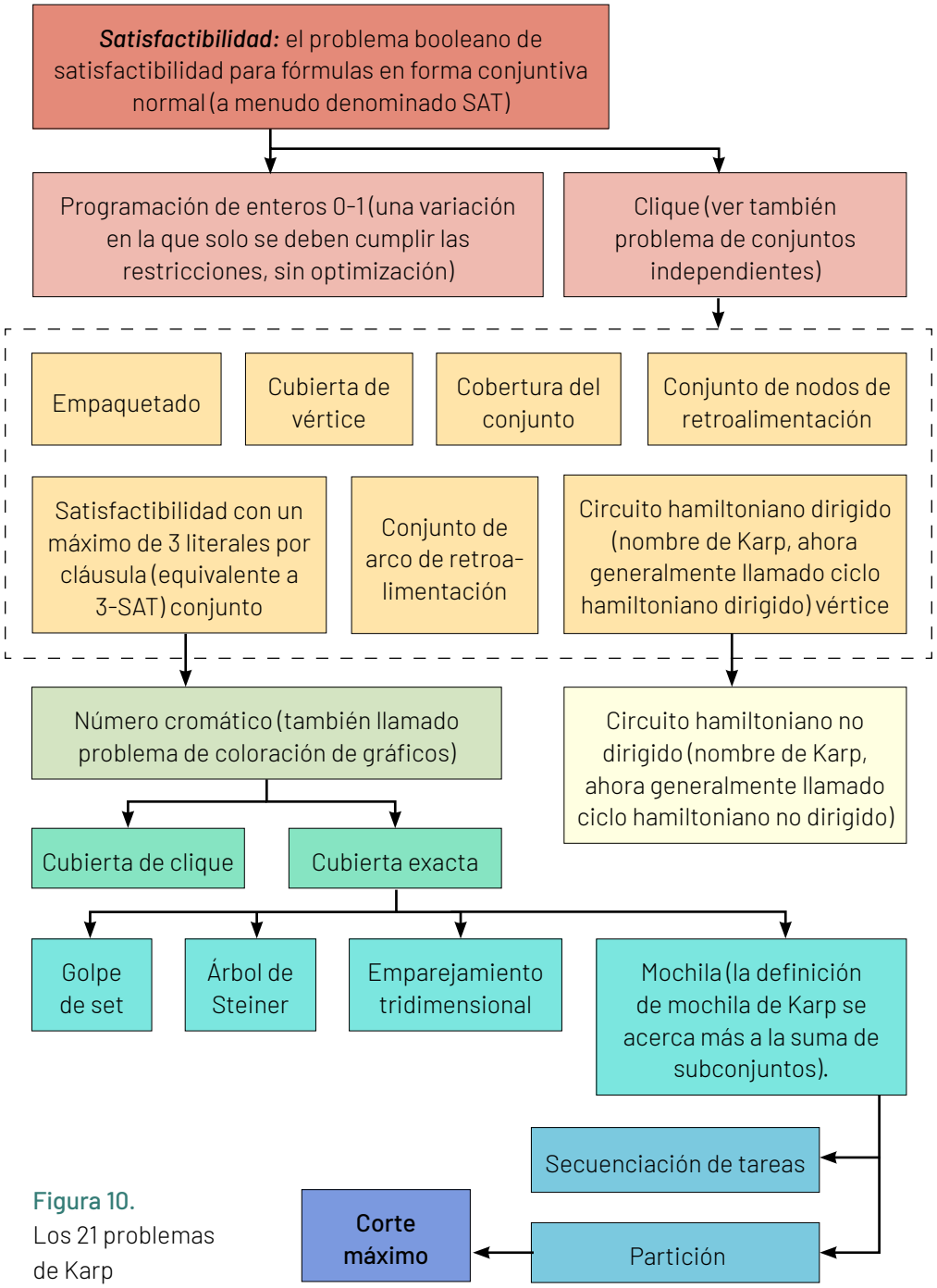


Figura 10.
Los 21 problemas de Karp

Con el paso del tiempo se descubrió que muchos de los problemas se pueden resolver de manera eficiente si se restringen a casos especiales, o se pueden resolver dentro de un porcentaje fijo del resultado óptimo.

Sin embargo, David Zuckerman mostró en 1996 que cada uno de estos 21 problemas tiene una versión de optimización restringida que es imposible de aproximar dentro de cualquier factor constante a menos que P = NP, al mostrar que el enfoque de reducción de Karp se generaliza a un tipo específico de reducción de aproximación. Sin embargo, tenga en cuenta que estos pueden ser diferentes de las versiones de optimización estándar de los problemas, que pueden tener algoritmos de aproximación (como en el caso del corte máximo).

En general las aplicaciones de problemas de optimización se pueden resumir en la siguiente lista:

* Logística
* Optimización de la cadena de suministro
* Desarrollar la mejor red de aerolíneas de radios y destinos
* Decidir a qué taxis de una flota enrutar para recoger las tarifas
* Determinar la forma óptima de entregar paquetes
* Asignar puestos de trabajo a las personas de manera óptima
* Diseño de redes de distribución de agua
* Problemas de ciencias de la tierra (p. ej., caudales de embalses)

Esta es una lista dinámica y es posible que nunca pueda satisfacer los estándares particulares de exhaustividad. Puede ayudar agregando elementos faltantes con fuentes confiables.

10 Conclusiones

Hemos visto la importancia del manejo de datos para resolver un problema, así como el diseño y el análisis de algoritmos para diferentes áreas de la Investigación de Operaciones, si bien el tema no está tratado de manera exhaustiva se busca que el lector (lectora) se familiarice con estos conceptos y pueda identificarlos en la literatura y aplicarlos a los problemas que se le presenten en las asignaturas de optimización y en las aplicaciones de estas a las diferentes áreas de la ingeniería.

Para resolver problemas de optimización y en general problemas de la Investigación de Operaciones se siguen los siguientes pasos con base en el método científico:

1. ***Formulación del problema, análisis y delimitación.*** La especificación de los objetivos es una de las tareas más importantes en un proyecto de I. de O. Todas las actividades de modelado y análisis deben basarse en los objetivos. Si estos no son claros o son poco concretos, existe el

peligro de no enfocar el problema correctamente y ser incapaces de responder a las expectativas generadas.

2. ***Construcción del modelo del problema.*** En esta fase, el investigador de operaciones debe decidir el modelo por utilizar para representar el sistema. Debe ser un modelo tal que relacione a las variables de decisión con los parámetros y restricciones del sistema. Los parámetros (o cantidades conocidas) se pueden obtener, ya sea a partir de datos pasados o estimados por medio de algún método estadístico. Es recomendable determinar si el modelo es probabilístico o determinístico.

3. ***Selección del método de solución y/o algoritmo.*** El modelo puede ser matemático, de simulación o heurístico, dependiendo de la complejidad de los cálculos matemáticos que se requieran y es en este punto donde es importante comparar los diferentes métodos, algoritmos y cuáles se reporta que son más eficientes para resolver el problema en cuestión, y donde lo que se ha desarrollado en este escrito cobra importancia.

4. ***Solución del modelo.*** Debemos tener en cuenta que las soluciones que se obtienen en este punto del proceso son matemáticas y debemos interpretarlas en el mundo real. Además, para la solución del modelo, se deben realizar análisis de sensibilidad, es decir, ver cómo se comporta el modelo a cambios en las especificaciones y parámetros del sistema. Esto se hace debido a que los parámetros no necesariamente son precisos y las restricciones pueden estar equivocadas.

5. ***Validación del modelo.*** La validación de un modelo requiere que se determine si dicho modelo puede predecir con certeza el comportamiento del sistema. Se puede hacer uso de datos históricos o bien si

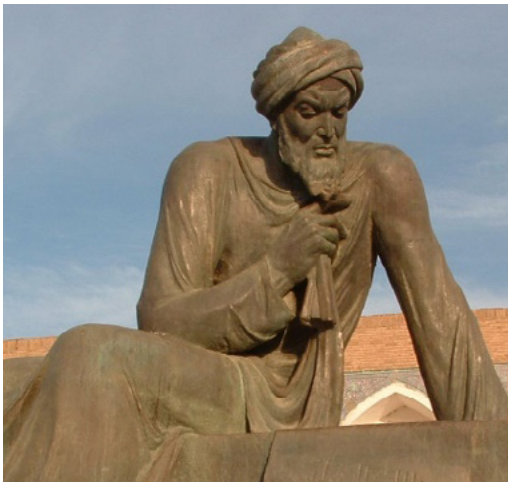
no se dispone de ellos o hay cambios en el sistema se usa algún otro modelo.

6. ***Implementación de los resultados.*** Es importante interpretar y verificar los resultados, las acciones a tomar y si el modelo puede servir para resolver otro problema, entonces se debe documentar y actualizar el modelo para nuevas aplicaciones.

De acuerdo con estos pasos, el objetivo de este escrito es contribuir en la teoría de análisis de algoritmos y problemas de optimización en el paso 3, y esperando que con esto haya mayor claridad en la lectura de artículos relativos a los temas que se abordan en los cursos de Investigación de Operaciones.

11 Notas históricas

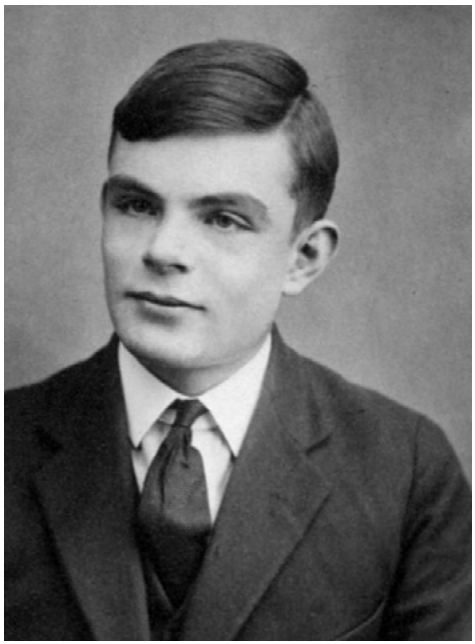
En algún lugar, entre 400 y 300 a. C., el gran matemático griego Euclides inventó un algoritmo para encontrar el máximo común divisor (m.c.d.) entre dos números naturales. El m.c.d. de X y Y es el mayor entero que divide a ambos. Por ejemplo, el m.c.d. de 80 y 32 es 16. Los detalles del algoritmo no nos interesan por el momento, pero el algoritmo de Euclides se considera el primer algoritmo no trivial desarrollado.



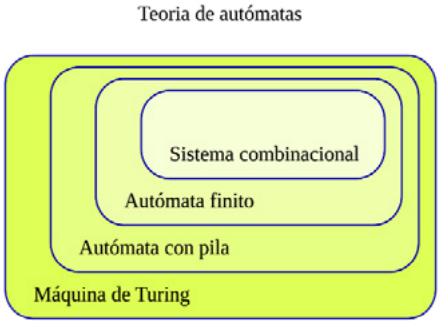
La palabra **algoritmo** se deriva del nombre del matemático persa Mohammed Al-Khowârizmî quien vivió durante el siglo noveno y quién tiene el mérito de haber elaborado paso a paso reglas para sumar, restar, multiplicar y dividir números decimales ordinarios. El nombre de este matemático en latín se convirtió en Algorismus, de donde declinó en algoritmo. Claramente Euclides y al- Khowârizmî fueron algorítmicos por excelencia.

(Alan Mathison Turing; Londres, 1912 - Wilmslow, Reino Unido, 1954). Matemático británico. Pasó sus primeros trece años en la India, donde su padre trabajaba en la administración colonial. De regreso al Reino Unido, estudió en el King's College y, tras su graduación, se trasladó a la Universidad estadounidense de Princeton, donde trabajó con el lógico Alonzo Church.

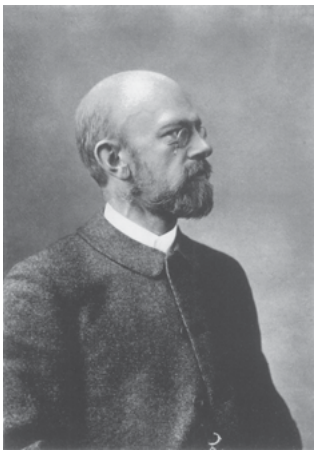
En 1937 publicó un célebre artículo en el que definió una máquina calculadora de capacidad infinita (máquina de Turing) que operaba basándose en una serie de instrucciones lógicas, sentando así las bases del concepto moderno de algoritmo. Turing describió en términos matemáticos precisos cómo un sistema automático con reglas extremadamente simples podía efectuar toda clase de operaciones matemáticas expresadas en un lenguaje formal determinado.



La máquina de Turing era tanto un ejemplo de su teoría de computación como una prueba de que un cierto tipo de máquina computadora podía ser construida.[23]



La importancia de la máquina de Turing en la historia de la computación es doble: primero, porque fue uno de los primeros (si no el primero) modelos teóricos para las computadoras, que vio la luz en 1936. Segundo, estudiando sus propiedades abstractas, la máquina de Turing ha servido de base para mucho desarrollo teórico en las ciencias de la computación y en la teoría de la complejidad. Una razón para esto es que las máquinas de Turing son simples, y por tanto amenas al análisis. Dicho esto, cabe aclarar que las máquinas de Turing no son un modelo práctico para la computación en máquinas reales, las cuales precisan modelos más rápidos como los basados en RAM.



David Hilbert (Königsberg, Prusia Oriental; 23 de enero de 1862-Gotinga, Alemania; 14 de febrero de 1943) fue un matemático alemán, reconocido como uno de los más influyentes del siglo XIX y principios del XX.

Estableció su reputación como gran matemático y científico inventando y/o desarrollando un gran abanico de ideas, como la teoría de invariantes, la axiomatización de la geometría

23 https://es.wikipedia.org/wiki/M%C3%A1quina_de_Turing

y la noción de espacio de Hilbert, uno de los fundamentos del análisis funcional.

Hilbert y sus estudiantes proporcionaron partes significativas de la infraestructura matemática necesaria para la mecánica cuántica y la relatividad general. Fue uno de los fundadores de la teoría de la demostración, la lógica matemática y la distinción entre matemática y metamatemática.

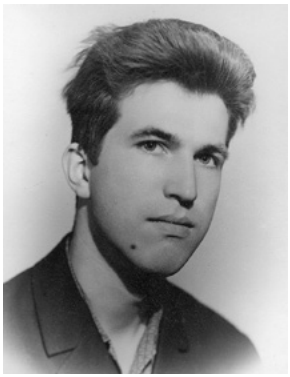
Adoptó y defendió vivamente la teoría de conjuntos y los números transfinitos de Cantor. Un ejemplo famoso de su liderazgo mundial en la matemática es su presentación en 1900 de un conjunto de problemas abiertos que incidió en el curso de gran parte de la investigación matemática del siglo XX.

El décimo problema de Hilbert es uno de los conocidos como veintitrés Problemas de Hilbert, publicados en 1900 por el matemático alemán David Hilbert. Su enunciado original es:

> *Dada una ecuación diofántica con cualquier número de incógnitas y con coeficientes numéricos racionales enteros:*
>
> *Idear un proceso de acuerdo con el cual pueda determinarse, en un número finito de operaciones, si la ecuación es resoluble en números racionales enteros.*

En términos de programación informática, Hilbert solicitaba a sus colegas del futuro un algoritmo capaz de admitir como entrada (input) una ecuación diofántica cualquiera, y de devolver SÍ como resultado (output)

si la ecuación procesada tenía soluciones en números enteros o NO si la ecuación[24] procesada carecía de soluciones en números enteros.



El problema no se resolvió hasta 70 años después, y en sentido negativo. En 1970 Yuri Matiyasévich culminó más de veinte años de trabajo de varios matemáticos, entre ellos Martin Davis, Julia Robinson y Hilary Putnam, con la demostración de imposibilidad del décimo problema: ningún algoritmo es capaz de determinar la resolubilidad de cualquier ecuación diofántica. El planteamiento, desarrollo y demostración del problema tienen gran interés en matemática moderna, porque en ellos participan conceptos de teoría de números y de lógica matemática, y se abren nuevos campos de investigación en ambas disciplinas.

El estudio del tiempo de ejecución de los programas y la complejidad computacional de problemas fue iniciado por Hartmanis y Stearns [1964].

Richard Edwin Stearns (nacido el 5 de julio de 1936 en Estados Unidos) y Juris Hartmanis (nacido el 5 de julio de 1928 en Riga, Letonia) son prominentes científicos de la computación.

24 https://es.wikipedia.org/wiki/D%C3%A9cimo_problema_de_Hilbert

Donald Ervin Knuth (Milwaukee, Wisconsin; 10 de enero 1938) es un reconocido experto en ciencias de la computación estadounidense y matemático, famoso por su fructífera investigación dentro del análisis de algoritmos y compiladores.

Es Profesor Emérito de la Universidad de Stanford.

En este escrito, nos hemos concentrado en los límites superiores para los tiempos de ejecución de programas Knuth [1976] describe notaciones análogas para límites inferiores y límites en los tiempos de ejecución.

Desde entonces, se ha desarrollado una rica teoría sobre la dificultad de los problemas. Muchas de las ideas clave se encuentran en Aho, Hopcroft y Ullman [1974, 1983].

Richard Karp nació en Boston, Massachusetts. Recibió su licenciatura por la Universidad de Harvard en 1955, su máster en 1956, y su Ph.D. en matemática aplicada en 1959. A partir de entonces trabajó en el Thomas J. Watson Research Center de IBM. En 1968 ingresó como profesor de Ciencias de la Computación, Matemáticas e Investigaciones Operacionales de la Universidad de California, Berkeley. Aparte de un periodo de 4 años en el que fue profesor en la Universidad de Washington, ha permanecido en Berkeley. Karp también ganó la Medalla

Benjamin Franklin de 2004 en Ciencias de la Computación y Cognitivas por sus contribuciones al campo de la complejidad computacional. La razón por la que se le otorgó el Premio Turing fue:

> *Por sus continuas contribuciones a la teoría de algoritmos, incluyendo el desarrollo de algoritmos eficientes para el flujo de redes y otros problemas de optimización combinatoria, la demostración de equivalencia de la noción intuitiva de eficiencia logarítmica con la computabilidad en tiempo polinómico y, principalmente, sus contribuciones a la teoría de NP-completitud. Karp la metodología hoy común para probar que ciertos problemas son NP-completos que ha llevado a determinar que muchos problemas teóricos y prácticos son computacionalmente difíciles.*

En 1971 codesarrolló junto con Jack Edmonds el algoritmo de Edmonds-Karp para resolver problemas de maximización de flujo en redes. En 1972 publicó su famosa lista de 21 problemas NP-completos. En 1987, junto con Michael O. Rabin desarrolló el algoritmo Rabin-Karp de búsqueda de cadenas.

Ha hecho muchos otros importantes descubrimientos en las ciencias de la computación e investigación operacional, en el área de optimización combinatoria. En 2006, cuando se escribió este artículo, su principal interés incluye la bioinformática.

12 Bibliografía

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. (1983). *Data Structures and Algorithms,* Addison-Wesley, Reading, Mass.

J. Dréo, A. Petroski, P. Siarry, E. Taillard. (2006). *Metaheuristics for Hard Optimization Methods and Case Studies Simulated Annealing, Tabu Search, Evolutionary and Genetic Algorithms, Ant Colonies*. Springer.

Flores De La Mota, I. (2002). *Apuntes de Programación Entera*, Facultad de Ingeniería UNAM.

Garey, M.R., Johnson, D.S. (1979). *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Bell Telephone Laboratories Incorporated.

Goldreich, O. (2010). *P, NP, and NP-Completeness: The Basics of Computational Complexity*. Cambridge University Press.

Lee, R.C.T., Tseng, S.S., Chang, R.C., Tsai, Y.T. (2007). *Introducción al diseño y análisis de algoritmos,* Ed. Mc. Graw-Hill.

López, G., Jeder, I., Vega, A. (2009). *Análisis y Diseño de Algoritmos*. Ed. Alfaomega.

Moret, B.M.E., Shapiro, H.D. (1991). *Algorithms from P to NP*. The Benjamin / Cummings Publishing Company, Inc.

WEBGRAFÍA

http://lwh.free.fr/pages/algo/tri/tri_insertion_es.html
http://lwh.free.fr/pages/algo/tri/tri_bulle_es.html
http://lwh.free.fr/pages/algo/tri/tri_shaker_es.html
http://lwh.free.fr/pages/algo/tri/tri_gnome_es.html
https://www.youtube.com/watch?v = aXXWXz5rF64
https://programas.cuaed.unam.mx/repositorio/moodle/pluginfile.php/1196/mod_resource/content/1/contenido/index.html
https://plataforma.josedomingo.org/pledin/cursos/programacion/curso/u03/
https://www.u-cursos.cl/ingenieria/2008/2/IN34A/3/material_docente/bajar?id = 185136
https://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_complejidad_computacional
https://www.youtube.com/watch?v = 92WHN-pAFCs
http://hi.baidu.com/nuclearspace/item/e0f8a1b777914974254b09f4
https://es.acervolima.com/diferencia-entre-problema-np-duro-y-np-completo/
https://www.xatakaciencia.com/computabilidad/problema-de-satisfacibilidad-sat
https://en.wikipedia.org/wiki/Boolean_satisfiability_problem
https://jcc.dcc.fceia.unr.edu.ar/2016/slides/2016-paredes-restrepo.pdf
https://hmong.es/wiki/Karp%27s_21_NP-complete_problems
https://es.wikipedia.org/wiki/M%C3%A1quina_de_Turing
https://es.wikipedia.org/wiki/D%C3%A9cimo_problema_de_Hilbert

https://es.wikipedia.org/wiki/Torres_de_Han%C3%B3i
https://es.wikipedia.org/wiki/Azulejos_Wang
https://es.wikipedia.org/wiki/Teselaci%C3%B3n_de_Penrose
https://es.wikipedia.org/wiki/Problema_del_caballo
https://www.bbc.com/mundo/noticias-60337860
file:///C:/Users/IDO/Downloads/Documat-LaDificultadDeJugarSudoku-6981033.pdf

13 Anexo

JUEGOS Y PROBLEMAS DE INTERÉS

INTRATABLE. Las torres de Hanói[25]

Las torres de Hanói es un rompecabezas o juego matemático inventado en 1883 por el matemático francés Édouard Lucas. Este juego de mesa individual consiste en un número de discos perforados de radio creciente que se apilan insertándose en uno de los tres postes fijados a un tablero. El objetivo del juego es trasladar la pila a otro de los postes siguiendo ciertas reglas, como que no se puede colocar un disco más grande encima de un disco más pequeño. La fórmula para encontrar el número de movimientos necesarios para transferir n discos desde un poste a otro es: 2^{n-1}

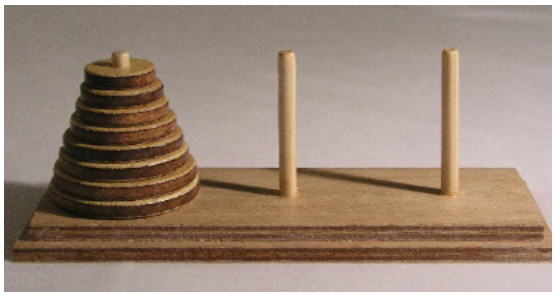


Figura A.1 Las torres de Hanói

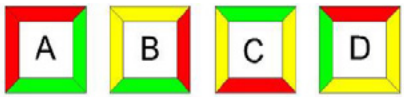
25 https://es.wikipedia.org/wiki/Torres_de_Han%C3%B3i

INDECIDIBLE. Azulejos Wang (o Dominó Wang)[26]

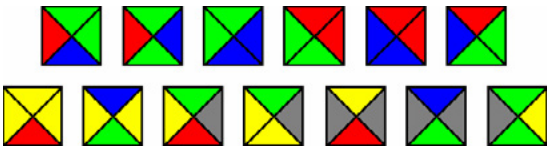
Primero propuestos por el matemático, lógico y filósofo Hao Wang en 1961, es una clase de sistemas formales. Son modelados visualmente por azulejos cuadrados con un color en cada lado. Un conjunto de tales azulejos está seleccionado, y las copias de los azulejos son puestas lado a lado con colores iguales, sin rotarlos ni reflejándolos.

La cuestión básica sobre un conjunto de azulejos de Wang es si puede enladrillar el plano o no, por ejemplo, si un plano infinito entero puede ser llenado de este modo. La siguiente pregunta es si esto puede ser hecho en un patrón periódico. ¿Puede embaldosarse con ellos el plano, de forma que los lados contiguos tengan el mismo color? (Se pueden hacer tantas copias como se quiera, no se pueden girar ni invertir).

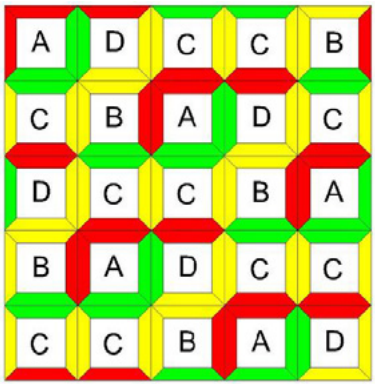
Ejemplos fáciles: periódicos. Datos:



Ejemplos difíciles: aperiódicos. Datos:



Embaldosado ampliable



Respuesta: Se puede embaldosar el plano de forma no periódica

Figura A.2 Azulejos Wang

26 https://es.wikipedia.org/wiki/Azulejos_Wang

INDECIDIBLE. Un mosaico de Penrose [27]

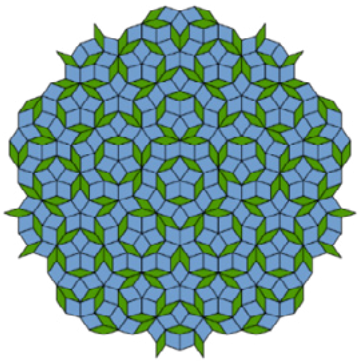


Figura A.3 Mosaico de Penrose

Es un ejemplo de un mosaico aperiódico, razón por la que es indecidible. Aquí, un mosaico es una cubierta del plano por polígonos que no se superponen u otras formas, y aperiódico significa que el desplazamiento de cualquier mosaico con estas formas en cualquier distancia finita, sin rotación, no puede producir el mismo mosaico. Sin embargo, a pesar de su falta de simetría de traslación, los mosaicos de Penrose pueden tener simetría de reflexión y simetría rotacional quíntuple. Los mosaicos de Penrose llevan el nombre del matemático y físico Roger Penrose, quien los investigó en la década de 1970.

NP-COMPLETO. El juego del caballo[28]

Se tiene un tablero de $n \times n$ con n^2 campos. Un caballo (a quien se le permite moverse conforme a las reglas de ajedrez) se pone en el campo con las coordenadas iniciales x_0, y_0. El problema radica en encontrar una cobertura de todo el tablero (si es que existe), o sea calcular un circuito de $n^2 - 1$ movimientos (jugadas) tales que cada campo del tablero sea visitado exactamente una vez. La manera obvia de reducir el problema es abarcar n^2 campos y consiste en considerar el problema realizando una jugada siguiente o descubriendo que ninguna es posible.

27 https://es.wikipedia.org/wiki/Teselaci%C3%B3n_de_Penrose
28 https://es.wikipedia.org/wiki/Problema_del_caballo

Se tiene noticia de que Euler (1759) y Vandermonde (1771) discutieron el problema del circuito de un caballo. Este problema es un circuito hamiltoniano en una gráfica cuyos vértices son los 64 cuadros de un ajedrez, con dos vértices adyacentes si y solo si el caballo se puede mover en un paso de un cuadrado al otro. Euler diseñó una solución donde cada fila suma 260. Al detenerse en la mitad de cada una resulta 130.

1	48	31	50	33	16	63	18
30	51	46	3	62	19	14	35
47	2	49	32	15	34	17	64
52	29	4	45	20	61	36	13
5	44	25	56	9	40	21	60
28	53	8	41	24	57	12	37
43	6	55	26	39	10	59	22
54	27	42	7	58	23	38	11

Figura A.4 El juego del caballo

Los capítulos de la novela *La vida instrucciones de uso* (1978) de Georges Perec siguen una ordenación que corresponde a una solución del problema del caballo sobre una cuadrícula de 10×10. La solución fue encontrada experimentalmente por el mismo autor.

NP-COMPLETO. Problema de las ocho reinas[29]

El problema de las ocho reinas es un pasatiempo que consiste en poner ocho reinas en el tablero de ajedrez sin que se amenacen. Fue propuesto por el ajedrecista alemán Max Bezzel en 1848. En el juego del ajedrez la reina amenaza a aquellas piezas que se encuentren en su misma fila, columna o diagonal. El juego de las 8 reinas consiste en poner sobre un tablero de ajedrez ocho reinas sin que estas se amenacen entre ellas. Para resolver este problema se puede emplear un esquema vuelta atrás (o *Backtracking*).

29 https://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas

El problema de las ocho reinas se puede plantear de modo general como problema de las *n* reinas. El problema consistiría en colocar *n* reinas en un tablero de ajedrez de tal manera que ninguna de las reinas quede atacando a otras. Su análisis y solución es isomorfo al de las ocho reinas. El tablero de 27 × 27 es el más grande hasta ahora numerado.

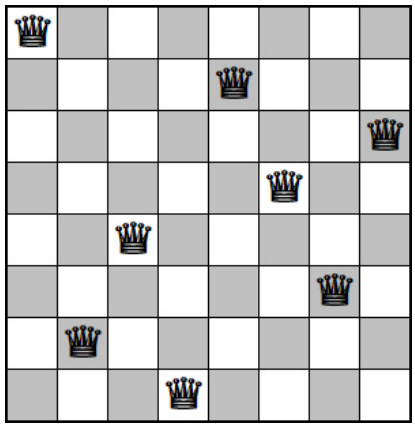


Figura A.5 El problema de las 8 reinas

El 21 de enero de 2022, *The Harvard Gazette*, el órgano de prensa oficial de la Universidad de Harvard, informó que uno de sus matemáticos, Michael Simkin, había resuelto «en gran medida un problema de ajedrez de 150 años».

Simkin calculó que para tableros de ajedrez enormes (*n* por *n* casillas) y con muchas reinas, hay alrededor de $(0{,}143n)^n$ maneras de colocar las reinas sin que ninguna se amenace:

«Supongamos que queremos saber cuántas configuraciones para 1.000.000 de reinas hay», indica el investigador. Es decir, queremos determinar el número de formas en que se pueden colocar 1.000.000 de reinas en un tablero de 1.000.000 x 1.000.000 (de casillas) sin que se ataquen entre sí. Para calcular ese número, que es una aproximación, debemos multiplicar 1.000.000 por 0,143 y el resultado, 143.000, lo elevamos a la potencia de 1.000.000. *«En otras palabras, multiplica 143.000 por sí mismo un millón de veces. El resultado es un número muy grande, con aproximadamente cinco millones de dígitos».*

¿Lo quieres ver con un número más pequeño? Tomemos el 1 000, Jesús Fernando Barbero, matemático e investigador científico del Consejo Superior de Investigaciones Científicas de España, hizo el cálculo usando la ecuación de Simkin.

Si queremos saber aproximadamente cuántas configuraciones hay para 1000 reinas, donde está la *n* ponemos 1000: 0,143 × 1000 = 143 y lo elevamos a 1.000.

El docente usó su computadora y este es el resultado que arrojó:


RESULTADO DE LA FÓRMULA DE SIMKIN PARA UN TABLERO 1000 × 1000 CON 1000 REINAS

2167891113116473423113406307190952568372622319850312016219797079674827894246033117211338
3443039117493960524710852535274553130531688785894566118567115644017021598756855567493900
2431292492979974575348808076523547237281146490915237440549074532101161128803128317473164
6151157109269817228650112037524870710218412801710937253447495223539751964125163583179483O
6467780329511120436150614341806980700590441237713610086420582185464773040527994275740981O
2332357242966842274055498681204430889779136103324503151097979334388402429812370014007942З
3593107100412030811539578874067151408549302963347111048137956687556802951860158991397154В
4081758469906033781100447933050158345207882087988526850260308359327478424998535208127259
5916665538976853468996117431902092299618561720897398605105359185157855305707232196422864
8352803308414309074363894419216409213058519608129733563447363496091480026355373881821508Э
6699303695755570716756588447177150888376510891760781523903532488836798516357467777880668З0
4788621152648622732559496344336893303971339098707577688203044398024950969983383294901411885
4273180341366441144834785056580270758061452021891353464041422013574608773780116558045750З
7198246519108982580729920744972685852888812717627875523461972892614086737614546702318603
0947078389735872895884241049651300641780308064548326795915893484235712320236370272945409б
8989388204655311012309419639716798410217037131462502421922594620543088910826316249347560Э
0215122954331723842595631352126835342220682439673565485561074229464342950547411029181457
0139904753054550174716876248821925688074585259217187876490293235798779583326017607306565Э
0978415381680647412575779715520269326132334942146182318140599238281927040954522344339131Ч
0552442209544718034676319116030284596963808212141423954009098665887022974191519206187816Э
7355264972957949078942705617329990486923759949861275639601477914673841018167758698344318О
3633683959244604594833051852685423748992021219288164212795037274231201312956185753418894
9490477007749938406773777569411798292517010946917795238623394034067283776802671559180921Э
3375947810446287670688477418256286675479016550557941823062979271033540829366220552042163З
9721866352822040000I

Figura A.6 El número de configuraciones de las mil reinas

«Me da un número enorme, de más de 2.000 dígitos, pero muy cercano al valor real del número de configuraciones que hay para un tablero de tamaño 1000 × 1000».

Con la ecuación final de Simkin se llega a un resultado aproximado, no es el número exacto de configuraciones, pero es la cifra más cercana al número real que se puede obtener hasta ahora.[30]

NP-COMPLETO. Sudoku[31]

Dado M un sudoku parcial de tamaño n^2 diremos que M es *completable* si y solo si existe un modo de reemplazar todas las ocurrencias del símbolo © por elementos del conjunto $\{1, \cdots, n^2\}$, de manera que tras realizar tales sustituciones se obtenga un sudoku.

Entrada: (M, n), donde M es un sudoku parcial de tamaño n^2.
Problema: Decida si M es completable. Es fácil verificar que el problema sudoku pertenece a **NP**. Sea $[n^2]$ el conjunto $\{1, \cdots, n^2\}$. Dado un sudoku parcial M de tamaño n^2, un completado parcial para M es una función f: $[n^2] \times [n^2] \to [n^2]$ tal que:

1. Si $M_{ij} \neq ©$, entonces $M_{ij} = f(i, j)$
2. La matriz $[f(i, j)]_{i,j \leq n^2}$ es un sudoku

Sea R la relación $\{(M, f)$: M es un sudoku parcial y f es un completado para $M\}$.

Es claro que R es una relación p-balanceada y también es claro que $(M, n) \in$ sudoku si y solo si existe f tal que $(M, f) \in R$.

30 https://www.bbc.com/mundo/noticias-60337860
31 file:///C:/Users/IDO/Downloads/Documat-LaDificultadDeJugarSudoku-6981033.pdf

Lo anterior implica que sudoku es un problema en **NP**, para el cual la noción de certificado es desempeñada por los completados.

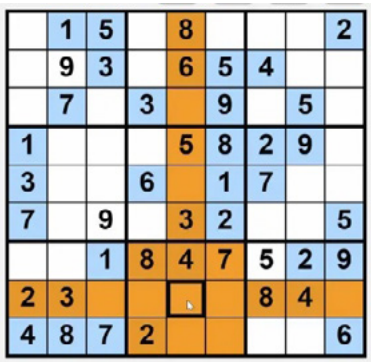


Figura A.7 Juego del sudoku

EL AGENTE VIAJERO Y LA TORRE DE HANOI

¿Cómo se relaciona este rompecabezas con el juego de Hamilton?

Para explicar la conexión debemos considerar primero una torre de tres discos solamente, etiquetando los discos, desde arriba hacia abajo A, B y C. Si seguimos el procedimiento mencionado para resolver el problema de la torre de Hanoi tendremos la solución moviendo los discos en el siguiente orden: ABACABA.

Llamemos ahora A, B y C a las tres coordenadas de un hexaedro regular, llamado comúnmente cubo, como se muestra en la figura siguiente:

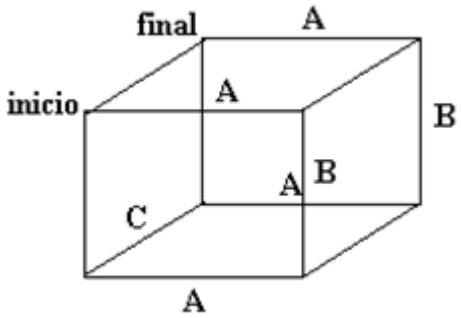


Figura A.8 Cubo que relaciona 3 discos con un recorrido hamiltoniano

Si trazamos una ruta a lo largo de los bordes del cubo eligiendo las coordenadas en el orden ABACABA la ruta formará un circuito hamiltoniano.

La generalización de esto la vio D. W. Crowe de la Universidad de Columbia Británica como sigue: el orden de la transferencia de *n* discos en la torre de Hanoi se corresponde exactamente con el orden de las coordenadas al trazar un camino hamiltoniano en un cubo de *n* dimensiones. Para el caso de $n = 4$ se proyecta la red de sus bordes en un modelo tridimensional como se muestra en la figura siguiente:

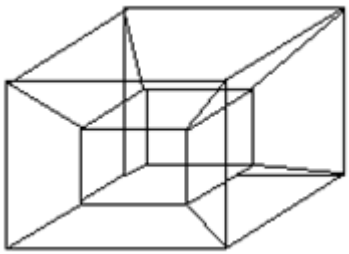


Figura A.9 Hipercubo para 4 discos

Este hipercubo tiene las coordenadas A, B, C y D, el camino que se sigue es ABACABADABACABA. Lo mismo se hace para un hipercubo de 5 dimensiones, 6 hasta *n*.

Análisis de algoritmos y optimización
se publicó la primera edición electrónica de un ejemplar (5 MB) en formato PDF el 29 de agosto de 2022, en el repositorio de la Facultad de Ingeniería, UNAM, Ciudad Universitaria, Ciudad de México. C.P. 04510

El diseño estuvo a cargo de la Unidad de Apoyo Editorial de la Facultad de Ingeniería. Las familias tipográficas utilizadas son Barlow para texto y Roboto Slab para títulos con sus respectivas variantes.