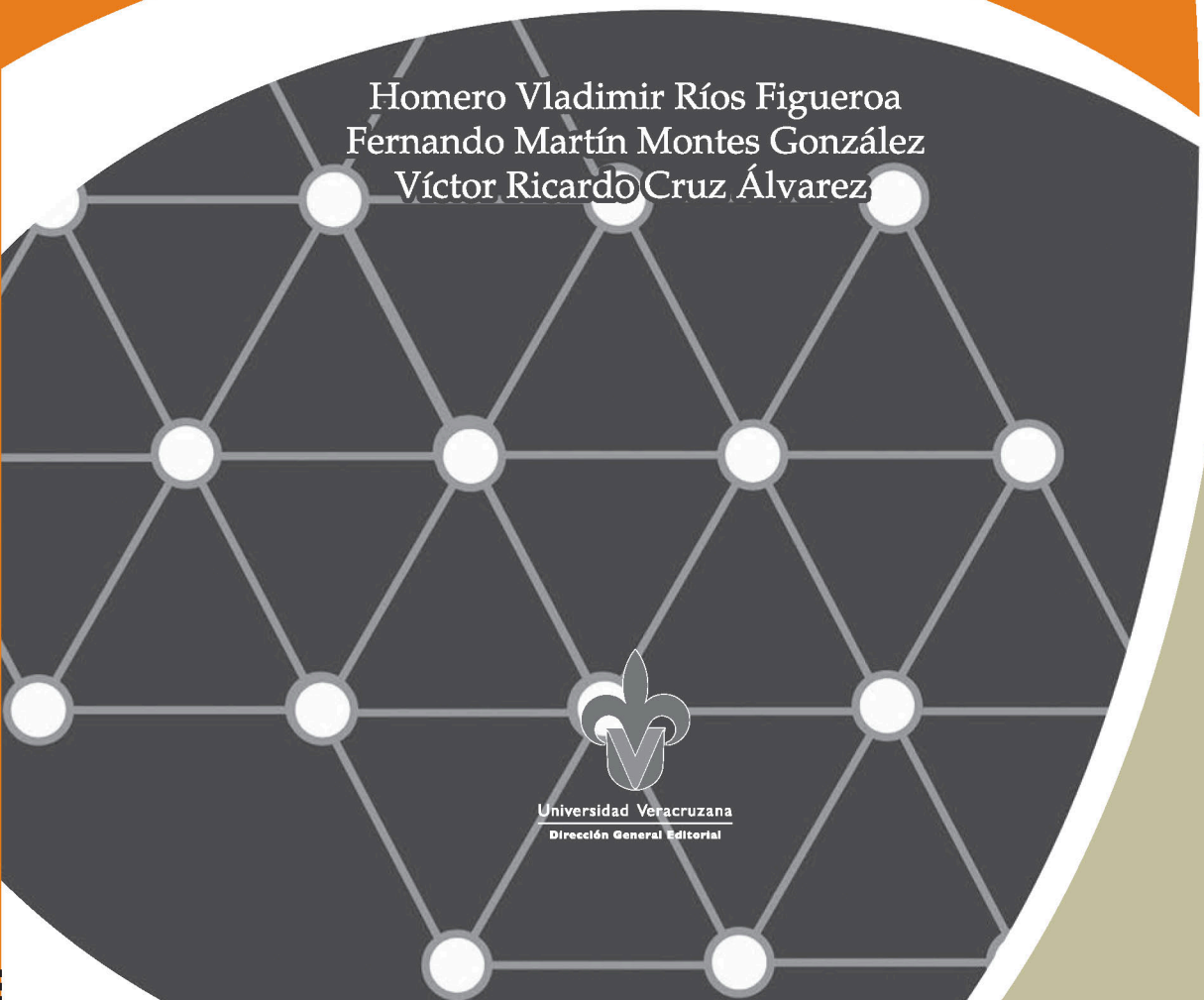
textos universitarios

Análisis de algoritmos

Homero Vladimir Ríos Figueroa
Fernando Martín Montes González
Víctor Ricardo Cruz Álvarez



Universidad Veracruzana
Dirección General Editorial

Análisis de algoritmos

Universidad Veracruzana

textos universitarios

Análisis de algoritmos

Homero Vladimir Ríos Figueroa
Fernando Martín Montes González
Víctor Ricardo Cruz Álvarez

Universidad Veracruzana
Dirección General Editorial
2013

Diseño de portada: Enriqueta del Rosario López Andrade, a partir de una imagen de Dafne Carmina Ríos Cázares.
Diseño de interiores: Pedro Gaspar

Clasificación LC:	QA9.58 R563 2013
Clasif. Dewey:	511.8
Autor personal:	Ríos Figueroa, Homero Vladimir
Título:	Análisis de algoritmos / Fernando Martín Montes González, Víctor Ricardo Cruz Álvarez.
Edición:	Primera edición.
Pie de imprenta:	Xalapa, Veracruz : Universidad Veracruzana, 2013.
Descripción física:	130 páginas : ilustraciones ; 23 cm.
Serie:	(Textos universitarios)
ISBN:	9786075022390
Materias:	Algoritmos. Algoritmos--Problemas, ejercicios, etc.
Autores secundarios:	Montes González. Fernando Martín, coautor. Cruz Álvarez, Víctor Ricardo, coautor.

DGBUV 2013/31

Primera edición, 19 de abril de 2013

Dirección General Editorial
Hidalgo 9, Centro, Xalapa, Veracruz
Apartado postal 97, CP 91000,
diredit@uv.mx
Tel/fax (228) 818 59 80; 818 13 88
Xalapa, Ver., 91000, México

ISBN: 978-607-502-239-0

Impreso en México
Printed in México

Agradecimientos

Agradezco el apoyo y ánimo de Dulce Elizabeth para la realización de esta obra. También el afecto incondicional de mis hijas, Dafne y Brenda. Adicionalmente deseo expresar mi agradecimiento a Nayeli Meléndez por su ayuda en la elaboración y mejora de las figuras del libro.

Homero Ríos

Agradezco a Homero Ríos por invitarme a participar en este proyecto y a Víctor Cruz por su valiosa participación.

Fernando Montes

Agradezco al Conacyt y a la Universidad Veracruzana por el apoyo proporcionado durante mis estudios de maestría.

Víctor Cruz

Prólogo

Esta obra está orientada para apoyar un curso de análisis de algoritmos a nivel maestría, en un periodo cuatrimestral. El contenido cubre ampliamente un curso de 40 horas.

El material expuesto se ha impartido en maestrías en Ciencias de la Computación y también en Inteligencia Artificial.

Los ejercicios propuestos permiten que los estudiantes amplíen los temas planteados en clase, al desarrollarlos y programarlos en computadora.

Para fortalecer el aspecto didáctico del libro, se ha incluido la respuesta a los ejercicios, y el código en Java de los algoritmos principales. Los programas se anexan en un medio magnético.

Consideramos que el texto también puede emplearse en cursos de algoritmos a nivel licenciatura, excluyendo los ejercicios más complejos y que llevan más tiempo en programar.

La diferencia con otros libros de algoritmos, es que éste se enfoca más al tema de gráficas, que consideramos de aplicación general y no tanto al problema específico de ordenamiento.

Además, en la mayor parte de los cursos de licenciatura se cubre considerablemente el tema de ordenamiento, por lo cual en esta obra se excluyó ese tema.

Otro aspecto diferente de este libro es que en menos tiempo se aborda el tema medular de complejidad de algoritmos, gráficas y límites al cómputo.

Finalmente, la inclusión de algoritmos selectos por categoría, permite que el estudiante se familiarice con ellos, siendo de amplia utilidad, por lo que este texto es un valioso apoyo debido a que incluye temas que en la mayoría de los cursos se omiten.

Introducción

Podemos considerar que los algoritmos son tan antiguos como la humanidad, ya que siempre que se sigue una secuencia de pasos para alcanzar un objetivo, se está aplicando un algoritmo. Los algoritmos están en todas partes, en el procedimiento para arreglar una llanta ponchada, para preparar un pastel, en la rutina que seguimos para arreglarnos y llegar al trabajo, en cómo se construye una pared, la manera en que se utiliza un teléfono celular, en fin, en prácticamente todo lo que hacen los seres humanos.

Aunque el origen de los algoritmos es muy antiguo y como hemos visto no se limita al campo de la ciencia y la tecnología, la identificación del concepto, como tal, sí es más reciente.

La palabra *algoritmo* fue introducida por al-Khwarizmi, autor del trabajo más antiguo conocido sobre álgebra. Él era un matemático de la primera mitad del siglo IX y uno de sus libros nos dio la palabra álgebra a partir de *al-Jabr*.

En el siglo XII, este trabajo y otros fueron traducidos del árabe al latín y el sistema numérico decimal, posicional, comenzó a difundirse en toda Europa. Aunque las fuentes escritas eran árabes, los números comenzaron a referirse como arábigos, no obstante los signos para los números habían sido adaptados de la India.

En los textos en latín de la Edad Media a menudo encontramos referencias a las ventajas del nuevo sistema posicional decimal, res-

pecto a las tablas tradicionales de conteo o métodos de ábaco. Los nuevos métodos se describieron como *algoritmos*. Así que el término se usó para describir procedimientos aritméticos particulares.

Durante el movimiento ilustrado, d'Alembert describió en un artículo en su enciclopedia, a un algoritmo como:

"Término árabe usado para la práctica del álgebra. También algunas veces se usa para referirse a la aritmética con dígitos. En general, se refiere al método y la notación para todos los tipos de cálculo"

La palabra algoritmo ha llegado a significar cualquier proceso de cálculo sistemático, esto es, un proceso que puede llevarse a cabo automáticamente.

En la actualidad, principalmente por la influencia de la computación, la idea de lo finito se ha incorporado en el significado de un algoritmo, como un elemento esencial.

La preocupación por lo finito surgió en un contexto general en el décimo problema de Hilbert, que planteó en el Segundo Congreso Internacional de Matemáticas en 1900: "dada una ecuación polinomial con coeficientes racionales arbitrarios, se busca un método por el cual pueda ser determinado, en un número finito de operaciones, si la ecuación tiene solución en los números racionales".

Derivado de este problema, surgió el concepto de un *procedimiento efectivo*, es decir, de uno que efectivamente llegara al resultado en un tiempo finito. El concepto se formalizó en los años treinta por los lógicos de la época.

La formalización de un procedimiento efectivo, o de un algoritmo, es fundamental para las tecnologías de la información y también para las bases de las matemáticas mismas. Esto fue necesario, ya que hay problemas para los cuales no existen algoritmos que puedan ser usados.

Los problemas matemáticos que llevaron a la formalización del concepto de algoritmo, se plantearon en el Congreso Internacional de Matemáticas de 1928. Hilbert formuló las siguientes preguntas fundamentales para las matemáticas:

1. ¿Son las matemáticas completas? ¿Cualquier enunciado matemático puede confirmarse como válido o inválido?

2. ¿Son consistentes? ¿Es imposible probar un enunciado y su contradicción?
3. ¿Son decidibles? ¿Existe un procedimiento mediante el cual pueda decidirse, sin necesidad de llevar a cabo una prueba, si un enunciado matemático es verdadero (no contradictorio)?

Para sorpresa de todos, en 1931, el matemático, Kurt Gödel, probó la incompletez de la aritmética, esto es, que existen algunas proposiciones para las cuales es imposible probar si son verdaderas o falsas. Para probar el teorema, Gödel introdujo la noción de una función recursiva, que tiene la propiedad de ser computable.

El trabajo de Gödel inspiró la investigación de Alonzo Church, Stephen Kleene, Alan Turing y Emil Post. Estos matemáticos mostraron, que la tercera pregunta planteada por Hilbert, no tenía una respuesta afirmativa. Es decir, existen problemas no decidibles. Estos son enunciados matemáticos para los cuales no existe un procedimiento, por el que pueda decidirse su verdad o falsedad. Para hacer esto, cada uno de ellos, definió un concepto de computabilidad, esto es, un concepto de algoritmo.

Posteriormente se ha mostrado, que los diferentes conceptos de computabilidad, formulados entre 1931 y 1936, son esencialmente equivalentes. Esto garantiza la coherencia de los diferentes resultados matemáticos obtenidos.

Organización del libro

El libro inicia con una referencia histórica de los antecedentes de los algoritmos a lo largo del tiempo y su uso en prácticamente cualquier actividad humana. Además, se destaca el valor que ha tenido el concepto de algoritmo desde un punto de vista pragmático y científico, estando en este último caso, relacionado inclusive, con las bases de las matemáticas.

A continuación en el capítulo 1, se revisan los conceptos de estructura de datos, tipo de datos abstractos y el concepto de algoritmo. Aquí se busca distinguir entre los conceptos de solución de un pro-

blema, algoritmo y programa, que aunque están relacionados, no son lo mismo, y en muchas ocasiones se confunden. Esta distinción permite establecer niveles de abstracción e inclusive aplicarla para el estudio de los sistemas de percepción de los seres vivos, usando un enfoque de inteligencia artificial.

El capítulo 2, revisa los conceptos matemáticos necesarios para definir y cuantificar la complejidad de algoritmos. En particular, se explica el concepto de conjuntos, recursión, sumatorias, recurrencias y algunos métodos de prueba matemática como inducción y reducción al absurdo.

El capítulo 3, presenta las ideas del peor de los casos, mejor de los casos, caso promedio y cómo cuantificar el esfuerzo computacional de un algoritmo. A partir de las cotas superiores, inferiores y orden, se determina el crecimiento del número de operaciones, en función del tamaño del conjunto de datos a procesar. De manera análoga pueden analizarse los algoritmos para determinar su crecimiento en almacenamiento, en función del número de datos a procesar. A partir de estas definiciones podemos definir categorías y clasificar los algoritmos por su orden de crecimiento, y también determinar si existen mejores algoritmos para un problema dado. Un ejercicio de valor práctico, se ejemplifica al calcular la mejora en tiempo de procesamiento, para un algoritmo dado y al cambiar el poder de cómputo disponible. Se concluye que la ganancia relativa depende directamente de la complejidad intrínseca del algoritmo y no tanto del poder de cómputo disponible. Esto nos lleva a que siempre es mejor un algoritmo más eficiente que disponer de más recursos computacionales.

El capítulo 4, define el concepto de gráfica desde el punto de vista matemático y algunas de las estructuras de datos para representarlas. Se optó por dar prioridad a las gráficas porque sirven de base para resolver problemas complejos y por su generalidad. Se presentan recorridos clásicos de gráficas y algunos problemas interesantes y de valor práctico, como el de los prerrequisitos, el encontrar el camino más corto desde una sola fuente o iniciando desde cualquier nodo. También se enuncia y se presenta una solución de referencia para el problema del vendedor viajero. Finalmente se enuncia y se presenta

una solución al problema de generar un árbol expansor de mínimo costo para una gráfica.

El capítulo 5, introduce el concepto de reducción lo cual nos permite hacer clases de equivalencia entre algoritmos de acuerdo a su complejidad. También se definen los problemas duros y los NP. Esto sirve para ubicar a los problemas más complejos y aquellos que tienen una solución polinomial en una computadora no determinista, pero que aún no se encuentra una solución polinomial en una computadora determinista. Finalmente, para mostrar que no todo problema tiene una solución algorítmica, se ilustra el problema de la terminación.

El capítulo 6 inicia dando diferentes clasificaciones de los algoritmos, ya sea por su complejidad, por el tipo de problema o bien por la técnica de diseño empleada en su concepción. Continúa el capítulo mostrando algoritmos selectos por categoría de aplicación, como son, correspondencia de patrones, compresión y codificación, geometría computacional, análisis numérico, generación de objetos combinatorios, exploración inteligente de gráficas, algoritmos de aproximación para problemas NP y criptografía.

Para darle un mayor valor didáctico a la obra, se consideran ejercicios para cada capítulo, incluyendo su solución en una sección al final del libro. Otro aspecto para realzar el aprendizaje de los temas es, que contiene código y pseudocódigo en el texto. Adicionalmente, estamos incluyendo en un *medio magnético* que acompaña a la obra, programas en Java, con su documentación y ejemplos, para que puedan ser fácilmente compilados, ejecutados y probados.

Para cada programa se adjunta un archivo de documentación, además de los comentarios.

Los programas incluidos en el medio magnético son los siguientes:

- Quicksort, para ilustrar el orden de crecimiento $n \log n$, de algoritmos y una cota del mismo (capítulo 3).
- Dijkstra, solución al problema de caminos más cortos desde una fuente (capítulo 4).
- Floyd, solución al problema de caminos más cortos desde cualquier nodo de la gráfica (capítulo 4).

- Ordenamiento topológico, solución al problema de los prerrequisitos (capítulo 4).
- Generación de la secuencia de Collatz, para ilustrar que aún en algoritmos muy sencillos no es obvio definir si termina o no, para todas sus entradas (capítulo 5).
- Codificación del algoritmo de Horspool, para correspondencia de patrones (capítulo 6).
- Codificación y compresión de Huffman de acuerdo a la frecuencia de ocurrencia de los símbolos en un alfabeto (capítulo 6).

1. Estructuras de datos y algoritmos

Una filosofía de estructura de datos

La necesidad de estructura de datos

La representación de la información es fundamental para las Ciencias de la Computación. Para que un programa sea práctico en términos de requerimientos de almacenamiento y tiempo de ejecución, debe organizar sus datos en una forma que apoye el procesamiento eficiente.

Una *estructura de datos* es cualquier representación de datos y sus operaciones asociadas (por ejemplo, una representación de números enteros o flotantes y sus operaciones).

Más típicamente, se piensa en una estructura de datos como una organización o estructuración para una colección de datos (por ejemplo, un arreglo de enteros).

Dado suficiente espacio, casi siempre es posible realizar todas las operaciones necesarias en cualquier estructura de datos. Sin embargo, la elección de la estructura de datos puede hacer la diferencia entre un programa ejecutado en unos segundos, o en varios días.

Siempre que se diseña un programa para resolver un problema, hay que considerar cuáles son las metas de desempeño, para seleccionar la estructura de datos apropiada.

Una solución se dice que es *eficiente*, si resuelve el problema dentro de las restricciones de recursos requeridas.

El *costo* de una solución es la cantidad de recursos que una solución consume.

Metodología para seleccionar una estructura de datos para resolver un problema

1. Analice su problema para determinar las restricciones de recursos que cualquier solución debe satisfacer.
2. Determine las operaciones básicas que deben ser soportadas y cuantifique las restricciones de recursos para cada operación. (Ejemplos: Insertar o borrar un dato en la estructura de datos, buscar un elemento, etcétera).
3. Seleccione la estructura de datos que mejor cumple estos requerimientos.

Costo y beneficio

Cada estructura de datos tiene asociados costos y beneficios, generalmente es incorrecto decir que una estructura de datos es mejor que otra en todos los casos. Una estructura de datos requiere una cierta cantidad de espacio para cada dato que almacena, una cantidad de tiempo para realizar una operación básica y un esfuerzo de programación.

Ejemplo: En una base de datos para el manejo de cuentas en un banco, las operaciones de alta y baja de clientes, son en general más lentas, que las de retiros y depósitos.

Tipo de datos abstractos y estructura de datos

Un *tipo* es una colección de valores. Por ejemplo, el tipo Booleano consiste de los valores *verdadero* y *falso*.

Un *tipo de datos* es un tipo junto con una colección de operaciones para manipular el tipo. Por ejemplo, sobre el conjunto de los números enteros tenemos definidas las operaciones de suma, resta y multiplicación.

Un *dato elemental* es un elemento de información cuyo valor es tomado de un tipo. Se dice que un dato elemental es un miembro de un tipo de datos.

Ejemplo: Un entero es un *dato elemental* porque no contiene subpartes.

Ejemplo: Un registro de cuenta de banco puede contener varios elementos de información tal como nombre, dirección, número de cuenta, balance. Tal registro es un ejemplo de un *dato agregado*.

Existe una distinción importante entre la *descripción* de un tipo de datos y su *implementación* en un programa de cómputo. Por ejemplo, el tipo de datos lista puede ser implementado con una lista ligada o con arreglos.

Un *tipo de datos abstracto* (TDA) define un tipo de datos solamente en términos de un tipo y un conjunto de operaciones en ese tipo. La definición de un TDA no especifica cómo el tipo de datos es implementado.

El proceso de ocultar los detalles de implantación es conocido como *encapsulación*.

Una *estructura de datos* es la implementación de un TDA. Cada operación asociada con un TDA es implantada con una o más subrutinas (Figura 1). El término *estructura de datos* a menudo se refiere a datos almacenados en la memoria principal de la computadora. El término relacionado *estructura de archivos*, a menudo se refiere a la organización de datos en almacenamiento periférico.

Ejemplo: La actividad de conducir un auto puede ser visto como un TDA con operaciones *dar dirección*, *acelerar* y *frenar*. Dos autos pueden implementar estas operaciones en formas radicalmente diferentes. Sin embargo, la mayoría de los conductores pueden operar cualquier auto ya que el TDA presenta un método uniforme de operación.

Los datos tienen una *forma física* y *lógica*. La definición de un dato por un TDA es su forma lógica. La implementación del dato dentro de una estructura de datos es su forma física.

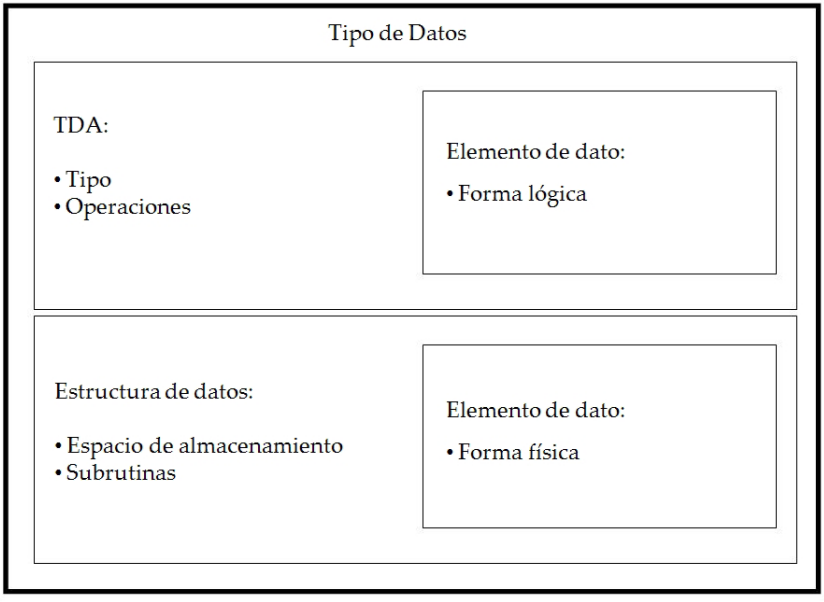


Figura 1. Relación entre tipo de datos abstractos y estructura de datos.

Problemas, algoritmos y programas

Un *problema* es simplemente una tarea a realizar o resolver. Una definición de problema no debe incluir restricciones en cómo debe resolverse. Sin embargo, una definición debe incluir restricciones sobre los recursos que pueden ser consumidos por cualquier solución aceptable.

Los problemas pueden ser vistos como funciones en un sentido matemático:

Una *función* es un mapeo entre las entradas (el dominio) y las salidas (el rango). La entrada a una función puede ser un solo valor o una colección de información.

- Los valores que forman la entrada se llaman *parámetros de la función*. Cada entrada particular debe resultar siempre en la misma salida, cada vez que la función es calculada.

Un *algoritmo* es un método o proceso seguido para resolver un problema. Si el problema es visto como una función, entonces el algoritmo toma una entrada y la transforma en la salida. Un problema puede tener muchos algoritmos.

Ejemplo: Un problema de ordenamiento:

Entrada: una secuencia n de números $(a_1, a_2 \ldots, a_n)$

Salida*:* una permutación (reordenamiento) $(a'_1, a'_2 \ldots, a'_n)$ de la entrada tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Dada una secuencia tal como (54, 37, 91, 21, 18, 7) el algoritmo de ordenamiento regresa como salida la secuencia (7, 18, 21, 37, 54, 91). A la entrada se le denomina una instancia del problema de ordenamiento.

Un *algoritmo* posee varias propiedades:

1. Un algoritmo debe ser correcto. En otras palabras, debe calcular la función deseada, convirtiendo cada entrada en la salida correcta.
2. Un algoritmo está compuesto de una serie de pasos concretos (cada uno claramente entendido).
3. No debe existir ambigüedad en cuál es el paso siguiente a realizar.
4. Debe estar compuesto de un número finito de pasos.
5. Un algoritmo debe terminar.

Programas: podemos pensar de un programa de computadora como una instancia, o representación concreta, de un algoritmo en algún lenguaje de programación.

Eficiencia de algoritmos

Existen muchos enfoques para resolver un problema. ¿Cómo escogemos entre ellos? Generalmente hay dos metas en el diseño de programas de cómputo:

- El diseño de un algoritmo que sea fácil de entender, codificar y depurar. De esta meta se encarga la *Ingeniería de Software*.

- El diseño de un algoritmo que haga uso eficiente de los recursos de la computadora. De esta meta se encarga el Análisis y Diseño de algoritmos. El *análisis de algoritmos* nos permite medir la dificultad inherente de un problema y evaluar la eficiencia de un algoritmo.

Ejercicios

1. Dé tres ejemplos de tipo de datos.
2. Dé el ejemplo de un problema, que tenga más de un algoritmo para resolverlo, y diferentes programas para implementarlo.
3. Dé tres ejemplos de tipo de datos abstractos y las estructuras de datos que pueden usarse para implementarlos.
4. Para el problema de cambiar una llanta ponchada, dé dos algoritmos para resolverlo y explique cuál es más eficiente.

2. Preliminares matemáticos

Conjuntos

Un *conjunto* es una colección de miembros o elementos distinguibles. Los miembros se toman típicamente de alguna población más grande conocida como tipo base. Cada miembro del conjunto es un elemento primitivo del tipo base o es un conjunto. No hay concepto de duplicación en un conjunto. Un *orden lineal* tiene las siguientes propiedades:

1. Para cualesquiera elementos a y b en un conjunto C, sólo una de las siguientes condiciones es verdadera: $a < b$, $a = b$, o $a > b$. Esta es la *propiedad de tricotomía*.
2. Para todos los elementos a, b y c en el conjunto C, si $a < b$, y $b < c$, entonces $a < c$. Esta es la *propiedad de transitividad*.

Notación miscelánea

Una *permutación* de una secuencia es simplemente los miembros de la secuencia arreglados en cualquier orden. Si la secuencia contiene n elementos, existen $n!$ permutaciones.

Piso y techo de un número real x. Son respectivamente el mayor entero menor o igual que x, y el menor entero mayor o igual a x.

Operador módulo. Esta función regresa el residuo de una división entera. Generalmente se escribe n mod m y el resultado es el entero r, tal que $n = qm + r$ para q un entero y $0 \leq r$ y $r < m$. Por ejemplo, 5 mod 3 = 2, y 25 mod 3 = 1.

Un *logaritmo de base b* para y, es la potencia x al cual debe elevarse b para obtener y. $\text{Log}_b\, y = x \Leftrightarrow b^x = y$.

Ejemplo de uso de los logaritmos:

¿Cuál es el número mínimo de bits necesarios para codificar una colección de objetos?

1. La respuesta es techo($\log_2 n$).
2. Por ejemplo, si se necesita almacenar 750 códigos, se requerirá al menos techo($\log_2 750$) = 10 bits para tener 750 códigos distintos. De hecho con 10 bits hay $2^{10} = 1\ 024$ códigos distintos disponibles.

En el análisis de algoritmos se trabaja descomponiendo un problema en subproblemas más pequeños.

- Por ejemplo, la búsqueda binaria de un valor dado en una lista ordenada. ¿Cuántas veces se puede dividir a la mitad una lista de tamaño n, hasta que quede un solo elemento en la lista final? La respuesta es techo($\log_2 (n + 1)$).

Los logaritmos tienen las siguientes propiedades:

- $\text{Log}(n^*m) = \log n + \log m$.
- $\text{Log}\ n/m = \log n - \log m$.
- $\text{Log}\ n^r = r \log n$.
- $\text{Log}_a\ n = \log_b\ \text{n}\ /\ \log_b\ a$ (para a y b enteros). Esta última propiedad dice que el logaritmo de n en distintas bases, está relacionado por una constante (que es el logaritmo de una base en la otra). Así que el análisis de complejidad que se hace en una base de logaritmos, puede fácilmente traducirse en otra, simplemente con un factor de proporcionalidad.

Recursión

Un algoritmo es *recursivo* si se aplica a sí mismo para hacer parte del trabajo. Para que este enfoque sea exitoso, la aplicación debe ser en un problema menor que el originalmente planteado.

En general, un algoritmo recursivo tiene dos partes:

- El caso base, que maneja una entrada simple que puede ser resuelta sin una llamada recursiva.
- La parte recursiva, que contiene una o más llamadas recursivas al algoritmo, donde los parámetros están en un sentido más cercano al caso base, que la llamada original.

Ejemplos:

- Cálculo del factorial, $n! = n\ (n - 1)!$, $1! = 1$
- Secuencia de Fibonacci $f(n) = f(n - 1) + f(n - 2)$, $f(1) = 1$, $f(0) = 1$.
- La solución del problema de las torres de Hanoi.
- Ordenamiento por quicksort.

El problema de las torres de Hanoi

Existen tres postes A, B y C, y en el poste A se ponen n discos de diámetro diferente, de tal manera que un disco de diámetro mayor siempre queda debajo de uno de diámetro menor. El objetivo consiste en mover los n discos del poste A, al poste C usando B como auxiliar.

Algoritmo recursivo

1. Si $n = 1$, mover el disco único de A a C y parar.
2. Mover el disco superior de A a B, $n - 1$ veces, usando C como auxiliar.
3. Mover el disco restante de A a C.
4. Mover los $n - 1$ discos de B a C usando A como auxiliar.

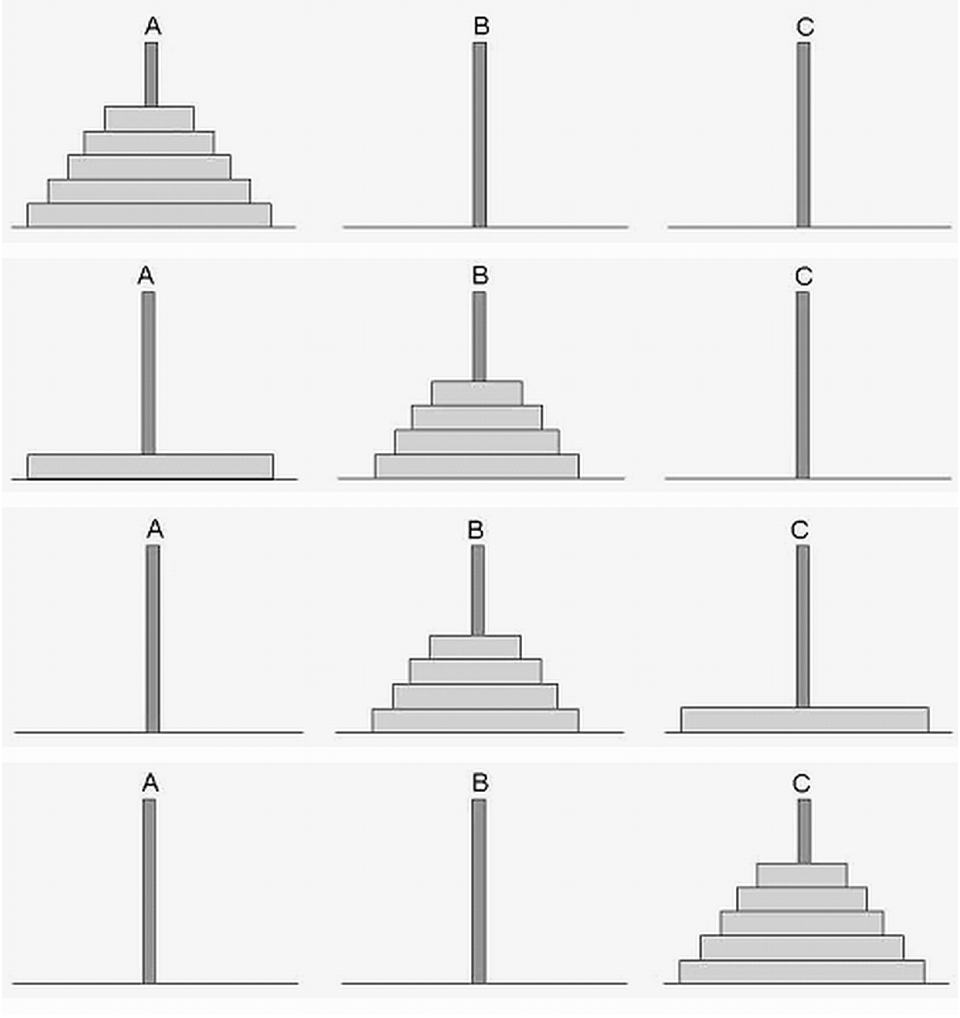


Figura 2. Ilustración de algunos de los pasos para resolver el problema de las torres de Hanoi, para 5 discos.

Ejemplo, para $n = 4$. Los discos se numeran de arriba hacia abajo. Los discos más pequeños reciben números más pequeños. Los pasos que realiza el algoritmo son los siguientes:

1. Mover disco 1 del poste A al poste B.
2. Mover disco 2 del poste A al poste C.
3. Mover disco 1 del poste B al poste C.
4. Mover disco 3 del poste A al poste B.
5. Mover disco 1 del poste C al poste A.
6. Mover disco 2 del poste C al poste B.
7. Mover disco 1 del poste A al poste B.
8. Mover disco 4 del poste A al poste C.
9. Mover disco 1 del poste B al poste C.
10. Mover disco 2 del poste B al poste A.
11. Mover disco 1 del poste C al poste A.
12. Mover disco 3 del poste B al poste C.
13. Mover disco 1 del poste A al poste B.
14. Mover disco 2 del poste A al poste C.
15. Mover disco 1 del poste B al poste C.

Sumatorias y recurrencias

Las sumatorias se usan mucho para el análisis de la complejidad de programas, en especial de ciclos, ya que realizan conteos de operaciones cada vez que se entra al ciclo. Cuando se conoce una ecuación que calcula el resultado de una sumatoria, se dice que esta ecuación representa una solución en forma cerrada.
Ejemplos:

$$\sum_{i=1}^{n} i = n(n+1)/2$$

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n)/6$$

$$\sum_{i=0}^{\infty} a^i = 1/(1-a),\ a \neq 1,\ |a| \leq |$$

El número de operaciones de un algoritmo recursivo se escribe naturalmente mediante una expresión recursiva.

Una *relación de recurrencia* define una función mediante una expresión que incluye una o más instancias (más pequeñas) de sí misma. Por ejemplo:

- $n! = (n - 1)!\, n$, $1! = 0! = 1$

La secuencia de Fibonacci:

- $\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$; $\text{Fib}(1) = \text{Fib}(2) = 1$
- 1, 1, 2, 3, 5, 8, 13,...

Técnicas de prueba matemática

Prueba por contradicción

La forma más fácil de refutar un teorema o enunciado, es mediante un contraejemplo.

Desafortunadamente, no importa el número de ejemplos que soporte un teorema, esto no es suficiente para probar que es correcto.

Para probar un teorema por contradicción, primero suponemos que el teorema es falso. Luego encontramos una contradicción lógica que surja de esta suposición. Si la lógica usada para encontrar la contradicción es correcta, entonces la única forma de resolver la contradicción es suponer que la suposición hecha fue incorrecta, esto es, concluir que el teorema debe ser verdad.

Ejemplos de pruebas, en donde se aplica la prueba por contradicción:

- Demostrar que no existe el entero más grande.
- Demostrar que $\sqrt{2}$ no es racional.
- Demostrar que el número de irracionales no es numerable.

Prueba por inducción matemática

La inducción matemática es similar a la recursión y es aplicable a una variedad de problemas.

La inducción proporciona una forma útil de pensar en el diseño de algoritmos, ya que estimula a resolver un problema, en términos de pequeños subproblemas.

Sea *T* un teorema a probar, y exprese *T* en términos de un parámetro entero positivo *n*. La inducción matemática expresa que *T* es verdad para cualquier valor de *n*, si las siguientes condiciones se cumplen:

- *Caso base*. *T* es verdad para $n = 1$.
- *Paso inductivo*. Si *T* es verdad para *n*, entonces *T* es verdad para *n*-1.

Estimación

Es un método para encontrar soluciones aproximadas a un problema. Consiste en hacer estimaciones rápidas para resolver un problema.

Puede formalizarse en tres pasos:

1. Determine los parámetros principales que afectan el problema.
2. Derive una ecuación que relacione los parámetros del problema.
3. Seleccione valores para los parámetros, y aplique la ecuación para obtener una solución estimada.

Ejemplo: ¿Cuántos libreros se necesitan para guardar libros que contienen en total dos millones de páginas?

Se estima que 500 páginas de un libro requieren cerca de 5 cm en la repisa del librero, con lo cual da 20 000 cm = (2 000 000/500)*5. Si una repisa es alrededor de 120 cm, entonces se necesitan 166 repisas (20 000/120). Si un librero contiene 5 repisas, entonces se necesitan 33 libreros (166/5) aproximadamente.

Ejercicios

1. Para $n = 5$, desarrolle y contabilice los movimientos del problema de las torres de Hanoi.
2. Pruebe por contradicción $\sqrt{2}$ que no es racional.
3. Pruebe por inducción matemática que $1 + 2 + 3 + \ldots + n = n(n + 1)/2$.
4. Pruebe por contradicción que no existe un entero más grande.
5. Pruebe que el conjunto de los números racionales Q es numerable.
6. Pruebe por contradicción que el conjunto de los números irracionales I, no es numerable.

3. Análisis de algoritmos

El *análisis de algoritmos* estima el consumo de recursos de un algoritmo.

Esto nos permite comparar los costos relativos de dos o más algoritmos para resolver el mismo problema.

El análisis de algoritmos les da a los diseñadores de algoritmos una herramienta poderosa para estimar si una solución propuesta satisface las restricciones de recursos de un problema.

El concepto de *razón de crecimiento*, es la razón a la cual el costo de un algoritmo crece conforme el tamaño de la entrada crece.

Introducción

¿Cómo comparar dos algoritmos para resolver un mismo problema en términos de eficiencia?

El análisis de algoritmos mide la eficiencia de un algoritmo, conforme crece el tamaño de la entrada.

Usualmente se mide el tiempo de ejecución de un algoritmo, y el almacenamiento primario y secundario que consume.

De consideración principal para estimar el desempeño de un algoritmo, es el número de *operaciones básicas* requeridas por el algoritmo para procesar una entrada de cierto tamaño.

Ejemplo*:* Algoritmo de búsqueda secuencial del máximo elemento en un arreglo.

Un programa en C para buscar el elemento máximo en un arreglo es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
int maximo (int* arreglo, int n)
{ int i, maxi;
 maxi = -65535;
 for(i = 0; i < n; i++)
 if(arreglo[i] > maxi)
 maxi = arreglo[i];
 return maxi;
}
main()
{
 int ejemplo[100], n, i, max;
 n=20;
 for(i=0; i<n; i++)
 {
 ejemplo[i]=rand();
 printf("%d ", ejemplo[i]);
 }
 max=maximo(&ejemplo[0], n);
 printf("%d\n", max);
}
```

Después de analizar el algoritmo de búsqueda secuencial, se obtiene que el número de operaciones realizadas es:

$T(n) = cn$ (donde n es el número de elementos a examinar, c es el tiempo que lleva examinar una variable y $T(n)$ es el número de operaciones en función de n).

Ejemplo: El tiempo requerido para copiar la primera posición de un arreglo es siempre c_1. (independientemente de n). Así $T(n) = c_1$.

Ejemplo: ¿Cuál es el orden de ejecución del siguiente fragmento de código?

```
#include <stdio.h>
main()
{
int i, j, sum, n;
sum=0 ;
n=20;
for(i=1 ; i<= n ; i++)
 for(j=1; j <= n; j++)
 sum++;
printf("%d\n", sum);
}
```

Solución: $T(n) = c_2\ n^2$ (c_2 es el tiempo en incrementar una variable).

El concepto de razón de crecimiento es extremadamente importante. Nos permite comparar el tiempo de ejecución de dos algoritmos sin escribir dos programas y ejecutarlos en la misma computadora.

Una razón de crecimiento de *cn* se le llama a menudo razón de *crecimiento lineal*.

Si la razón de crecimiento tiene el factor n^2, se dice que tiene una *razón de crecimiento cuadrático*.

Si el tiempo es del orden a^n se dice que tiene una razón de *crecimiento exponencial*.

Notemos que $2^n > 2n^2 > \log n$;

también para toda a, $b > 1$, $n^a > (\log n)^b$ y $n^a > \log n^b$ para toda a, $b > 1$, $a^n > n^b$

Mejor, peor y caso promedio

Para algunos algoritmos, las diferentes entradas para un tamaño dado pueden requerir diferentes cantidades de tiempo.

Por ejemplo, consideremos el problema de encontrar la posición de un valor *K*, que sólo ocurre una vez, dentro de un arreglo de *n* ele-

mentos. En el mejor caso encontraríamos a K en la primera posición del arreglo. En el peor caso, al final del arreglo. En el caso promedio, cercano a la mitad del arreglo.

¿Cuál es la ventaja de analizar cada caso? Si examinamos el peor de los casos, sabemos que al menos el algoritmo se desempeñará de esa forma.

En cambio, cuando un algoritmo se ejecuta muchas veces en muchos tipos de entrada, estamos interesados en el comportamiento promedio o típico. Desafortunadamente, esto supone que sabemos cómo están distribuidos los datos.

Si conocemos la distribución de los datos, podemos sacar provecho de esto, para un mejor análisis y diseño del algoritmo. Por otra parte, sino conocemos la distribución, entonces lo mejor es considerar el peor de los casos.

¿Una computadora más rápida o un algoritmo más rápido?

Si compramos una computadora diez veces más rápida, ¿en qué tiempo podremos ahora ejecutar un algoritmo? ¿En una décima parte del tiempo?

La respuesta depende del tamaño de la entrada de datos, así como de la razón de crecimiento del algoritmo.

Si la razón de crecimiento es lineal (por ejemplo $T(n) = cn$) entonces, 100 000 números serán procesados en la nueva máquina en el mismo tiempo que 10 000 números en la antigua computadora.

¿De qué tamaño (valor de n) es el problema que podemos resolver con una computadora X veces más rápida (en un intervalo de tiempo fijo)?

Por ejemplo, supongamos que una computadora resuelve un problema de tamaño n en un minuto. Ahora supongamos que tenemos una computadora 10 veces más rápida, ¿de qué tamaño es el problema que podemos resolver?

Para obtener la respuesta, necesitamos conocer la capacidad de procesamiento de la computadora inicial, y el orden de crecimiento que caracteriza a cada problema.

Supongamos que tenemos una computadora que puede ejecutar 10 000 operaciones básicas en un segundo.

Si el orden de crecimiento del algoritmo analizado es: $f(n) = 10n$. Igualando $f(n) = 10\ 000$. Resulta: $10n = 10\ 000$. Lo que implica $n = 1\ 000$. Es decir, que si el orden del algoritmo es $10n$, y disponemos de 10 000 operaciones básicas por segundo, entonces podemos procesar 1 000 datos en un segundo.

A continuación presentamos una tabla que calcula el número de elementos n, que podemos procesar con una computadora de 10 000 operaciones por segundo, en función de la tasa de crecimiento $f(n)$ del algoritmo. También se muestra el número de elementos n' que se puede procesar con una computadora 10 veces más rápida que procese 100 000 elementos por segundo. La cuarta columna de la tabla muestra como se relaciona $n´$ en términos de n. La quinta columna muestra el beneficio relativo, n'/n, de contar con una computadora 10 veces más rápida, en función de la tasa de crecimiento $f(n)$ de un algoritmo.

$f(n)$	n	$n´$	*cambio*	$n´/n$
$10n$	1 000	10 000	$n´ = 10n$	10.00
$20n$	500	5 000	$n´ = 10n$	10.00
$5n \log_2 n$	250	1 844	$\sqrt{}10n < n´ < 10n$	7.34
$2n^2$	70	223	$n´ = \sqrt{}10n$	3.18
2^n	13	16	$n´ = n + 3$	1.23

La conclusión del análisis de la tabla, es que si el orden de crecimiento del algoritmo es lineal, obtenemos un beneficio de 10 veces en procesamiento, con una computadora 10 veces más rápida. Sin embargo, al observar la quinta columna de arriba hacia abajo, notamos que el beneficio se va degradando, conforme crece la tasa de crecimiento del algoritmo.

Análisis asintótico

Comparemos la razón de crecimiento entre $10n$, $20n$, y $2n^2$, notaremos que $2n^2$, supera eventualmente a $10n$ y $20n$, difiriendo solamente en un valor mayor de n, donde ocurre el corte.

Por las razones anteriores, usualmente se ignoran las constantes cuando queremos una estimación del tiempo de ejecución u otros requerimientos de recursos del algoritmo. Esto simplifica el análisis y nos mantiene pensando en el aspecto más importante: la razón de crecimiento. A esto se le llama *análisis asintótico del algoritmo.*

En términos más precisos, el análisis asintótico se refiere al estudio de un algoritmo conforme el tamaño de entrada se vuelve grande o alcanza un límite (en el sentido del cálculo).

Sin embargo, no siempre es razonable ignorar las constantes, cuando se comparan algoritmos que van a ejecutar en valores relativamente pequeños de n.

Cota superior

La *cota superior de un algoritmo*, indica una cota o la máxima razón de crecimiento que un algoritmo puede tener. Generalmente hay que especificar si es para el mejor, peor o caso promedio.

Por ejemplo, podemos decir: "este algoritmo tiene una cota superior a su razón de crecimiento de n^2 en el caso promedio".

Se adopta una notación especial llamada *O-grande*, por ejemplo $O(f(n))$ para indicar la cota superior del algoritmo de orden $f(n)$.

En términos precisos, si $T(n)$ representa el tiempo de ejecución de un algoritmo, y $f(n)$ es alguna expresión para su cota superior, $T(n)$ está en el conjunto $O(f(n))$, si existen dos constantes positivas c y n_0 tales que $|T(n)| \leq c\ |f(n)|$ para todo $n > n_0$.

Ejemplo: Consideremos el algoritmo de búsqueda secuencial para encontrar un valor especificado en un arreglo de tamaño n. Si el visitar y comparar contra un valor en el arreglo, requiere c_s pasos, entonces en el caso promedio $T(n) = c_s\ n/2$. Para todos los valores $n > 1$. $|c_s\ n/2| \leq c_s\ |n|$. Por lo tanto, por definición, $T(n)$ está en $O(n)$ para $n_0 = 1$, y $c = c_s$.

El sólo saber que algo es $O(f(n))$ únicamente nos dice qué tan mal se pueden poner las cosas. Quizás la situación no es tan mala. De la definición podemos ver que si $T(n)$ está en $O(n)$, también está en $O(n^2)$ y $O(n^3)$, etcétera. Por lo cual se trata en general de definir la mínima cota superior.

Cota inferior

Existe una notación similar para indicar la mínima cantidad de recursos que un algoritmo necesita para alguna clase de entrada. La *cota inferior de un algoritmo*, denotada por el símbolo Ω, pronunciado *Gran Omega* u *Omega*, tiene la siguiente definición:

$T(n)$ está en el conjunto $\Omega(g(n))$, si existen dos constantes positivas c y n_0 tales que $|T(n)| > c|g(n)|$ para todo $n > n_0$.

Ejemplo: Si $T(n) = c_1n^2 + c_2n$ para $c1$ y $c2 > 0$, entonces:
$|c_1n^2 + c_2n| \geq |c_1 n^2| \geq c_1|n^2|$
Por lo tanto, $T(n)$ está en $\Omega(n^2)$.

Notación Θ

Cuando las cotas superior e inferior son la misma, indicamos esto usando la notación Θ (Theta). Se dice que un algoritmo es $\Theta(h(n))$, si está en $O(h(n))$ y está en $\Omega(h(n))$.

Por ejemplo, como un algoritmo de búsqueda secuencial está tanto en $O(n)$, como en $\Omega(n)$ en el caso promedio, decimos que es $\Theta(n)$ en el caso promedio.

Dada una expresión aritmética que describe los requerimientos de tiempo para un algoritmo, las cotas inferior y superior siempre coinciden. En general se usa la notación O y Ω , cuando no conocemos exactamente, sino sólo de manera acotada a un algoritmo.

Reglas de simplificación

Una vez que se determina la ecuación del tiempo de ejecución para un algoritmo, es relativamente sencillo derivar las expresiones para: O, Ω y Θ.

Existen algunas reglas sencillas que nos permiten simplificar las expresiones:

1. Si $f(n)$ está en $O(g(n))$ y $g(n)$ está en $O(h(n))$, entonces $f(n)$ está en $O(h(n))$.
   Esta regla nos dice que si alguna función $g(n)$ es una cota superior para una función de costo, entonces cualquier cota superior para $g(n)$, también es una cota superior para la función de costo.
   Nota: Hay una propiedad similar para la notación Ω y Θ.

2. Si $f(n)$ está en $O(k\, g(n))$ para cualquier constante $k > 0$, entonces $f(n)$ está $O(g(n))$
   El significado de la regla es que se puede ignorar cualquier constante multiplicativa en las ecuaciones, cuando se use notación de O-grande.

3. Si $f_1(n)$ está en $O(g_1(n))$ y $f_2(n)$ está en $O(g_2(n))$, entonces $f_1(n) + f_2(n)$ está en $O(\max(g_1(n), g_2(n)))$.
   La regla expresa que dadas dos partes de un programa ejecutadas en secuencia, sólo se necesita considerar la parte más cara.

4. Si $f_1(n)$ está en $O(g_1(n))$ y $f_2(n)$ está en $O(g_2(n))$, entonces $f_1(n)f_2(n)$ está en $O(g_1(n)g_2(n))$.
   Esta regla se emplea para simplificar ciclos simples en programas. Si alguna acción es repetida un cierto número de veces, y cada repetición tiene el mismo costo, entonces el costo total es el costo de la acción multiplicado por el número de veces que la acción tuvo lugar.

Tomando las tres primeras reglas colectivamente, se pueden ignorar todas las constantes y todos los términos de orden inferior para determinar la razón de crecimiento asintótico para cualquier función de costo, ya que los términos de orden superior pronto superan a los términos de orden inferior en su contribución en el costo total, conforme n se vuelve grande.

Ejemplos de cálculo del tiempo de ejecución de un programa

Veamos el análisis de un simple enunciado de asignación a una variable entera: $a = b$; como el enunciado de asignación toma tiempo constante, está en $\Theta(1)$. Consideremos un simple ciclo *for*:

```
#include <stdio.h>
main()
{
int i, n, sum;
sum=0;
n=20;
for(i=1; i<=n; i++)
sum += n;
printf("%d\n", sum);
}
```

La primera línea es $\Theta(1)$. El ciclo *for* es repetido n veces. La tercera línea toma también un tiempo constante, por la regla de simplificación (4), el costo total por ejecutar las dos líneas que forman el ciclo *for* es $\Theta(n)$. Por la regla (3), el costo por el entero fragmento de código es también $\Theta(n)$. Analicemos un fragmento de código con varios ciclos *for*, algunos de los cuales están anidados.

```
#include <stdio.h>
main()
{
int i, j, k, n, a[100], sum;
n=20;
sum=0;
for(j=1; j<=n; j++)
 for(i=1; i<=j; i++)
 sum++;
for(k=1; k<=n; k++)
        a[k]= k-1;
}
```

Este código tiene tres partes: una asignación y dos ciclos.

La asignación toma tiempo constante, llamémosla c_1. El segundo ciclo es similar al ejemplo anterior y toma $c_2n = \Theta(n)$. Analicemos ahora el primer ciclo, que es un doble ciclo anidado. En este caso trabajemos de adentro hacia fuera.

La expresión sum++ requiere tiempo constante, llamémosle c_3.

Como el ciclo interno es ejecutado j veces, por la regla (4), tiene un costo de $c_3 j$. El ciclo exterior es ejecutado n veces, pero cada vez el costo del ciclo interior es diferente.

El costo total del ciclo es c_3 veces la suma de los números 1 a n, es decir:

$\sum_{i=1}^{n} = n(n+1)/2$, que es $\Theta(n^2)$. Por la regla (3), $\Theta(c_1 + c_2n + c_3n^2)$ es simplemente $\Theta(n^2)$.

Comparemos el análisis asintótico de los siguientes fragmentos de código:

```
#include <stdio.h>
main()
{
int i, j, n, sum1, sum2;
n=20;
sum1=0;
for(i=1; i<=n; i++)
      for(j=1; j<=n; j++)
      sum1++;
printf("%d\n", sum1);
sum2=0;
for(i=1; i<=n; i++)
 for(j=1; j<=i; j++)
 sum2++;
printf("%d\n", sum2);
}
```

El primer fragmento ejecuta el enunciado sum1++, precisamente n^2 veces. Por otra parte, el segundo fragmento de código tiene un costo aproximado de ½ n^2. Así ambos códigos tienen un costo de $\Theta(n^2)$.

Ejemplo: No todos los ciclos anidados son $\Theta(n^2)$:

```
#include <stdio.h>
main()
{
int j, k, sum1, n;
n=20;
sum1=0;
for(k=1; k<=n; k*=2)
      for(j=1; j<=n; j++)
      sum1++;
printf("%d\n", sum1);
}
```

El costo es $\Theta(n \log_2 n)$.

¿Qué hay sobre otros enunciados de control?

Los ciclos *while* son analizados en una manera similar a los *for*.

El costo de un enunciado *if* en el peor caso es el mayor de los costos de la parte *if* y *else*.

Otros ejemplos, el factorial dado en forma recursiva, su costo, está dado por una relación de recurrencia: $T(n) = T(n - 1) + c$.

La solución en forma cerrada es $\Theta(n)$. La búsqueda en general en un arreglo es $\Theta(n)$. Por otra parte, si el arreglo está ordenado y si empleamos búsqueda binaria, el costo es $\Theta(\log_2 n)$.

Analizando problemas

Tiene sentido decir que la cota superior para un problema no puede ser peor que la cota superior para el mejor algoritmo que conocemos

para el problema. A menos, claro está, que no conozcamos que hay un mejor algoritmo. Por ejemplo, para problemas de ordenamiento sabemos que al menos debemos revisar los n elementos, por lo cual $\Omega(n)$. Por otra parte, existen algoritmos de ordenamiento cuyo tiempo de ejecución es $O(n \log n)$ en el peor de los casos. Se puede demostrar que cualquier algoritmo de ordenamiento debe ejecutar en $\Omega(n \log n)$ en el peor de los casos. Por lo cual, se puede concluir que el problema de ordenamiento es $\Theta(n \log n)$ en el peor de los casos.

Múltiples parámetros

En ocasiones puede ser conveniente expresar el costo de un algoritmo en función de más de una variable.

Por ejemplo, consideremos el problema de determinar la frecuencia de C colores (0... C - 1), en una imagen de P pixeles y luego ordenar por la frecuencia de mayor a menor.

```
for(i=0; i<C; i++)
cuenta[i]=0;
for(j=0; j<P; j++)
 cuenta[imagen(j)]++;
ordena(cuenta)
```

El costo para el primer ciclo es $\Theta(C)$. Para el segundo ciclo el costo es $\Theta(P)$.

El tiempo del ordenamiento es $\Theta(C \log C)$. Por lo cual, el costo total es: $\Theta(P + C \log C)$.

Cota de espacio

De manera similar al costo de procesamiento, también puede calcularse el orden en el almacenamiento. Por ejemplo, ¿cuáles son los requerimientos de espacio para un arreglo de n enteros? Si cada entero requiere c bytes, entonces el arreglo requiere cn bytes, que es: $\Theta(n)$.

Imaginemos que queremos registrar la amistad entre n personas. Podemos hacer esto con un arreglo de tamaño n x n. Para n personas, el tamaño total del arreglo es $\Theta(n^2)$.

El propósito principal de una estructura es almacenar datos en una forma que permita accesar de manera eficiente esos datos. Para lograrlo, puede ser necesario almacenar información adicional, de donde están los datos dentro de la estructura de datos.

Toda la información almacenada adicionalmente a los datos actuales se llama gastos de administración (*overhead*), el propósito es mantener esta cantidad en un mínimo, mientras que se maximiza el acceso.

Un principio importante en el diseño de algoritmos es a menudo conocido como principio de balance espacio/tiempo (*space/time tradeoff*). Este principio dice que uno puede lograr una reducción en tiempo, si uno sacrifica en espacio y viceversa.

Ejercicios

1. Para un algoritmo de su interés que acepte diferentes cantidades de datos, grafique el número de datos contra el tiempo de ejecución, y a partir de la comparación con gráficas de funciones conocidas, estime cotas inferiores y superiores para el algoritmo.
2. Determine cotas superiores e inferiores para el algoritmo tradicional de multiplicación de matrices.
3. Si una torre de Hanoi tiene n discos, obtenga una expresión en función de n, para el número de movimientos que es necesario realizar para resolver el problema.
4. Ordene la siguientes funciones por orden de crecimiento: $n!$, n^3, $\ln n$, e^n.
5. Encuentre una cota superior para el algoritmo de ordenamiento de su elección.
6. ¿Cuál es el orden del mejor algoritmo de ordenamiento conocido?
7. ¿Cuál es el orden de crecimiento de la búsqueda secuencial de datos?

8. ¿Cuál es el orden de crecimiento de la búsqueda binaria?
9. ¿Cuál es el orden de crecimiento del algoritmo que obtiene los números primos menores o iguales a un número entero n, si utiliza el método de la división por prueba, a partir de la definición de número primo?
10. ¿Cuál es la expresión que da la cantidad de números primos menores o iguales a n?
11. Si se compra una computadora tres veces más rápida, genere una tabla que muestre para algoritmos con distintos órdenes de crecimiento, ¿cuál sería el beneficio en este caso?
12. Para un algoritmo que ya tenga programado, compare su ejecución en al menos 2 computadoras adicionales a la usual, y de acuerdo a las diferencias entre las capacidades de las computadoras, vea si la mejora en rendimiento, vista experimentalmente, coincide con la esperada teóricamente y de acuerdo al orden del algoritmo. Por ejemplo, si se ejecuta en una computadora que es el doble de rápida, y si el algoritmo es lineal, vea si se percibe una mejora del doble de desempeño. Obtener conclusiones.

4. Gráficas

Una gráfica está compuesta de dos partes. La primera, es un conjunto de *nodos* o *vértices*. La segunda, es un conjunto de interconexiones entre vértices, llamadas *arcos, aristas* (*edges*).

En general, las gráficas permiten conexiones entre cualquier par de vértices. Los árboles y listas pueden ser vistas como una forma restringida de gráficas, así que en cierto sentido, las gráficas son lo más general de las estructuras de datos.

Las gráficas son ampliamente usadas para modelar tanto sistemas del mundo real como problemas abstractos.

Ejemplos de aplicaciones de las gráficas:

- Modelos de conectividad en redes de cómputo y telecomunicaciones.
- Representación de un mapa como un conjunto de localidades con distancias entre las mismas, para calcular las rutas más cortas.
- Modelado de capacidad de flujo en redes de transporte.
- Para encontrar una ruta entre una condición inicial y una meta; por ejemplo, en la resolución de problemas en inteligencia artificial.
- Para modelar algoritmos de computadora, mostrando las transiciones de un estado a otro.

- Para encontrar un orden aceptable en la realización de tareas en una actividad compleja, tal como la construcción de grandes edificios (determinación de la ruta crítica en un plan de trabajo o por ejemplo, el tema conocido como *planning* en *IA*).
- Modelando relaciones entre familias, negocios, o taxonomías científicas.

Terminología y representaciones

Una *gráfica G = (V, E)* consiste en un conjunto de vértices *V* y un conjunto de arcos *E*, tal que cada arco en *E* es una conexión entre un par de vértices en *V*. El número de vértices se denota por $|V|$, y el número de arcos por $|E|$.

$|E|$ puede variar de cero a un máximo de $\Theta(|V|^2)$.

Una gráfica con pocos arcos es llamada *poco densa o dispersa*, mientras que una gráfica con muchos arcos es llamada *densa*. Una gráfica conteniendo todos los posibles arcos se llama *completa.*

Una gráfica cuyos arcos están dirigidos de un vértice al otro se llama *gráfica dirigida* o *dígrafo*.

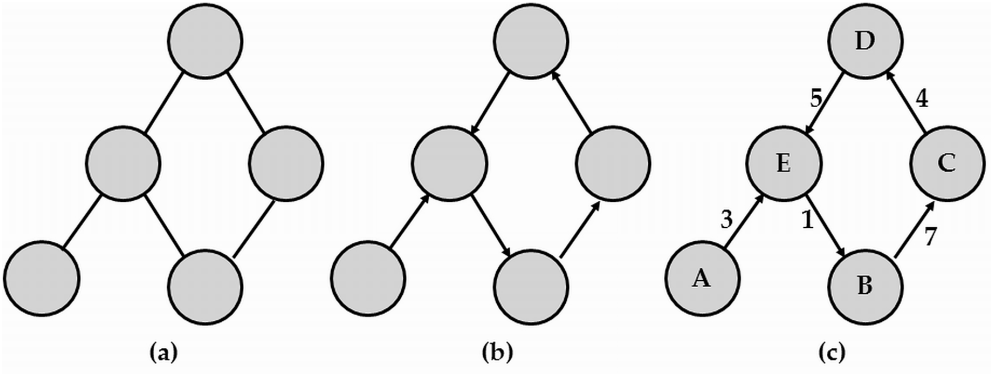


Figura 3. (a) Una gráfica. (b) Una *gráfica dirigida (dígrafo).* (c) Una gráfica dirigida con pesos asociados a los arcos. Existe un camino simple del vértice D al B, conteniendo los vértices D, E, B. Los vértices A, E, B, C, D, E, también forman un camino pero no uno simple. Los vértices E, B, C, D forman un ciclo simple.

Una gráfica cuyos arcos no están dirigidos se llama *gráfica no dirigida*.

Una gráfica con etiquetas asociadas a sus vértices es llamada *gráfica etiquetada*.

Dos vértices son *adyacentes* si están unidos por un arco. Tales vértices también se llaman vecinos. Un arco que conecta los vértices u y v se escribe (u, v). Tal arco se dice que es incidente en los vértices u y v. Asociado con cada arco podemos tener un costo o peso.

Una secuencia de vértices $v_1, v_2, ..., v_n$ forma un *camino* de longitud n - 1 si existen arcos de v_i a v_{i+1} para $1 \leq i < n$.

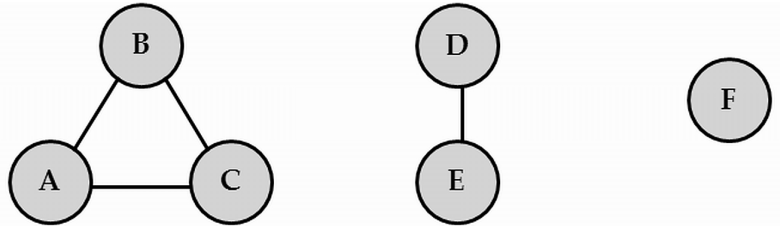
Un camino es *simple* si todos los vértices en el camino son distintos. La *longitud de un camino* es el número de arcos que contiene.

Un ciclo es un camino de longitud 2 o más que conecta algún vértice v_i consigo mismo. Un ciclo es simple si el camino es simple, excepto por el primer y último vértice que son el mismo.

Una *subgráfica S* es una gráfica formada de la gráfica G, seleccionando un subconjunto V_s de los vértices de G y algunos arcos de G, ambos de los cuales están en V_s.

Una gráfica no dirigida es *conexa* si existe al menos un camino de cualquier vértice al otro.

Las subgráficas máximamente conectadas de una gráfica no dirigida se llaman *componentes conexas*.



Una gráfica no dirigida con tres componentes conexas. Los vértices A, B y C forman una componente conexa. Los vértices D, E forman una segunda componente conexa. El vértice F por sí mismo forma una tercera componente conexa.

Una gráfica sin ciclos es llamada *acíclica*. Así, una gráfica dirigida sin ciclos es llamada *gráfica acíclica dirigida*.

Un *árbol libre* es una gráfica no dirigida, conexa, sin ciclos simples. Equivalentemente, es conexo y tiene $|V|-1$ arcos.

Hay dos métodos comúnmente usados para representar gráficas:

- La *matriz de adyacencia*, que emplea un arreglo de tamaño $|V| \times |V|$, y está marcada en la posición v_{ij}, si existe un arco de v_i a v_j. De aquí que los requerimientos de espacio son $\Theta(V^2)$. Puede colocarse un bit en la posición v_{ij} para indicar si existe un arco entre ambos nodos, o bien un número que indique el costo del arco.
- La *lista de adyacencia* es un arreglo de listas ligadas. El arreglo es de tamaño $|V|$, con la posición i almacenando un apuntador a la lista ligada de arcos para el vértice v_i. El costo de almacenamiento de la lista de adyacencia es: $\Theta(|V|+|E|)$.

Tanto la matriz de adyacencia como la lista de adyacencia puede usarse para almacenar gráficas dirigidas o no dirigidas. Cada arco de una gráfica no dirigida conectando a los vértices u y v es representado por dos arcos dirigidos: uno de u a v y otro de v a u.

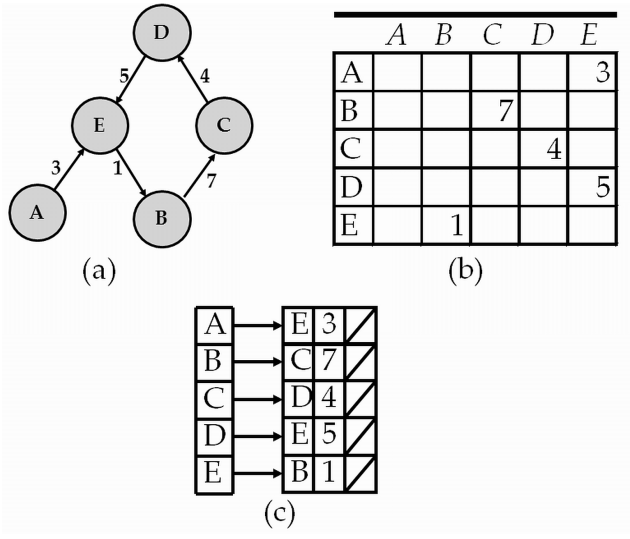


Figura 4. (a) *Gráfica dirigida*. (b) *Matriz de adyacencia*. (c) *Lista de adyacencia*.

Ventajas y desventajas de las dos representaciones:

- La ventaja de la matriz de adyacencia es que el costo en almacenamiento y velocidad de acceso es constante.
- La ventaja de la lista de adyacencia se encuentra en un menor costo de almacenamiento, siempre y cuando la gráfica tenga pocos arcos.

Recorridos de gráficas

A menudo es útil visitar los vértices de una gráfica en algún orden específico basado en la topología de la gráfica. Esto es conocido como *recorrido de la gráfica* y es similar en concepto al recorrido de árboles.

Los algoritmos de recorrido de gráficas típicamente inician con un vértice e intentan visitar los vértices restantes desde ahí. Sin embargo existen algunos problemas que resolver:

1. Puede no ser posible alcanzar todos los vértices desde el vértice inicial (por ejemplo, si la gráfica es no conexa).
2. La gráfica puede contener ciclos (y podemos caer en ciclos infinitos).

Los algoritmos de recorrido de gráficas generalmente resuelven estos problemas manteniendo un *bit de marcación* para cada vértice de la gráfica. Al inicio todos los bits de marcación se limpian y el bit es marcado, cuando se visita el vértice durante el recorrido.

Una vez que el algoritmo de recorrido termina, podemos checar si todos los vértices han sido procesados, checando el arreglo de bits marcados. Sino están todos los vértices marcados, podemos continuar el recorrido desde otro vértice no marcado.

Para asegurar que se visitan todos los vértices, *visita_grafica* se llamaría como sigue en la siguiente gráfica G:

```
void visita_grafica(Grafica &G) {
 for(v=0; v< |V|; v++)
```

```
 G.marca[v] = NOVISITADO / /Inicialización bits de marcación.
for(v=0; v<| V |; v++)
 if(G.marca[v] == NOVISITADO)
 has_visita(G, v);
}
```

donde *has_visita* se reemplaza por un algoritmo de recorrido de gráfica.

Búsqueda en profundidad

Es cuando un vértice v es visitado durante la búsqueda, y recursivamente se visitan todos los vecinos no visitados. El efecto es seguir una rama a través de la gráfica hasta su conclusión, luego regresar y seguir la siguiente rama y así sucesivamente.

Este proceso crea un árbol de búsqueda en profundidad (*depth first*).

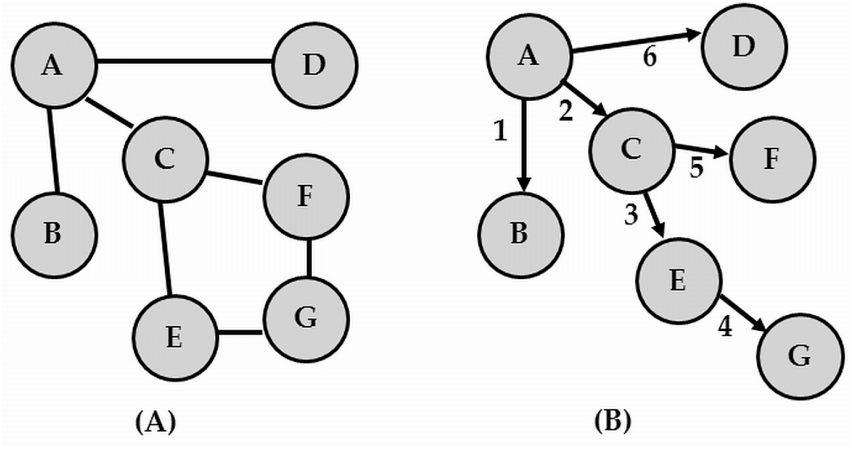


Figura 5. (A) Una gráfica. (B) El árbol de búsqueda en profundidad para la gráfica que inicia en el vértice A. Los números en los arcos indican el orden de expansión del árbol en profundidad.

Búsqueda en amplitud

El algoritmo de búsqueda en amplitud examina todos los vértices conectados al vértice inicial antes de examinar los vértices más allá. Este algoritmo es implementado de manera similar al de profundidad, excepto que una *cola* reemplaza la pila de recursión.

Nótese que si la gráfica es un árbol, y el vértice inicial está en la raíz, este algoritmo es equivalente a visitar los vértices nivel por nivel, de arriba hacia abajo.

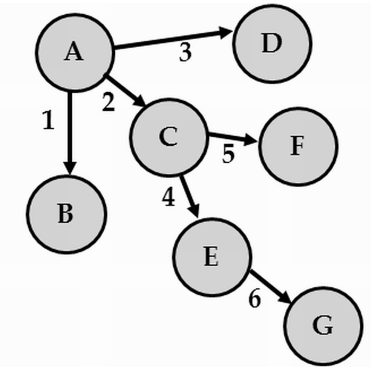


Figura 6. Árbol de búsqueda en amplitud.

Ordenamiento topológico

Una aplicación para el recorrido de gráficas es la solución del problema de los prerrequisitos.

Supongamos que queremos planificar una serie de tareas tales como clases o actividades, donde no podemos iniciar una tarea hasta que sus prerrequisitos se completen.

Queremos organizar las tareas en un orden lineal que nos permita completarlas una a la vez, sin violar ningún prerrequisito.

Podemos modelar el problema usando un DAG (*directed acyclic graph*). La gráfica es dirigida porque una tarea es un prerrequisito de la otra. No es cíclica porque un ciclo indicaría una serie conflictiva

de prerrequisitos que no podrían ser completados sin violar al menos un prerrequisito.

Los nodos que apuntan hacia un nodo son sus prerrequisitos.

El proceso de colocar los vértices de una DAG en un orden lineal para satisfacer las reglas de prerrequisitos es llamado un *orden topológico*.

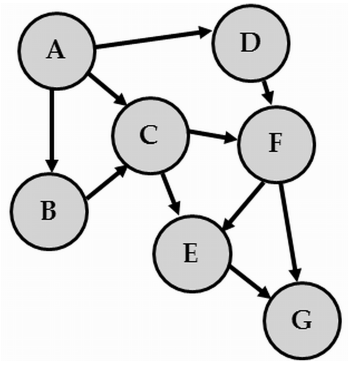


Figura 7. Una gráfica con dependencias que ilustra prerrequisitos. Un ordenamiento topológico válido es: A, B, C, D, F, E, G. Otro ordenamiento válido es: A, D, B, C, F, E, G.

Puede implantarse el ordenamiento topológico usando una cola. Para lograrlo, primero visitamos todos los vértices, contando el número de arcos que llevan a ese vértice (por ejemplo, contamos el número de prerrequisitos para cada vértice). Todos los vértices sin prerrequisitos son colocados en la cola. Entonces procesamos la cola. Cada vértice eliminado de la cola es impreso, y todos sus vecinos tienen sus conteos decrementados en uno. Cualquier vecino cuyo conteo sea cero es colocado en la cola. Si la cola se vuelve vacía sin imprimir todos los vértices, entonces la gráfica contiene un ciclo.

Problemas de caminos más cortos

Imaginemos que modelamos una red carretera como una gráfica dirigida cuyos arcos están etiquetados con números reales. Los números

representan la distancia (u otra métrica de costo, tal como tiempo de viaje) entre dos vértices.

Estas etiquetas pueden ser llamadas pesos, costos o distancias, dependiendo de la aplicación.

Dada tal gráfica, un problema típico es encontrar la longitud total del camino más corto entre dos vértices especificados. Este es un problema no trivial, ya que el camino más corto puede no estar a lo largo del arco que conecta dos vértices, sino en una ruta o camino, que involucra uno o más vértices intermedios.

Notación: $d(A, D)$ distancia más corta entre el vértice A y D. Sino existe un camino directo entre E y B, escribimos $d(E, B) = \infty$. $w(A, D)$ indica el peso del arco de A a D.

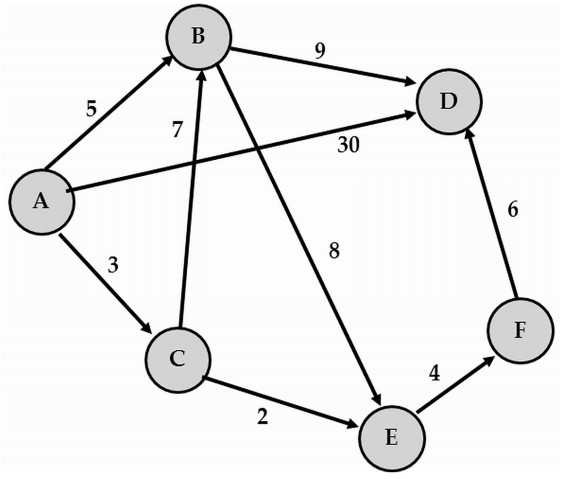
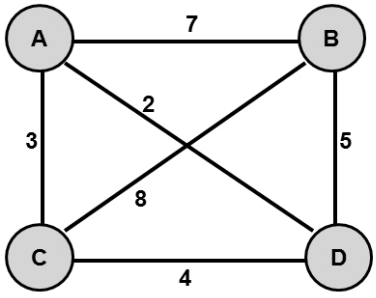


Figura 8. Existen rutas que visitan más vértices que pueden tener menor costo que una ruta directa. Ejemplo A, C, B, E, F, D; tiene un menor costo que A, D. Otra ruta de menor costo que el camino directo es A, B, D.

Problema del vendedor viajero

Consiste en encontrar el camino más corto a través de un conjunto de n ciudades donde cada ciudad es visitada exactamente una vez y se regresa a la ciudad de inicio.



RECORRIDO					LONGITUD	
A	B	C	D	A	21	
A	B	D	C	A	19	
A	C	B	D	A	18	ÓPTIMO
A	C	D	B	A	19	
A	D	B	C	A	18	ÓPTIMO
A	D	C	B	A	21	

Figura 9. Una solución de problema del viajero es por búsqueda exhaustiva.

El problema puede ser modelado como una gráfica con pesos, en donde los vértices representan las ciudades y los pesos las distancias.

El problema puede plantearse: cómo encontrar el circuito Hamiltoniano más corto.

Un *circuito Hamiltoniano* es definido como un ciclo que pasa por todos los vértices de la gráfica exactamente una vez.

Un circuito H puede definirse como una secuencia de $n + 1$ vértices adyacentes, v_{i0}, v_{i1},... $v_{in\text{-}1}$, v_{i0}, donde el primer vértice de la secuencia es el mismo que el último, y todos los otros n - 1 vértices son distintos.

Para solucionar el problema del vendedor viajero, podemos usar un enfoque de genera y prueba. El enfoque de genera y prueba es

un método de solución a problemas combinatorios. Sugiere generar cada uno de los elementos del dominio de un problema, seleccionar aquellos que satisfacen las restricciones del problema, y encontrar un elemento deseado (por ejemplo, el que optimiza alguna función objetivo).

Podemos obtener todos los viajes (que salen de un vértice), generando todas las permutaciones de las n - 1 ciudades intermedias. Luego calcular las longitudes de los viajes y encontrar el viaje más corto.

Se puede observar que hay recorridos que son equivalentes en donde sólo cambia la dirección del recorrido. Así, podemos reducir el número de permutaciones en la mitad. Aún así, el número de permutaciones sigue siendo $(n - 1)!/2$.

Caminos más cortos de una sola fuente

El problema del camino más corto de una sola fuente, consiste en: dado un vértice s en una gráfica G, encontrar el camino más corto entre s y cualquier otro vértice en G.

Podríamos querer solamente el camino más corto entre los vértices s y t. Desafortunadamente, no existe un mejor algoritmo (en el peor caso) para encontrar el camino más corto a un solo vértice, que encontrar caminos más cortos a todos los vértices.

Una aplicación de este problema ocurre en las redes de cómputo. En este caso se trata de encontrar la forma más barata para que una computadora envíe un mensaje (*broadcast*) a todas las otras computadoras en la red.

La red se modela con una gráfica con los pesos de los arcos indicando tiempo o costo para enviar un mensaje a una computadora vecina.

Para gráficas sin pesos (o donde los arcos tienen el mismo costo), los caminos más cortos de una sola fuente pueden encontrarse usando una simple búsqueda en amplitud.

Un enfoque para resolver este problema cuando los arcos tienen diferentes pesos, puede ser, procesar los vértices en un orden fijo. Etiquetando los vértices v_0 hasta v_{n-1}, con $s = v_0$.

Cuando procesamos el vértice v_1, tomamos el arco que conecta v_0 a v_1. Cuando procesamos v_2, consideramos la distancia más corta desde v_0 y comparamos esto a la distancia más corta de v_0 a v_1 y a v_2.

Cuando procesamos el vértice v_i, consideramos los caminos más cortos para los vértices v_0 hasta v_{i-1} que han sido procesados. Desafortunadamente, el verdadero camino más corto puede ir a través del vértice v_j, para $j > i$. Este camino o ruta no sería considerado por este algoritmo.

Sin embargo, este problema no ocurriría si procesamos los vértices en orden de distancia a s. Supongamos que hemos procesado en orden de distancia los primeros i - 1 vértices que están más cercanos a s. Llamemos a este conjunto de vértices S. Vamos ahora a procesar el i-ésimo vértice más cercano; llamémosle x.

El camino más corto de s a x debe tener su penúltimo vértice en S. Así:

$$d(s, x) = \min_{u \,\varepsilon\, S}(d(s, u) + w(u, x))$$

En otras palabras, el camino más corto de s a x es el mínimo sobre todos los caminos que van de s a u, y que tienen un arco de u a x, siendo u un vértice en S.

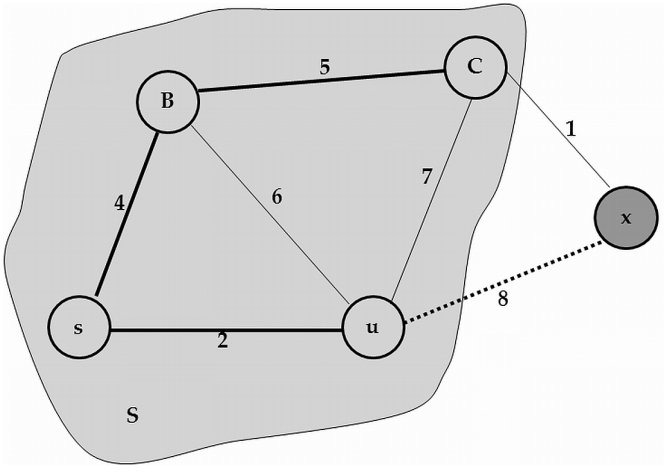


Figura 10. Ilustración del camino más corto de s a x. El arco final que conecta con x debe partir de un nodo que forme parte de S.

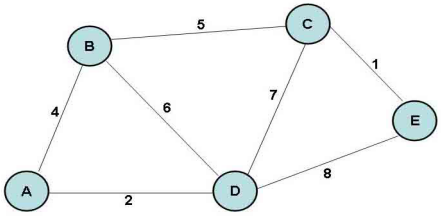
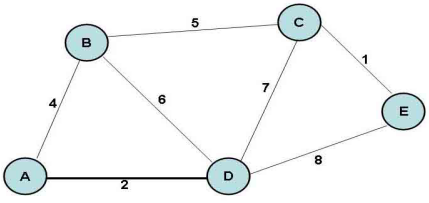
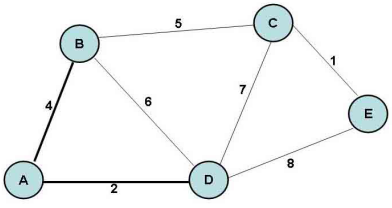
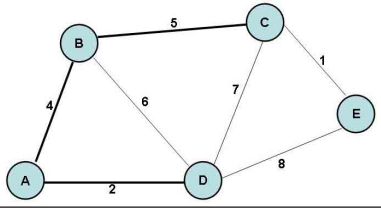
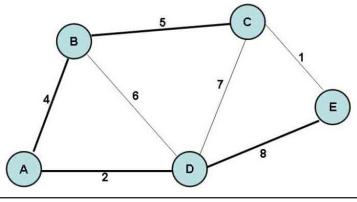
Vértice del árbol	*Vértices*	*Ilustración*
A (-1, 0)	*B(A, 4), C(-, ∞)* *D(A, 2), E(-, ∞)*	
D(A, 2)	*C(D, 7 + 2) B(A, 4),* *E(D, 8 + 2)*	
B(A, 4)	*C(B, 5 + 4) E(D, 8 + 2)*	
C(B, 9)	*E(D, 10)*	
E(D, 10)		

Figura 11. El vértice inicial es A. Listado de etapas del algoritmo de Dijkstra en la gráfica de ejemplo.

En la anterior figura el subárbol de los caminos más cortos ya encontrado es marcado con negro. El siguiente más cercano al vértice fuente v_0, u_t, es seleccionado comparando las longitudes de los subárboles extendidos por las distancias a los vértices adyacentes a los vértices de los subárboles.

Esta solución es conocida como el *algoritmo de Dijkstra*. Funciona manteniendo una estimación de distancia $D(x)$ para todos los vértices en V. Los vértices son procesados en orden de distancia de s. Siempre que un vértice v es procesado, $D(x)$ es actualizado para todo vecino x de v. Al final, el arreglo D contendrá los valores de distancia más cercanos.

En este algoritmo, la búsqueda del vértice no visitado con valor mínimo de D, se realiza simplemente buscando la lista de $|V|$ vértices. Como esta búsqueda es realizada $|V|$ veces, y como cada arco requiere un tiempo constante para actualizar D, el costo total para este enfoque es $\Theta(|V|^2 + |E|)$

(a) a (d):	a-d	Longitud 2
(a) a (b):	a-b	Longitud 4
(a) a (c):	a-b-c	Longitud 9
(a) a (e):	a-b-d-e	Longitud 10

Figura 12. Rutas óptimas para llegar del vértice inicial a los demás vértices en la gráfica. En la columna derecha se muestra la longitud de las rutas óptimas.

Caminos más cortos para todos los pares

Otro problema es encontrar la distancia más corta entre todos los pares de vértices en una gráfica.

Una solución es correr el algoritmo de Dijkstra $|V|$ veces, cada vez calculando caminos más cortos desde diferentes vértices iniciales.

Otra solución es conocida como el *algoritmo de Floyd*.

Definimos una *k-ruta* del vértice v al u, como cualquier ruta cuyos vértices intermedios (aparte de u y v), sus índices, son menores que k. Un 0-camino se define como un arco dirigido de v a u.

Definimos $D_k(v, u)$ como la longitud de la k-ruta más corta del vértice v al u.

Supongamos que ya conocemos la k-ruta más corta de v a u. La $(k + 1)$-ruta pasa a través del vértice k o bien no pasa. Si pasa a través de k, entonces el mejor camino es la mejor k-ruta de v a k seguido de la mejor k-ruta de k a u. De otra manera, guardamos la mejor k-ruta vista antes.

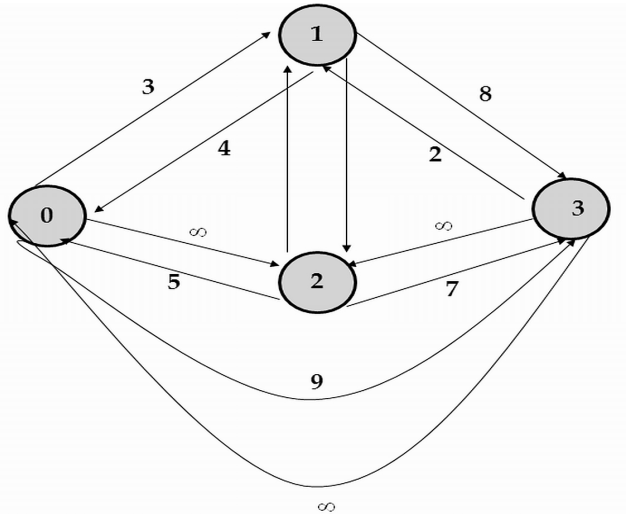


Figura 13. Ejemplos de k-rutas. La ruta 1, 3 es una 0-ruta. La ruta 3, 0, 2 no es una ruta 0, sino una 1-ruta (también es 2-ruta, 3-ruta y 4-ruta), ya que el vértice intermedio es 0.

El algoritmo de Floyd calcula la matriz de una gráfica con pesos con n vértices a través de una serie de matrices de n x n.

$$\boldsymbol{D}^{(0)},\ldots \boldsymbol{D}^{(k-1)}, \boldsymbol{D}^{(k)},\ldots \boldsymbol{D}^{(n)}$$

Cada matriz $\boldsymbol{D}^{(k)}$ contiene la longitud de los caminos más cortos de k-rutas.

El elemento $\boldsymbol{dij}^{(k)}$ en la matriz $\boldsymbol{D}^{(k)}$ ($k = 0..., n$) es igual a la longitud del camino más corto donde todos los caminos entre los vértices $\boldsymbol{i}$ y $\boldsymbol{j}$ son vértices intermedios numerados no más grandes que $\boldsymbol{k}$.

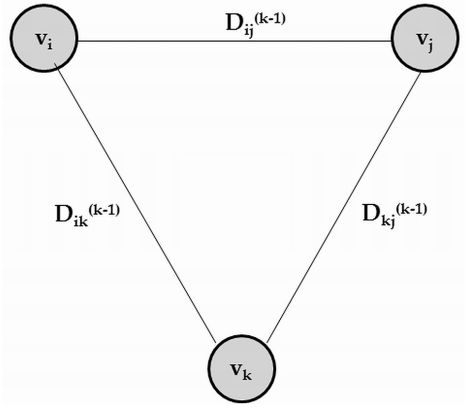


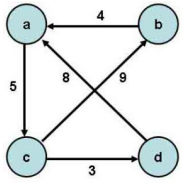
Figura 14. Ilustración de la condición que checa el algoritmo de Floyd:
$\boldsymbol{d_{ij}^{(k)} = \min \{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}}$ **para** $\boldsymbol{k \geq 1}$, $\boldsymbol{d_{ij}^{(0)} = w_{ij}}$

El algoritmo de Floyd simplemente checa todas las posibilidades en un triple ciclo. Al finalizar, el arreglo **D** guarda las distancias más cortas entre todos los pares.

```
void Floyd(Grafica& G) { //rutas óptimas para pares
      Int D[|V|][|V|]; //guardamos distancias
 for (i=0; i<|V|;i++) //Inicializamos D con los pesos de la gráfica
 for(j=0; j<|V|; j++)
 D[i][j]=G.peso(i, j);
 for(k=0; k<|V|; k++) //Calculamos todas las k-rutas
```

```
for(i=0; i<|V|; i++)
for(j=0; j<|V|; j++)
if(D[i][j] > (D[i][k] + D[k][j]))
D[i][j]= D[i][k]+D[k][j];
}
```

Claramente este algoritmo requiere $\Theta(|V|^3)$ tiempo de ejecución, y es la mejor elección para gráficas densas, porque es relativamente rápido y fácil de implantar.



D_0

	a	*b*	*c*	*d*
a	0	∞	5	∞
b	4	0	∞	∞
c	∞	9	0	3
d	8	∞	∞	0

Rutas más cortas sin vértice intermedio. D_0 es simplemente la matriz de pesos.

D_1

	a	*b*	*c*	*d*
a	0	∞	5	∞
b	4	0	**9**	∞
c	∞	9	0	3
d	8	∞	**13**	0

Rutas más cortas con vértices intermedios numerados no más grandes que 1, es decir, sólo *a*. Notemos las 2 nuevas rutas más cortas a través de *a*, marcadas en negritas.

D_2

	a	*b*	*c*	*d*
a	0	∞	5	∞
b	4	0	9	∞
c	**13**	9	0	3
d	8	∞	13	0

Rutas más cortas con vértices intermedios numerados no más grandes que 2. Es decir, se puede pasar por *a* y *b*.

D_3

	a	*b*	*c*	*d*
a	0	**14**	5	**8**
b	4	0	9	**12**
c	13	9	0	3
d	8	**22**	13	0

Rutas más cortas con vértices intermedios numerados no más grandes que 3. Es decir, se puede pasar por *a* y *b* y *c*.

D_4

	a	*b*	*c*	*d*
a	0	14	5	8
b	4	0	9	12
c	**11**	9	0	3
d	8	22	13	0

Rutas más cortas con vértices intermedios numerados no más grandes que 4. Es decir, se puede pasar por *a* y *b*, *c* y *d*.

Figura 15. Ejemplo de ejecución del algoritmo de Floyd.

Árboles expansores de mínimo costo

El problema del árbol expansor de mínimo costo *(minimum-cost spanning tree* (MST)) de una gráfica toma como entrada una gráfica conexa, no dirigida G, donde cada arco tiene una distancia o medida de peso asociado.

El MST es un subconjunto de los vértices de *G* que tiene costo mínimo total, sumando todos los valores en los arcos del subconjunto, y mantiene los vértices conectados.

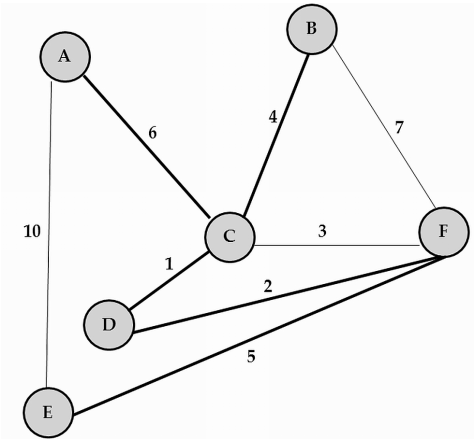


Figura 16. Una gráfica y su MST. Todas las líneas son arcos en la gráfica original. Las líneas en negritas, indican el subconjunto de arcos que forman el MST.

Ejemplos de aplicaciones donde una solución a este problema es útil, incluye:

- soldar el conjunto más pequeño de cables necesarios para conectar un conjunto de terminales en un circuito, y
- conectar un conjunto de ciudades por teléfono, con la menor cantidad de alambre.

Un MST no contiene ciclos. Si un conjunto propuesto de arcos tuviera un ciclo, un MST más barato podría tenerse removiendo cualquier arco en el ciclo. Así, el MST es un árbol con $|V|$ - 1 arcos.

El nombre de árbol expansor de costo mínimo proviene del hecho, que el conjunto requerido de arcos forma un árbol, expande los vértices (es decir, los conecta juntos), y tiene un costo mínimo.

Algoritmo de Prim para el cálculo del MST

El algoritmo de Prim comienza con cualquier vértice *N* en la gráfica, asignando este vértice en el MST. Seleccionamos el arco de mínimo costo conectado a *N*. Este arco conecta *N* a otro vértice; llamémosle *M*. Agregamos el vértice *M* y el arco (*N*, *M*) al MST.

A continuación seleccionamos el arco de costo mínimo que va, ya sea de *N* o *M* a cualquier otro vértice en la gráfica. Agregamos este arco y el nuevo vértice que llega al MST.

Este proceso continúa, y en cada paso expandemos el MST seleccionando el arco de mínimo costo desde un vértice actualmente en el MST, a un vértice no actualmente en el MST.

El algoritmo de Prim es muy similar al algoritmo de Dijkstra para encontrar caminos más cortos desde una sola fuente. La diferencia principal es que estamos buscando no el siguiente vértice más cercano al vértice inicial, sino el siguiente vértice más cercano a cualquier vértice actualmente en el MST.

A continuación se presenta una implantación del algoritmo de Prim que busca en la matriz de distancia, el siguiente vértice más cercano. El arreglo *V*[*i*] almacena el vértice previamente visitado que está más cercano al vértice *i*, de tal forma que sabemos qué vértice va en el MST, cuando el vértice *i* es procesado.

El algoritmo de Prim es un ejemplo de algoritmo codicioso (*greedy*). En cada paso del ciclo *for*, seleccionamos el arco de menor costo que conecta un vértice marcado a uno no marcado. No hay preocupación de si el MST realmente debería incluir el arco de mínimo costo o no. Esto conduce a una importante pregunta: ¿El algoritmo de Prim trabaja correctamente?

Claramente genera un árbol expansor (ya que en cada paso del ciclo adiciona un vértice no marcado al árbol expansor hasta que todos los vértices han sido agregados, pero ¿este árbol tiene costo mínimo?

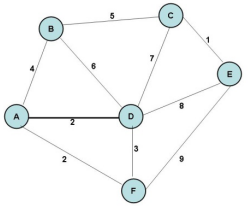
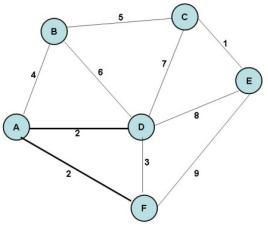
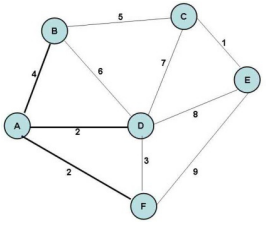
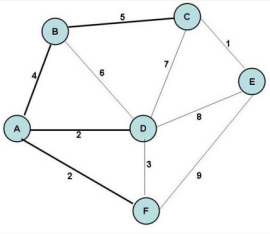
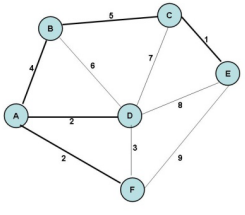
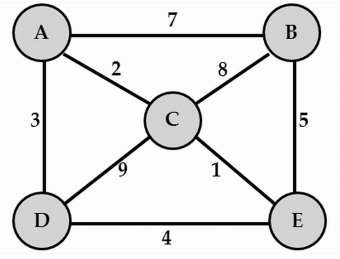
Vértice	*Vértice*	*Ilustración*
a(-, -)	*b*(*a*, 4), *c*(-, ∞), ***d(a, 2)***, *e*(-, ∞), *f*(*a*, 2)	
d(*a*, 2)	*b*(*a*, 4), *c*(*d*, 7), *e*(*d*, 8), **f*(a, 2)***	
f(a, 2)	***b(a, 4)***, *c*(*d*, 7), *e*(*d*, 8)	
b(a, 4)	***c(b, 5)***, *e*(*d*, 8)	
c(b, 5)	***e(c, 1)***	

Figura 17. Aplicación del algoritmo en una gráfica para encontrar el MST.

Ejercicios

1. Encuentre el árbol expansor de mínimo costo para la siguiente gráfica, empleando el algoritmo de Prim:



2. Para una compañía repartidora de bienes de consumo y que tenga al menos 6 puntos de entrega, genere la gráfica correspondiente y calcule la solución al problema del viajero.
3. Para la gráfica del problema 1, encuentre todos los caminos más cortos, para todos los pares, aplicando el algoritmo de Floyd.

5. Límites al cómputo

Ejemplos de algoritmos eficientes:

- Algoritmos de búsqueda en el peor caso: $O(\log n)$.
- Algoritmos de ordenamiento en el peor caso: $O(n \log n)$.
- Ruta más corta, todos los pares, Floyd, $\Theta(n^3)$.

Dado un problema para el cual conocemos algún algoritmo, siempre es posible escribir un algoritmo ineficiente para resolver el mismo problema.

- Ordenamiento, checar todas las permutaciones $O(n!)$.
- Árbol expansor de mínimo costo, probar todos los subconjuntos de arcos, $O(2^{|E|})$.

Desafortunadamente, existen muchos problemas de cómputo para los cuales los mejores algoritmos conocidos toman un largo tiempo.

- No es posible que ningún programa de cómputo que resuelva el problema de las torres de Hanoi, corra en menos que $2^n - 1$. Ya que al menos ese número de movimientos debe ser impreso.

Además de aquellos problemas cuyas soluciones deben tomar un largo tiempo en ejecutar, también existen algoritmos para los cuales no sabemos si existen o no, algoritmos eficientes.

Los mejores algoritmos que conocemos para tales problemas son muy lentos, pero quizás existen mejores algoritmos que esperan ser descubiertos.

Aunque el tener una solución de alto orden de computabilidad es malo, es aún peor tener un problema que no pueda ser resuelto.

Veremos que existen problemas intrínsecamente caros o imposibles de resolver.

Reducción

El concepto de reducción nos permite resolver un problema en términos de otro. Igualmente importante, cuando queremos entender la dificultad de un problema, la reducción nos permite hacer enunciados sobre cotas inferiores o superiores en el costo de un problema.

Un problema puede describirse o definirse, como un mapeo entre entradas y salidas. Por ejemplo:

Ordenamiento

Entrada: Una secuencia de enteros $x_0, x_1, x_2, \ldots x_{n-1}$
Salida: Una permutación: $y_0, y_1, \ldots y_{n-1}$ de la secuencia tal que $y_i \leq y_j$ siempre que $i < j$.

Una vez que tenemos un algoritmo para un problema (tal como para ordenamiento), podemos usarlo como una herramienta para un problema diferente. Esto se conoce en ingeniería de software como *reutilización de software.*

Emparejamiento

Entrada: Dos secuencias de enteros $X = (x_0, x_1, \ldots x_{n-1})$ y $Y = (y_0, y_1, \ldots y_{n-1})$.

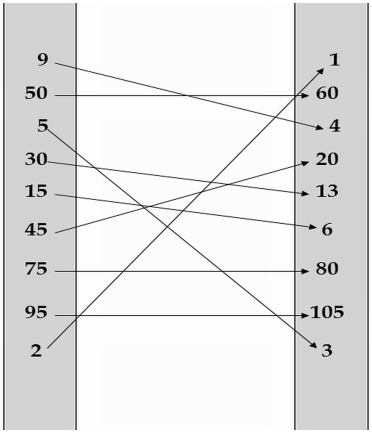
Salida: Un emparejamiento de elementos en las dos secuencias tal que el menor valor de X es correspondido con el menor elemento

de *Y*, el siguiente valor en magnitud de *X* es correspondido con el siguiente valor en magnitud de *Y*, y así sucesivamente.

Una forma de resolver el problema de emparejamiento es usar un programa existente de ordenamiento, para ordenar cada una de las secuencias, y luego aparear los elementos basados en sus posiciones del ordenamiento. Técnicamente, decimos que el emparejamiento es reducido a ordenamiento, ya que este último es usado para resolver el primero. La reducción de emparejamiento a ordenamiento ayuda a establecer una cota superior en el costo de emparejamiento.

Podemos definir formalmente la reducción como un proceso de tres pasos:

1. Transformamos una instancia arbitraria del primer problema en una instancia del segundo problema. En otras palabras, debe existir una transformación de cualquier instancia P del primer problema en una instancia P´ del segundo problema.
2. Aplicamos un algoritmo al segundo problema a la instancia P´, dando la solución SOL´.
3. Transformamos SOL´ a la solución de P, conocida como SOL. Notemos que SOL debe ser la solución correcta para P, para que la reducción sea aceptable.



Ejemplo: ¿Es la elevación al cuadrado de un número de *n* dígitos tan difícil como multiplicar dos números de *n* digitos?

Una simple prueba de reducción sirve para mostrar que elevar al cuadrado es tan duro como multiplicar.

$$X * Y = ((X + Y)^2 - (X - Y)^2) / 4$$

Un corolario de esto es que si puede encontrarse un algoritmo lineal para elevar al cuadrado, entonces puede ser usado para construir un algoritmo lineal para multiplicar.

Ejemplo: el algoritmo estándar de multiplicación de matrices, requiere un tiempo total de $\Theta (n^3)$. Se conocen algoritmos más rápidos, pero ninguno tan rápido para ser del orden de $O(n^2)$.

Ejemplo: Si las matrices simétricas pueden multiplicarse rápidamente, entonces también podríamos multiplicar rápidamente cualquier tipo de matrices, como lo demuestra la identidad:

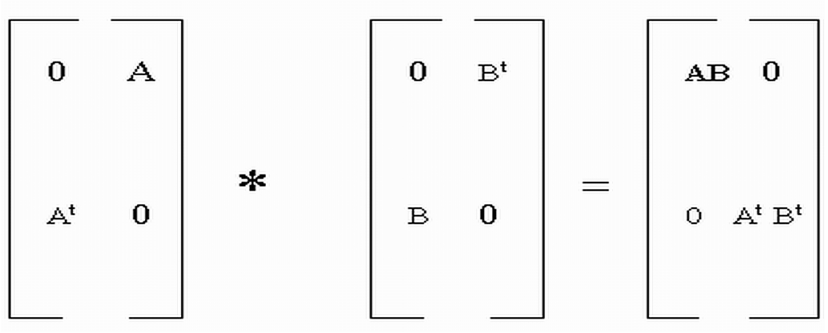


Figura 18. Descomposición de la multiplicación de matrices en términos de matrices simétricas.

Problemas duros

Por un problema *duro* se entiende aquel cuyo mejor algoritmo de solución conocido es caro en tiempo de ejecución.

Un ejemplo son las torres de Hanoi que toma tiempo exponencial: $2^n - 1$.

Se define un *algoritmo duro* como aquel que ejecuta en tiempo exponencial, esto es, en $O(c^n)$ para alguna constante $c > 1$.

La idea de adivinar la respuesta correcta a un problema o checar todas las posibles soluciones en paralelo, para determinar cuál es correcta, es llamada *no-determinismo*.

Un algoritmo que trabaja de esta manera es llamado un *algoritmo no-determinista*. Cualquier problema con un algoritmo que ejecuta en una máquina no-determinista en tiempo polinomial, se le da un nombre especial: Problema NP.

No todos los problemas que requieren tiempo exponencial en una computadora regular son NP. Por ejemplo, las torres de Hanoi no está en NP, ya que requiere $O(2^n)$ pasos imprimir el conjunto correcto de movimientos para n discos. Una máquina no-determinista no puede adivinar e imprimir la respuesta correcta en tiempo polinomial.

Vendedor viajero (1)

Entrada: Una gráfica dirigida completa G con distancias asociadas a cada arco en la gráfica.

Salida: El ciclo simple más corto que incluye cada vértice.

Ejemplo: (Ver gráfica) si el vendedor visita las ciudades en el orden ACEDBA, viajará una distancia total de 34. Una mejor ruta será ACBEDA, con costo 23. La mejor ruta en esta gráfica será ADCBEA, con costo 20.

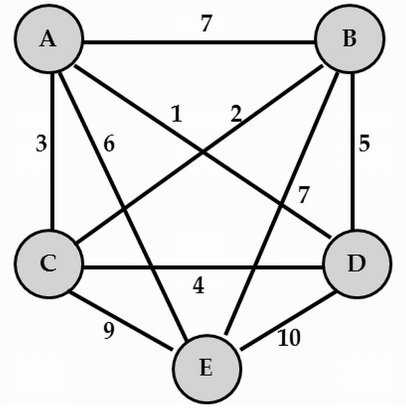


Figura 19. Ejemplo de gráfica para ilustrar el problema del vendedor viajero (1).

No podemos resolver este problema en tiempo polinomial con una computadora no-determinista. El problema es que, dado un ciclo candidato, aunque podemos checar rápidamente que la respuesta es un ciclo de la forma apropiada, no tenemos una forma fácil de saber si en efecto es el ciclo más corto.

Sin embargo, podemos resolver una variante de este problema, que está en la forma de un problema de decisión. Un *problema de decisión* es aquel cuya respuesta es sí o no.

Vendedor viajero (2)

Entrada: Una gráfica dirigida completa G con distancias asociadas a cada arco de la gráfica, y un entero *K*.

Salida: SI, si hay un ciclo simple con distancia total ≤ *K*, conteniendo cada vértice G, y NO en otro caso.

Podemos resolver esta versión del problema en tiempo polinomial en una computadora no-determinista. El algoritmo no-determinista simplemente checa todos los posibles subconjuntos de arcos en la gráfica en paralelo. Si cualquier subconjunto de arcos es un ciclo apropiado de longitud total menor o igual a *K*, la respuesta es *sí*; en otro caso es *no*.

El checar un subconjunto en particular es realizarlo en tiempo polinomial sumando las distancias de los arcos y verificando que los arcos forman un ciclo que visita cada vértice exactamente una vez. Desafortunadamente, hay $|E|!$ Subconjuntos qué checar, así que este algoritmo no puede ser convertido a un algoritmo de tiempo polinomial en una computadora regular.

Resulta que hay una gran colección de problemas con esta propiedad: conocemos algoritmos eficientes no-deterministas, pero no sabemos si hay algoritmos deterministas eficientes. Al mismo tiempo, no podemos probar que cualquiera de estos problemas no tienen algoritmos deterministas eficientes. Esta clase de problemas se llaman NP-completos (NP-*complete*).

Lo que es extraño y fascinante acerca de los problemas NP-completos es que si alguien alguna vez encuentra la solución a cualquiera de ellos en tiempo polinomial en una computadora regular, entonces

por una serie de reducciones, cualquier otro problema que está en NP también puede ser resuelto en tiempo polinomial en una computadora regular.

Un problema X es definido como NP-completo, si: X está en NP, y cualquier otro problema en NP puede ser reducido a X en tiempo polinomial.

La principal ventaja teórica de conocer que un problema Q_1 es NP-completo es que puede ser usado para mostrar que cualquier otro problema Q_2 es NP-completo, si existe una reducción polinomial de Q_1 en Q_2.

Existe una ventaja práctica en conocer que un problema es NP-completo. Consiste en que si una solución en tiempo polinomial puede ser encontrada para cualquier problema NP-completo, entonces una solución polinomial puede ser encontrada para todos estos problemas. La implicación es:

Como nadie ha encontrado aún tal solución, debe ser difícil o imposible de hacer, y el esfuerzo empleado en encontrar una solución polinomial para un problema NP-completo, puede ser considerado que se empleó para todos los problemas NP.

Al mostrar que un problema es NP-completo, es una indicación que los científicos más brillantes de la computación de las últimas décadas han intentado encontrar una solución polinomial, pero han fracasado.

Los problemas que son solucionables en tiempo polinomial en una computadora regular se dice que están en la clase P.

Podemos considerar todos los problemas solucionables en tiempo exponencial como una clase mayor de problemas.

La pregunta sin resolver más importante en Ciencias de la Computación teórica, es si P = NP.

Si por ejemplo, se encontrara un algoritmo polinomial para el problema del viajero, entonces todos los problemas NP serían solucionables en tiempo polinomial. Por otra parte, si lográramos probar que el problema del viajero tiene una cota inferior exponencial, esto mostraría que P ≠ NP.

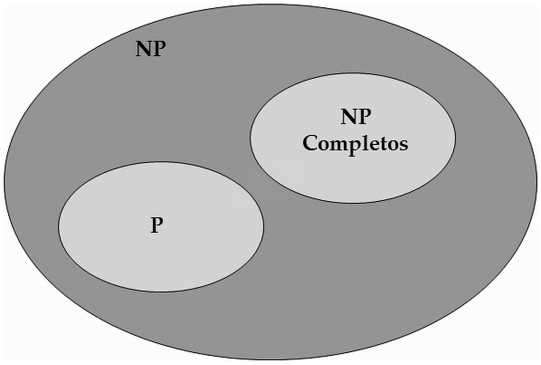


Figura 20. Relación entre las categorías P, NP completos y NP.

Solucionando problemas NP-completos

¿Qué hacer cuando se tiene que resolver un problema NP-completo?
Existen varias técnicas que pueden aplicarse:

1. Ejecutar sólo pequeñas instancias del problema.
2. Resolver una instancia especial del problema que no sea tan difícil de resolver.
3. Buscar una solución aproximada al problema. Por ejemplo, usando una heurística para resolver el problema. En el caso del viajero, esto podría traducirse en siempre proceder a la siguiente ciudad más cercana que no haya sido visitada.

Problemas imposibles

Dado un programa y una entrada particular, sería útil saber si ejecutando el programa en esa entrada resultaría en un ciclo infinito, sin ejecutar el programa.

Desafortunadamente, el problema de la terminación (*Halting problem*), no puede ser resuelto. Nunca existirá un programa de computadora que pueda determinar si otro programa se detendrá para todas las entradas.

No siempre es obvio el determinar si un programa terminará:

```
While (n > 1)
 If(odd(n))
      n = 3 * n + 1;
 else
      n = n / 2:
```

Este es un fragmento famoso de código. La secuencia de valores que es asignado a *n* por este código algunas veces es llamado la *secuencia de Collatz* para el valor de entrada *n*.

¿Este código se detiene para todos los valores de *n*?

Nadie sabe la respuesta aún. Cada entrada en la que se ha probado se detiene. ¿Pero siempre se detiene?

Las pruebas de computabilidad (la rama de las Ciencias de la Computación que estudia qué es imposible de hacer con una computadora), nos hacen creer que siempre podremos encontrar otro programa que no podamos analizar. Esto surge como resultado de que el *Halting problem* es irresoluble.

Incontabilidad (Uncountability)

Para probar que el problema de la terminación (*Halting problem*) es irresoluble, se probará primero que no todas las funciones pueden ser programadas. Esto resultará porque el número de programas es mucho menor que el número de posibles funciones.

Definición: Un conjunto se dice que es contable, si todo miembro del conjunto puede ser asignado unívocamente a un entero positivo. Un conjunto es no-contable si no es posible asignar cada miembro del conjunto a un entero positivo.

¿Es el conjunto de programas, contable o no-contable?

Un programa puede ser visto simplemente como una cadena de caracteres (incluyendo puntuación especial, espacios y saltos de línea).

Supongamos que el número de caracteres diferentes que puede aparecer en un programa es P. Si el número de cadenas es contable, entonces seguramente el número de programas también lo es.

Podemos asignar cadenas a posiciones enteras como sigue:

Se asigna la cadena vacía a la primera posición. Tomemos ahora todas las cadenas de un caracter, y asignémoslas a las siguientes P posiciones en orden alfabético. A continuación tomemos todas las cadenas de dos caracteres y asignémoslas a las siguientes P^2 posiciones. Las cadenas con 3 caracteres son asignadas de manera similar, luego las cadenas de longitud 4, y así sucesivamente. En esta forma, una cadena de cualquier longitud se le asigna un número entero.

Cualquier programa que simplemente es una cadena de longitud finita se le asigna un entero. De este modo, el conjunto de todos los programas es contable.

Consideremos ahora el conjunto de todas las posibles funciones. Tomemos primero las funciones que toman un entero positivo como entrada y genera un entero positivo como salida. A estas funciones se les llama *funciones enteras*.

Puede considerarse que todo lo que leen o escriben las computadoras puede codificarse como valores enteros, así el modelo simplista de funciones enteras cubre todos los posibles programas de cómputo.

Imaginemos cada función entera como una tabla con dos columnas y un número infinito de renglones. La primera columna lista los enteros positivos iniciando en 1. La segunda columna lista la salida correspondiente de la función.

¿Podemos asignar una función a cada posición? La respuesta es no, ya que siempre existe una forma de crear una función que no esté asignada a una posición.

Supongamos que una asignación incluyera todas las funciones. Podemos construir una nueva función que no haya sido asignada como sigue: $F_{\text{nueva}}(i) = F_i(i) + 1$. Así la nueva función debe ser diferente a cualquier función $F_i(i)$ al menos en la posición i.

El significado de esto es que no todas las funciones pueden ser asignadas a programas; deben existir funciones sin programas correspondientes.

El problema de la terminación es irresoluble

Supongamos que existiera una función *detiene* que puede resolver el problema de la terminación. Esta función regresaría a *verdad* si el programa de entrada con los datos entrada termina, y *falso* en otro caso.

```
Bool detiene(char * prog, char* input)
{
 if(prog se detiene con input) then
       return(TRUE)          ;
 else
       return(FALSE);
}
bool auto-detiene(char *prog){
 //regresa true si el programa se detiene cuando se da a sí
 // mismo como entrada
if(detiene(prog, prog))
       return(TRUE);
else
       return(FALSE);
}
void contrario(char *prog){
       if (auto-detiene(prog))
 while(TRUE); // entra en loop infinito
}
```

¿Qué pasa cuando la función *contrario* se ejecuta en sí misma?

Una posibilidad es que la llamada a *auto-detiene* regrese a *TRUE*; esto es, *auto-detiene* afirma que *contrario* se detendrá cuando se ejecute consigo misma. En este caso, *contrario* entra en un *loop* infinito (y por lo tanto, no se detiene). Por otra parte, si *auto-detiene* regresa a *FALSE*, entonces *detiene* dice que *contrario* no se detiene consigo misma, y *contrario* entonces regresa a *detiene*. Así *contrario* hace lo contrario de lo que *detiene* dice que hará.

La acción de *contrario* es lógicamente inconsistente con la suposición que *detiene* resuelve el problema de la terminación correctamente.

Así por contradicción, se ha probado que *detiene* no puede resolver el problema de la terminación, y por lo tanto, no existe un programa que resuelva este problema.

Un corolario del resultado probado es el siguiente:

El determinar el comportamiento de programas es irresoluble (de manera automática).

No es posible escribir un programa de cómputo que pueda checar para todos los programas de entrada, si una línea específica de código será ejecutada, cuando al programa se le dé una entrada específica.

Ejercicios

1. Dé tres ejemplos de problemas duros.
2. Programe el código de Collatz y muestre experimentalmente que de uno a la n que se desee, siempre se detiene. Analice el número de iteraciones que toma en cada caso y concluya si encuentra alguna regularidad.
3. Note que el proceso de reducción de un problema en otro, es similar a los métodos de transformadas que se aplican en ingeniería. Dé dos ejemplos del uso de transformadas en ingeniería y explique por qué es similar a un proceso de reducción.
4. Investigue cuál es la complejidad del algoritmo de Strassen para la multiplicación de matrices.
5. Investigue qué es la computación cuántica y sus implicaciones en relación a la complejidad de algoritmos.

6. Algoritmos selectos por categoría

Clasificación de algoritmos

Tipos de problemas estudiados en algoritmos:

- Ordenamiento
- Búsqueda
- Procesamiento de cadenas
- Recorrido de gráficas
- Combinatorios
- Geométricos
- Numéricos
- Otros

Podemos *clasificar a los algoritmos* por:

- Su eficiencia
- Técnica de diseño

Técnicas de diseño:

- Fuerza bruta (*brute force*)
- Divide y vencerás (*divide and conquer*)

- Decrementa y conquista (*decrease and conquer*)
- Transforma y conquista (*transform and conquer*)
- Balance de espacio y tiempo (*space-time tradeoff*)
- Codicioso (*greedy*)
- Programación dinámica (*dynamic programming*)
- Marcha atrás (*backtracking*)
- Ramifica y acota (*Branch and bound*).

Fuerza bruta

Es un enfoque directo para resolver un problema, usualmente basado en el enunciado del problema y la definición de los conceptos involucrados.

La fuerza implicada en la definición de la estrategia es de la computadora y no del intelecto de alguien. ¡Sólo hazlo! A menudo esta estrategia es más fácil de aplicar.

Aunque rara vez es fuente de algoritmos eficientes o inteligentes, este enfoque puede ser aplicable en algunos casos. Puede aplicarse a una gran variedad de problemas. Para algunos problemas importantes (ordenamiento, búsqueda, multiplicación de matrices, búsqueda de cadenas), da algunos algoritmos razonables de valor práctico.

El gasto de diseñar un algoritmo más eficiente puede no ser justificable, si sólo algunas instancias del problema necesitan ser resueltas, y si se resuelve en tiempos aceptables. Estos algoritmos pueden ser útiles para pequeñas instancias del problema. Provee una referencia para diseñar algoritmos más eficientes.

*Divide y vencerás (*divide and conquer*)*

Estos algoritmos trabajan de acuerdo al siguiente plan:

Una instancia del problema es dividida en varias más pequeñas de aproximadamente el mismo tamaño. Las pequeñas instancias son resueltas (típicamente de manera recursiva, aunque algunas veces un algoritmo diferente es usado cuando las instancias se vuelven sufi-

cientemente pequeñas). Si es necesario, las soluciones obtenidas de las pequeñas instancias son combinadas para obtener una solución del problema original. Es conveniente tener en mente que esta técnica es aplicable en la computación paralela.

Una instancia de tamaño n puede dividirse en varias instancias de tamaño n/b, con a de ellas que se necesitan resolver. (a, b, constantes, $b > 1$, $a \geq 1$). Si n es una potencia de b, obtenemos la siguiente relación de recurrencia para el tiempo de ejecución $T(n)$:

$$T(n) = a\ T(n/b) + f(n),$$

donde $f(n)$ es una función que da el tiempo empleado en dividir el problema y combinar las soluciones parciales. Esta recurrencia se llama la *recurrencia general de divide y vencerás*.

El análisis de eficiencia de muchos algoritmos se simplifica grandemente por el siguiente teorema:

> Si $f(n)\ \varepsilon\ \Theta(n^d)$, donde $d \geq 0$ en la ecuación de recurrencia general, entonces $T(n)\ \varepsilon$
> $\Theta(n^d)$ si $a < b^d$
> $\Theta(n^d \log n)$ si $a = b^d$
> $\Theta(n^{\log_b a})$ si $a > b^d$

Por ejemplo, la ecuación de recurrencia para el número de adiciones $A(n)$ en la suma de n números, por un algoritmo de divide y vencerás en entradas de tamaño $n = 2^k$ es:

$$A(n) = 2\ A(n/2) + 1$$

Así para este ejemplo, $a = 2$, $b = 2$ y $d = 0$, por lo cual $a > b^d$

$$A(n)\ \varepsilon\ \Theta\ (n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$$

Ejemplos de algoritmos con este enfoque: mergesort, quicksort, búsqueda binaria, recorridos de árboles binarios, multiplicación de enteros largos, multiplicación de matrices por el método de Strassen.

Decrementa y conquista

Esta técnica está basada en explotar la relación entre una solución a una instancia del problema, y una solución a una instancia menor del mismo problema. Una vez que tal relación está establecida, puede ser explotada ya sea de arriba hacia abajo (recursivamente) o de abajo hacia arriba (sin una recursión). Existen tres variaciones principales de esta técnica:

Decrementar por una constante
Decrementar por un factor constante
Decrementar por un tamaño variable

Ejemplo:

$$a^n = a^{(n\text{-}1)}\, a$$

mcd(m, n) = mcd(n, m mod n), donde mcd = máximo común divisor

Ejemplos de algoritmos que siguen esta técnica: ordenamiento por inserción, búsqueda en profundidad, búsqueda en amplitud, ordenamiento topológico, algoritmos para generar objetos combinatorios

Transforma y conquista

Esta técnica general trabaja como un procedimiento de dos etapas. Primero, en *la etapa de transformación*, la instancia del problema es modificada para ser, por una razón u otra, más sencilla de resolver. En la segunda, se resuelve.

Existen tres variantes principales de esta idea, que difieren en que transformamos una instancia:

1. Transformación a una instancia más simple o más conveniente del mismo problema –le llamamos a esto *simplificación de la instancia*.

2. Transformación a una representación diferente de la misma instancia –le llamamos a esto– *cambio de representación*.
3. Transformación a una instancia de un problema diferente para el cual ya existe un algoritmo –le llamamos a esto *reducción del problema*.

Ejemplos de algoritmos que se basan en las ideas de transforma y conquista:

Eliminación Gaussiana, árboles de búsqueda balanceados, regla de Horner para evaluar polinomios.

Balance de espacio y tiempo (space-time tradeoff)

Un ejemplo es el cálculo de una función en muchos puntos de su dominio. Si el tiempo es lo más valioso, podemos precomputar los valores de una función y guardarlos en una tabla.

En términos más generales, la idea es preprocesar la entrada del problema, en parte o en todo, y guardar la información adicional obtenida para acelerar la resolución de problemas posteriormente. Le llamamos a este enfoque: *realce de la entrada*.

Ejemplos de algoritmos que aplican esta idea son:

Horspool, para correspondencia de cadenas.

La transformada rápida de Fourier, teniendo precalculados los valores de las funciones senos y cosenos.

Los otros tipos de técnicas que explotan los balances de espacio y tiempo simplemente usan espacio extra para facilitar un acceso más flexible y rápido a los datos.

Otro ejemplo, que además realiza un preprocesamiento a los datos es el método de Hashing.

Otra técnica de diseño que está relacionada con la idea de balance de espacio y tiempo es: la *programación dinámica*. Esta estrategia está basada en registrar soluciones de subproblemas que se traslapan de un problema en una tabla, a partir de la cual, la solución a un problema es luego obtenida.

En algunas ocasiones el espacio y el tiempo no tienen por qué ser antagónicos. Por ejemplo, en el recorrido de gráficas poco densas se puede minimizar el espacio requerido y a la vez el tiempo de procesamiento.

En otros casos, la reducción de espacio, puede ser el objetivo en sí mismo, como en la *compresión de datos*.

Programación dinámica (dynamic programming)

Es una técnica que fue inventada por el matemático Richard Bellman en 1950, como un método general para optimizar procesos de decisión multietapa.

Sirve para resolver problemas definidos mediante subproblemas que se traslapan. Típicamente, estos problemas surgen de una relación de recurrencia.

Un ejemplo es la sucesión de Fibonacci.

Otros ejemplos de algoritmos que usan la programación dinámica son: cómputo del coeficiente binomial, los algoritmos de Warshall y de Floyd, árboles de búsqueda binarios y el problema del saco.

Técnicas codiciosas (greedy)

Problema del cambio. Dar cambio para una cantidad específica n con el menor número de monedas o billetes de las denominaciones $d1 > d2 \ldots > dm$. Por ejemplo, $d1 = 20$ pesos, $d2 = 10$ pesos, $d3 = 5$ pesos y $d4 = 1$ peso.

¿Cómo daría cambio de 48 pesos?

El enfoque codicioso sugiere construir una solución a través de una secuencia de pasos, cada uno expandiendo una solución parcialmente construida, obtenida hasta ese momento, en que la solución completa del problema sea obtenida.

En cada paso, y este es el punto central de esta técnica, la elección hecha debe ser:

Factible (tiene que satisfacer las restricciones del problema)

Localmente óptima (tiene que ser la mejor elección local entre todas las posibles elecciones en ese paso).

Irrevocable (una vez hecha, no puede ser cambiada en pasos subsecuentes del algoritmo).

En el caso del problema del cambio, una solución es dar primero billetes o monedas de la mayor denominación y ya que no es posible, continuar con la de la siguiente denominación y así sucesivamente hasta terminar. Estos requerimientos explican el nombre de la técnica. En cada paso, sugiere un agarre codicioso de la mejor alternativa, esperando que la secuencia de elecciones óptimas locales, conducirá a una solución óptima global de todo el problema.

Ejemplos de algoritmos codiciosos:

Prim y Kruskal para el MST. Dijkstra para el problema del camino más corto desde una fuente. Y los códigos de Huffman.

Algoritmos de correspondencia de patrones (*Pattern-matching algorithms*)

La correspondencia de patrones es una parte integral de muchos problemas en edición de textos, recuperación de datos y manipulación simbólica. En un problema típico de correspondencia de cadenas (de caracteres) se nos da una cadena de texto x y un conjunto de cadenas patrones $\{y_1, y_2, \ldots\}$. El problema es, ya sea localizar una ocurrencia o todas las ocurrencias de las cadenas patrón en x.

Comparación de cadenas

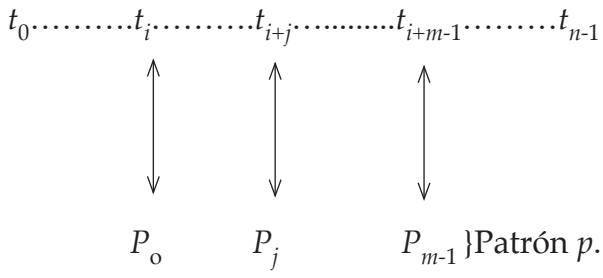
La comparación de cadenas se aplica en: procesamiento de textos, búsqueda, bioinformática.

Comparación de cadenas por fuerza bruta

Encontrar un patrón de m caracteres en una cadena de n caracteres llamada el texto ($m < n$).

Esto es, se trata de encontrar una subcadena de texto que corresponde con el patrón.

Queremos encontrar el entero i, que es el índice del caracter más a la izquierda, de la primera subcadena que se encuentre en el texto, tal que:



Algoritmo busca cadena fuerza bruta (t[0..n - 1)], p[0..m - 1])

```
for(i=0 to n-m do
    j       =0
            while (j<m) and (p[j]=T[i+j]) do
                    j=j+1
            if j=m return(i)
return(–1)
```

Ejemplo:

Encontrar el patrón: ROSA

En el texto:

EN EL PATIO HAY ROPA Y ROSAS

```
ROSA
   ROSA
          RO…
                 ROSA
```

Orden del algoritmo: O(nm)

Algoritmo de Horspool

La mayoría de los algoritmos de reconocimiento de cadenas que superan al de fuerza bruta, explotan la idea del realce del patrón de entrada y del balance *espacio-tiempo*: preprocesan el patrón para obtener información sobre él, almacenan esta información en una tabla, y luego usan esta información durante la búsqueda del patrón en un texto. Esta es exactamente la idea detrás de los dos algoritmos mejor conocidos de este tipo: Knuth-Morris-Pratt y el de Boyer-Moore.

El algoritmo de Boyer-Moore comienza alineando el patrón con los caracteres iniciales del texto; si el primer intento falla, traslada el patrón a la derecha. El algoritmo hace las comparaciones de derecha a izquierda, comenzando con el último caracter en el patrón.

Veremos una idea simplificada del algoritmo de Boyer-Moore, conocido como algoritmo de Horspool, que no por eso es menos eficiente en cadenas aleatorias.

Veamos un ejemplo:

$S_0 \ldots \quad c \ldots\ldots s_{n-1}$
ROSALBA

Comenzamos con la última A del patrón, moviéndonos de derecha a izquierda, comparamos las partes correspondientes de caracteres en el patrón y el texto.

Si encontramos una discrepancia, necesitamos mover el patrón a la derecha. Claramente, nos gustaría hacer el desplazamiento tan grande como sea posible, sin omitir una subcadena correspondiente en el texto. El algoritmo determina el tamaño del desplazamiento viendo el caracter *c* del texto que estaba alineado con el último caracter del patrón.

Podemos tener cuatro posibilidades:

Caso 1: Si no está el caracter *c* en nuestro patrón (*c* es la letra *T* en el siguiente ejemplo), podemos trasladar seguramente el patrón en su longitud total (si trasladamos menos, algún caracter del patrón estará alineado con el caracter *c* del patrón, que sabemos que no está en el patrón).

S_0 … T …… s_{n-1}

ROSALBA
≠

ROSALBA

Caso 2: Hay ocurrencias del caracter *c* en el patrón pero no es el último (*c* es la letra S en nuestro ejemplo, el desplazamiento debe alinear la ocurrencia más a la derecha de *c* en el patrón, con la *c* en el texto):

S_0 … S …… s_{n-1}

ROSALBA
≠

ROSALBA

Caso 3: Si *c* es el último caracter en el patrón, pero no hay *c* entre los otros *m* - 1 caracteres, el desplazamiento es similar al caso 1: el patrón debe ser trasladado por la longitud total del patrón: m.

S_0 … MBA …… s_{n-1}

COSA
≠ =

COSA

Caso 4: Finalmente, si *c* es el último caracter en el patrón y hay otras *c* entre los primeros *m* - 1 caracteres, el desplazamiento debe ser similar al caso 2: la ocurrencia más a la derecha de *c* entre los primeros *m* - 1 caracteres en el patrón debe ser alineada con la *c* del texto.

S_0 … SA …… s_{n-1}

ROSALBA
≠ =

ROSALBA

No es necesario analizar todos los caracteres del patrón en cada intento. Podemos precalcular los tamaños del desplazamiento y

guardarlos en una tabla. La tabla será indexada por todos los posibles caracteres que puedan ser encontrados en el texto.

$t(c) = m$, la longitud del patrón, si c no está dentro de los primeros m - 1 caracteres del patrón.

La distancia de la c más a la derecha dentro de los primeros m - 1 caracteres del patrón con su último caracter.

Por ejemplo, para el patrón BARBER, todas las entradas de la tabla serán 6, excepto para las entradas E, B, R y A, que serán 1, 2, 3 y 4, respectivamente.

A continuación un algoritmo simple para calcular las entradas de la tabla. Se inicializan todas las entradas con m, la longitud del patrón, y escaneamos el patrón de izquierda a derecha, repitiendo lo siguiente m - 1 veces: para el caracter j-ésimo en el patrón ($0 \leq j \leq m - 2$), sobre escriba su entrada en la tabla con $m - 1 - j$, que es la distancia del caracter al extremo derecho del patrón.

```
Algoritmo Tabla_desplaza(P[0..m-1])
//Llena la tabla de desplazamientos usados por el algoritmo de //Horspool.
//Entrada: Patrón P[0..m-1] y un alfabeto de posibles caracteres
//Salida: Tabla[0..size-1] indexada por los caracteres del alfabeto y llenado con los tamaños de desplazamiento dados por la fórmula.
Inicializa todo los elementos de Tabla con m
For j=0 to m-2 do Tabla[P[j]]=m-1-j
Return Tabla.
```

A continuación resumimos el algoritmo como sigue:

Paso 1. Para un patrón dado de longitud m, y para el alfabeto usado en el patrón y el texto, construya la tabla de desplazamientos.

Paso 2. Alinee el patrón con el comienzo del texto.

Paso 3. Repita lo siguiente hasta que una subcadena en correspondencia sea encontrada o el patrón alcance más allá del último caracter en el texto. Comenzando con el último caracter en el patrón, compare los caracteres correspondientes en el patrón y el texto hasta que los m

caracteres sean correspondidos (en ese caso detener), o un par de discrepancias sea encontrado. En este último caso, recupere la entrada $t(c)$ de la tabla de desplazamientos, donde c es el caracter del texto actualmente alineado con el último caracter del patrón, y traslade el patrón en $t(c)$ caracteres a la derecha sobre el texto.

```
Algoritmo Correspondencia_Horspool(P[0..m-1], T[0..n-1])
//Implementa al algoritmo de Horspool para correspondencia de cadenas
//Entrada: Patrón P[0..m-1] y texto T[0..n-1]
//Salida: El índice del extremo izquierdo de la primera subcadena en //correspondencia, o -1 si
no hay correspondencias.
Tabla_desplaza(P[0..m-1]) //genera tabla de desplazamientos
i=m-1 //posición del extremo derecho del patrón
while i<= n-1 do
 k=0 //número de caracteres correspondidos
 while k<= m-1 and P[m-1-k]== T[i-k] do
 k=k+1
 if k == m
 return i-m+1
 else i=i+Tabla[T[i]]
return -1
```

Ejemplo: Búsqueda del patrón BARBER

caracter c	**A**	**B**	**C**	**D**	**E**	**F**	**…**	**R**	**…**	**Z**	**-**
desplaza-miento *t(c)*	4	2	6	6	1	6	6	3	6	6	6

Ejemplo:

```
KIM SAW ME IN A BARBERSHOP
BARBER
      BARBER
       BARBER
             BARBER
               BARBER
                  BARBER
```

Algoritmos de compresión de información

Árboles y códigos de Huffman

Supongamos que queremos codificar un texto compuesto de caracteres de un alfabeto de n caracteres, asignando a cada uno de los caracteres del texto una secuencia de bits llamada *palabra código* (*codeword*).

Por ejemplo, podemos usar una *codificación de longitud fija* que asigna a cada caracter una cadena m de caracteres de la misma longitud ($m \geq \log_2 n$).

Esto es lo que hace el código ASCII de 7 bits.

Una forma de obtener un esquema de codificación que dé cadenas de caracteres más cortas en promedio, se basa en la vieja idea de asignar palabras código más cortas a los caracteres más frecuentes y palabras código más largas a los caracteres menos frecuentes.

Esta idea fue usada a mediados del siglo XIX en el código telegráfico inventado por Samuel Morse. Por ejemplo, (e (.), a(.-), q(- - .-), z(- - ..)).

El usar una codificación de longitud variable, que asigna palabras código de diferentes longitudes a distintos caracteres, introduce un problema que la codificación de longitud fija no tiene.

¿Cómo podemos saber cuántos bits del texto codificado representan el primero o el i-ésimo caracter?

Para evitar esta complicación podemos limitarnos a códigos de prefijo libre (o *prefix*). En este caso, ninguna palabra código es un prefijo de una palabra código de otro caracter.

De esta manera, podemos examinar uno de cada bits hasta que encontremos el primer grupo de bits que es una palabra código de algún caracter, reemplazar estos bits por este caracter, y repetir esta operación hasta que el fin de la cadena de bits sea alcanzado.

Si queremos crear un código binario prefijo, es natural asociar los caracteres del alfabeto con las hojas de un árbol binario en el cual el lado izquierdo se etiqueta con 0 y el lado derecho con 1 (o viceversa).

La palabra código de un caracter puede ser obtenida recordando las etiquetas en el camino simple de la raíz a la hoja del caracter. Como no hay un camino simple de una hoja que continúe a otra hoja,

ninguna palabra clave puede ser prefijo de otra palabra clave. Esto es, cualquier árbol de este tipo da una codificación prefija.

¿Cómo podemos construir un árbol que asigne cadenas de bits más cortas a caracteres de mayor frecuencia, y cadenas más largas a caracteres con menor frecuencia? Puede hacerse con el algoritmo inventado por David Huffman (1952).

Algoritmo de Huffman

Paso 1. Inicialice *n* árboles de un nodo y etiquételos con los caracteres del alfabeto. Grabe la frecuencia de cada caracter en la raíz del árbol para indicar el peso del árbol.

Paso 2. Repita la siguiente operación hasta que un solo árbol sea obtenido. Encuentre dos árboles con los menores pesos. Hágalos los subárboles izquierdo y derecho de un nuevo árbol, y grabe la suma de sus pesos en la raíz del nuevo árbol como su peso.

El árbol construido con el algoritmo descrito se llama *Árbol de Huffman* y define un *código de Huffman*.

Ejemplo: Considere un alfabeto de 5 caracteres [*A*, *B*, *C*, *D*, -] con las siguientes probabilidades de ocurrencia:

Caracter	***A***	***B***	***C***	***D***	**-**
Probabilidad	0.35	0.1	0.2	0.2	0.15

La construcción del árbol para estos datos se muestra en la figura anexa. Las palabras código resultantes son:

Caracter	***A***	***B***	***C***	***D***	**-**
Probabilidad	0.35	0.1	0.2	0.2	0.15
Código	11	100	00	01	101

Así, *DAD* se codifica como 011101, y 10011011011101 se decodifica como *BAD-AD*.

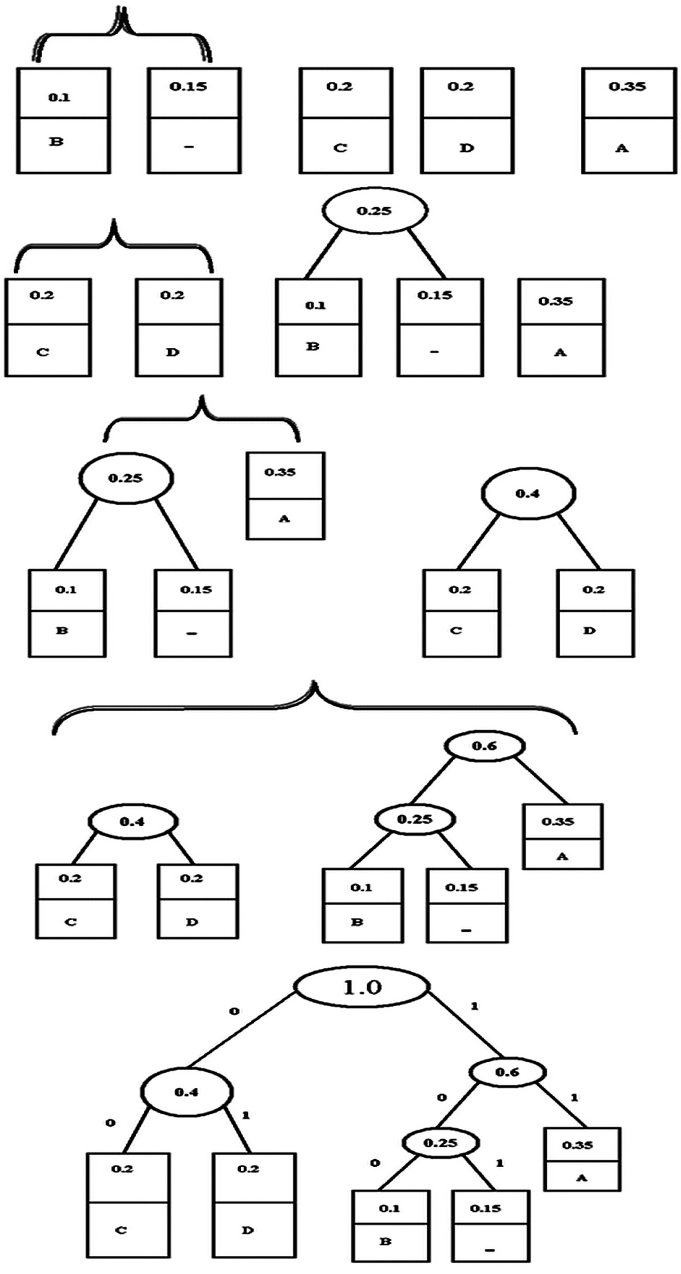


Figura 21. Ejemplo de construcción de un árbol de Huffman.

A partir de las probabilidades de ocurrencia y la longitud en bits de los códigos, es posible obtener el número esperado de bits por caracter en este código:

$$\sum longitud\_en\_bits * probabilidad$$

$$2*0.35 + 3*0.1 + 2*0.2 + 2*0.2 + 3*0.15 = 2.25$$

Si hubiéramos usado una codificación de longitud fija para el mismo alfabeto, hubiéramos tenido que usar al menos 3 bits por caracter. Así para este ejemplo, obtenemos una tasa de compresión de: $(3 - 2.25)/3 = 0.25$. En otras palabras, esperaríamos que una codificación de Huffman del texto usara 25% menos memoria que un esquema de codificación de longitud fija.

La codificación de Huffman es uno de los métodos de compresión más importantes. Además de su simplicidad y versatilidad, genera una codificación óptima, de mínima longitud. Esto, siempre y cuando las probabilidades de ocurrencia de los caracteres sean independientes y conocidas de antemano.

La versión más simple de la codificación de Huffman, requiere un escaneo previo del texto para contar y obtener las frecuencias de ocurrencia de los caracteres. Luego estas frecuencias son usadas para construir un árbol de Huffman y para codificar el texto. Este esquema requiere, sin embargo, que se incluya la información del árbol de codificación con el texto codificado, para hacer posible la decodificación.

Problemas de geometría computacional

Problemas en un conjunto finito de puntos en el plano

Los problemas a continuación además de su interés teórico, aparecen en dos importantes áreas aplicadas:

- Geometría computacional
- Investigación de operaciones

El problema del par más cercano

Consiste en encontrar en un conjunto de n puntos los dos puntos más cercanos.

El enfoque de fuerza bruta consiste en:

1. Calcule la distancia entre todos los pares y encuentre el par con la distancia más cercana.
2. Para no calcular la distancia entre el mismo par de puntos dos veces, consideramos sólo los pares de puntos (Pi, Pj), para los cuales $i < j$.

Si $Pi(x_i, y_i)$ y $Pj(x_j, y_j)$ son un par de puntos, su distancia euclideana está dada por: $d(P_i, P_j) = \sqrt{((x_i - x_j)^2 + (y_i - y_j)^2)}$.

```
Algoritmo Puntos-más-cercanos(P)
dmin = maxint
For i=1 to n-1
 For j=i+1 to n do
 d= √((xi-xj)^2 + (yi-yj)^2)
 if d < dmin
 dmin=d; index1=i; index2=j;
return index1, index2
```

Existe una forma en que las raíces cuadradas pueden ser evitadas. El truco consiste en notar que en lugar de comparar las raíces cuadradas podemos comparar $((xi - xj)^2 + (yi - yj)^2)$ directamente. Esto se debe a que mientras menor sea el número del que sacamos la raíz, menor es la raíz cuadrada misma, esto es, la función cuadrada es estrictamente creciente.

Asi en el algoritmo, en lugar de: $\sqrt{((x_i - x_j)^2 + (y_i - y_j)^2)}$, podemos usar: $((x_i - x_j)^2 + (y_i - y_j)^2)$.

La operación básica del algoritmo sería elevar al cuadrado.

El número de operaciones sería:

$$C(n) = O(n^2)$$

Problema de la cubierta convexa

Definición. Un conjunto de puntos en el plano es llamado *convexo* si para cualquier par de puntos P y Q en el conjunto, el segmento que une los puntos pertenece al conjunto.

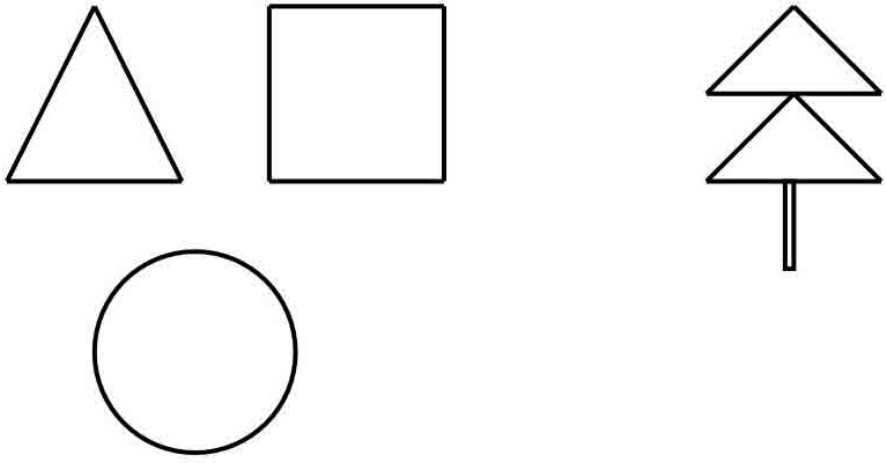


Figura 22. (a) Figuras convexas. (b) Figuras no convexas

Intuitivamente la cubierta convexa (*convex hull*) de un conjunto de n puntos en el plano, es el menor polígono convexo que los contiene.

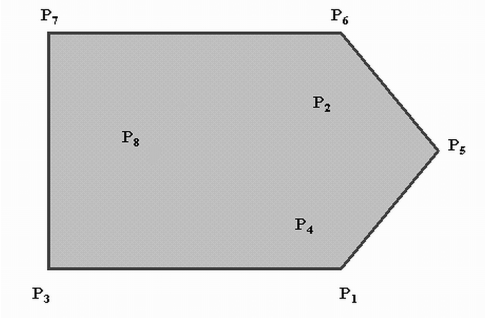


Figura 23. Ejemplo de un conjunto de puntos y su cubierta convexa. En este caso, formado por las puntos: P_1, P_3, P_7, P_6 y P_5.

Teorema. La cubierta convexa de cualquier conjunto S de $n > 2$ puntos (no todos en la misma línea), es el polígono convexo con los vértices en algunos de los puntos de S.

El problema de la cubierta convexa consiste en construir la cubierta convexa para un conjunto S de n puntos. Para resolverlo necesitamos encontrar los puntos que servirán como los vértices del polígono en cuestión. Estos puntos se llaman *puntos extremos*.

Los puntos extremos tienen propiedades especiales y son usados por el método simplex para resolver problemas de optimización de programación lineal.

Para encontrar los puntos se usa la siguiente propiedad: Un segmento que conecta dos puntos es parte de la cubierta convexa si y sólo si todos los otros puntos en el conjunto se encuentran en el mismo lado de la línea recta a través de los dos puntos.

Repitiendo esta prueba para cada par de puntos de una lista de segmentos de línea que forman la frontera de la cubierta convexa.

La línea que pasa por los puntos (x_1, y_1), (x_2, y_2) tiene por ecuación:

$$a\,\mathrm{x} + b\,\mathrm{y} + c = 0$$

La línea divide el plano en dos medios planos, unos son los puntos que $a\,\mathrm{x} + b\,\mathrm{y} + c > 0$, y los otros donde $a\,\mathrm{x} + b\,\mathrm{y} + c < 0$.

Así, para checar si ciertos puntos caen en el mismo lado de la línea, podemos checar si la expresión $a\,\mathrm{x} + b\,\mathrm{y} + c$ tiene el mismo signo en cada uno de estos puntos.

¿Cuál es la eficiencia de este algoritmo? Es $O(n^3)$, para cada $n(n - 1)/2$ pares de puntos distintos, necesitamos encontrar el signo de $a\,\mathrm{x} + b\,\mathrm{y} + c$ para cada uno de los otros $n - 2$ puntos.

Problemas de análisis numérico

Eliminación Gaussiana

$$\begin{aligned} a_{11}\,\mathrm{x} + a_{12}\,\mathrm{y} &= b_1 \\ a_{21}\,\mathrm{x} + a_{22}\,\mathrm{y} &= b_2 \end{aligned}$$

El método estándar para encontrar la solución es:

Usar cualquiera de las ecuaciones para expresar una de las variables como una función de la otra y luego sustituir el resultado en la otra ecuación, lo que da una ecuación lineal que es usada para encontrar el valor de la segunda variable.

En muchas aplicaciones necesitamos resolver un sistema de n ecuaciones en n incógnitas.

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

La idea de la eliminación Gaussiana es transformar el sistema en uno equivalente (con la misma solución que el original), con una matriz triangular superior de coeficientes y con ceros debajo de la diagonal.

$$\begin{aligned} a'_{11}\, x_1 + a'_{12}\, x_2 + \ldots + a'_{1n}\, x_n &= b_{1'} \\ a'_{22}\, x_2 + \ldots + a'_{2n}\, x_n &= b_{2'} \\ &\vdots \\ a'_{nn}\, x_n &= b_n' \end{aligned}$$

En notación matricial:

$$\boldsymbol{Ax = b => A' x = b'}$$

¿Por qué es mejor el sistema triangular?

Podemos obtener el valor de x_n de la última ecuación y luego sustituir este valor en la ecuación anterior para obtener $x_{n\text{-}1}$ y así sucesivamente hasta obtener x_1.

Para llegar a un sistema triangular se pueden usar las siguientes operaciones elementales:

Intercambiar 2 ecuaciones del sistema

Sustituir una ecuación con un múltiplo no cero de ella

Reemplazar una ecuación con la suma o diferencia de esta ecuación y algún múltiplo de otra ecuación.

Primero usamos a_{11} como pivote para hacer todos los coeficientes de x_1, ceros en las ecuaciones debajo de la primera.

Específicamente, reemplazamos la segunda ecuación con la diferencia entre ella y la primera ecuación multiplicada por a_{21} / a_{11} para obtener una ecuación con un coeficiente de cero para x_1.

Hacemos lo mismo con la ecuación tercera, cuarta y finalmente la ecuación n, con los múltiplos a_{31} / a_{11}, a_{41} / a_{11},.... a_{n1} / a_{11} de la primera ecuación. Lo que hace que todos los coeficientes de x_1 debajo de la primera ecuación sean cero.

Luego eliminamos los coeficientes de x_2 sustrayendo un múltiplo apropiado de la segunda ecuación de cada ecuación debajo de la segunda.

Repitiendo esta eliminación por cada una de las primeras n - 1 variables, nos lleva finalmente a un sistema con una matriz de coeficiente triangular superior.

Ejemplo:

$$2_{x1} - x_2 + x_3 = 1$$
$$4_{x1} + x_2 - x_3 = 5$$
$$_{x1} + x_2 + x_3 = 0$$

$$\begin{pmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 3/2 & ½ & -1/2 \end{pmatrix}$$ Renglón 2 - 4/2 renglón 1
Renglón 3 - 1/2 renglón 1

$$\begin{pmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{pmatrix}$$ Renglón 3 - 1/2 renglón 2

Ahora obtenemos la solución por sustitución hacia atrás:

$$x_3 = (-2)/2 = -1\ ,\ x_2 = (3-(-3)x_3)/3 = 0 \text{ y } x_1 = (1- x_3 + x_2)/2 = 1$$

Pseudo código:

```
For i=1 to n do A[i, n+1] = b[i] //Se aumenta la matriz
    For i=1 to n-1 do
            For j= i +1 to n do
                    For k= i to n+1 do
                            A[ j, k] =A[j , k] –A[i, k] * A[j , i]/A[i , i]
```

En el algoritmo: a(i, i) $\neq 0$

¿Cuál es el orden de la eliminación Gaussiana? Es cúbica.

Algoritmos para generar objetos combinatorios

Los tipos más importantes de objetos combinatorios son:

- permutaciones
- combinaciones
- subconjuntos de un conjunto

A continuación veremos algoritmos para generar este tipo de objetos combinatorios y algunas aplicaciones.

Generación de permutaciones

Por simplicidad, supondremos que el conjunto base del cual se harán las permutaciones es el conjunto de enteros de 1 a n.

¿Qué nos sugiere el enfoque de decrementar en uno para el problema de generar todas las n! permutaciones?

En este caso generaríamos las (n - 1)! permutaciones. Suponiendo que este problema más pequeño está resuelto, podemos obtener una

solución para el problema más grande, insertando n en cada una de las n posibles posiciones entre los elementos de cada permutación de n - 1 elementos.

En este caso se va insertando n en 1, 2…, (n - 1) de derecha a izquierda, y luego cambiando de dirección cada vez que una nueva permutación de {1…, n - 1} necesite ser procesada.

Ejemplo para $n = 3$, $3! = 6$.

Inicio: 1

Insertamos 2: 12 21 (de derecha a izquierda)

Insertamos 3: 123 132 312 (de derecha a izquierda)

321 231 213 (de izquierda a derecha)

La ventaja de este orden de generación de permutaciones es que satisface el *requerimiento del mínimo cambio*: Cada permutación puede ser obtenida de su predecesor inmediato intercambiando sólo dos elementos.

Esto es útil en aplicaciones como el del vendedor viajero.

Es posible obtener el mismo ordenamiento de las permutaciones de n elementos, sin explícitamente generar permutaciones para valores más pequeños de n. Esto puede hacerse asociando una dirección con cada componente k en una permutación. Indicamos tal dirección por una pequeña flecha que se escribe sobre la componente en cuestión.

Se dice que *la componente k es móvil en una permutación* marcada con flechas, si su flecha apunta a un número más pequeño adyacente a él. Por ejemplo:

$\rightarrow \leftarrow \rightarrow \leftarrow$

3 2 4 1.

3 y 4 son móviles pero 2 y 1 no.

Usando la noción de un elemento móvil, tenemos el algoritmo de Johnson-Trotter:

Algoritmo JohnsonTrotter(n)

//entrada: un entero positivo n

//salida: Una lista de todas las permutaciones de {1… n}

Inicializamos la primera permutación con 1 2… *n* (flechas apuntando a la izquierda)

While exista un entero móvil *k do*

Encuentre el entero móvil *k* más grande
Intercambie *k* y el entero adyacente que está siendo apuntado
Invierta la dirección de todos los enteros que son más grandes que *k*

Este algoritmo es uno de los más eficientes para generar permutaciones. Puede ser implementando en tiempo proporcional al número de permutaciones, es decir $\Theta(n!)$.

Ejemplo:

← ← ←	← ← ←	← ← ←	→ ← ←	← → ←	← ← →
1 2 **3**	1 **3** 2	3 1 **2**	**3** 2 1	2 **3** 1	2 1 3

Problema de la asignación

Hay *n* personas para ser asignadas a *n* trabajos, una persona por trabajo. (Cada persona es asignada a exactamente un trabajo, y cada trabajo es asignado exactamente a una persona).

El costo que llevaría asignar a la persona *i* al trabajo *j* es una cantidad conocida *C*(*i*, *j*), para cada par *i*, *j* = 1,... *n*. El problema es encontrar una asignación con el menor costo total.

Una pequeña instancia del problema se presenta a continuación, donde las entradas de la tabla representan los costos de asignación *C*(*i*, *j*):

	Trabajo 1	*Trabajo 2*	*Trabajo 3*	*Trabajo 4*
Persona 1	9	2	7	8
Persona 2	6	4	3	7
Persona 3	5	8	1	8
Persona 4	7	6	9	4

Una instancia del problema de la asignación está completamente especificada por su matriz de costo *C*.

En términos de la matriz, el problema consiste en seleccionar un elemento de cada renglón, para que todos los elementos estén en columnas diferentes, y la suma total de los elementos seleccionados sea la menor posible.

Como no hay solución directa aparente, un enfoque de solución sería emplear una búsqueda exhaustiva.

Podemos describir soluciones factibles al problema de la asignación como *n*-tuplas (j_1,... j_n), en donde la componente i-ésima, $i = 1... n$, indica la columna del elemento seleccionado en el renglón *i*. (es decir, el número de trabajo asignado a la persona *i*).

Por ejemplo, (2, 3, 4, 1) indica una asignación factible de la persona 1 al trabajo 2, la persona 2 al trabajo 3, la persona 3 al trabajo 4, y la persona 4 al trabajo 1.

Los requerimientos del problema de la asignación indican que hay una correspondencia uno a uno entre asignaciones factibles y permutaciones de los primeros *n* enteros.

Así, un enfoque exhaustivo requerirá generar todas las permutaciones de los enteros 1, 2,... *n*, calcular el costo total de cada asignación, sumando los elementos correspondientes de la matriz de costos, y finalmente seleccionando la asignación con la menor suma.

Ejemplo:
4! = 4*3*2*1 = 24

Permutación	*Costo*
(1, 2, 3, 4)	9 + 4 + 3 + 4 = 20
(1, 2, 4, 3)	9 + 4 + 8 + 9 = 30
(1, 3, 2, 4)	9 + 3 + 8 + 4 = 24
(1, 3, 4, 2)	9 + 3 + 8 + 6 = 26
(1, 4, 2, 3)	9 + 7 + 8 + 9 = 33
(1, 4, 3, 2)	9 + 7 + 1 + 6 = 23
(2, 1, 3, 4)	2 + 6 + 1 + 4 = 13 ÓPTIMA
(2, 1, 4, 3)	2 + 6 + 8 + 9 = 25
(2, 3, 1, 4)	2 + 3 + 5 + 4 = 14

Permutación	*Costo*
(2, 3, 4, 1)	2 + 3 + 8 + 1 = 14
(2, 4, 1, 3)	2 + 7 + 5 + 9 = 23
(2, 4, 3, 1)	2 + 7 + 1 + 7 = 17
(3, 1, 2, 4)	7 + 6 + 8 + 4 = 25
(3, 1, 4, 2)	7 + 6 + 8 + 6 = 27
(3, 2, 1, 4)	7 + 4 + 5 + 4 = 20
(3, 2, 4, 1)	7 + 4 + 8 + 7 = 26
(3, 4, 1, 2)	7 + 7 + 5 + 6 = 25
(3, 4, 2, 1)	7 + 7 + 8 + 7 = 29
(4, 1, 2, 3)	8 + 1 + 8 + 9 = 26
(4, 1, 3, 2)	8 + 6 + 1 + 6 = 21
(4, 2, 1, 3)	8 + 4 + 5 + 9 = 26
(4, 2, 3, 1)	8 + 4 + 1 + 7 = 20
(4, 3, 1, 2)	8 + 3 + 5 + 6 = 22
(4, 3, 2, 1)	8 + 3 + 8 + 7 = 26

Como el número de permutaciones a considerar en el caso general es $n!$, la búsqueda exhaustiva es impráctica excepto para pequeñas instancias del problema.

Para este problema existe un algoritmo más eficiente conocido como el *método húngaro*. El algoritmo puede consultarse en [5].

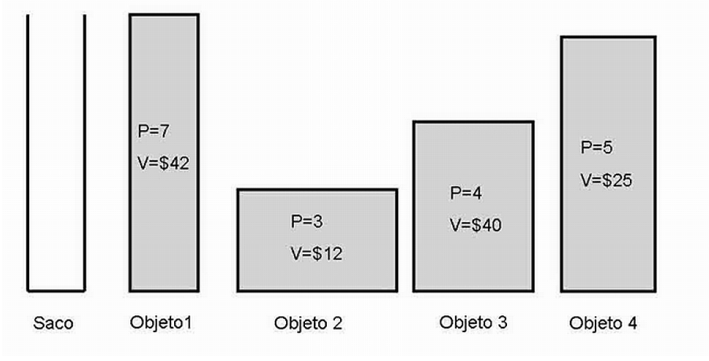
Problema del saco (Knapsack)

Dados n artículos de pesos conocidos $w_1 ... w_n$, y de valores $v_1 ... v_n$, y un saco de capacidad W, encuentre el subconjunto más valioso de artículos que caben en el saco. (Ejemplos: En compañías de paquetería implicaría decidir qué artículo es más redituable enviar primero sin rebasar los límites de capacidad).

La solución de búsqueda exhaustiva considera todos los subconjuntos de n artículos, calcula el peso total de cada subconjunto para identificar los factibles (los que su peso no exceda la capacidad del saco), y encontrando el subconjunto del valor más grande entre ellos.

Como el número de subconjuntos de n elementos es 2^n, la búsqueda exhaustiva lleva a un algoritmo de $\Omega(2^n)$.

El problema del viajero y del saco son muy ineficientes en cada entrada, y son dos de los ejemplos más conocidos de problemas NP. No se conocen aún algoritmos de tiempo polinomial para estos problemas.



Peso límite = 10

Subconjunto	*Peso total*	*Valor total*
Ø	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$52
{1, 3}	11 > 10	NA
{1, 4}	12 >10	NA
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	9	$65 ÓPTIMA
{1, 2, 3}	14 > 10	NA
{1, 2, 4}	15 > 10	NA
{1, 3, 4}	16 > 10	NA
{2, 3, 4}	12 > 10	NA
{1, 2, 3, 4}	19 > 10	NA

Figura 24. Ejemplo de búsqueda de una solución óptima para el problema del saco.

Exploración inteligente de gráficas

Marcha atrás (backtrack)

Existen problemas que requieren encontrar un elemento con una propiedad especial dentro de un dominio que crece exponencialmente con el tamaño de la entrada del problema.

Por ejemplo:

- Encontrar un circuito Hamiltoniano entre todas las permutaciones de los vértices de una gráfica.
- Encontrar el subconjunto más valioso de artículos en el problema del saco.

Un enfoque de solución es la *búsqueda exhaustiva* que consiste en generar todas las posibles soluciones, y luego identificar las que satisfacen la propiedad deseada.

La marcha atrás o *backtrack* es una variación más inteligente de la búsqueda exhaustiva.

La idea principal es construir las soluciones, una componente a la vez, y evaluar esos candidatos parcialmente construidos como sigue.

Si una solución parcialmente construida puede ser más desarrollada, sin violar las restricciones del problema, se desarrolla tomando la primera opción legítima para la siguiente componente.

Sino hay opción legítima para la siguiente componente, ninguna alternativa para ninguna componente restante necesita ser considerada.

En este caso, el algoritmo da marcha atrás para reemplazar la última componente de la solución parcialmente construida con su siguiente opción.

Para implementar el tipo de procesamiento descrito se construye un árbol de las elecciones hechas, que se llama el *árbol del espacio de estados*. Su raíz representa el estado inicial antes de que inicie la búsqueda de una solución. Los nodos en el primer nivel representan las elecciones hechas para la primera componente de la solución. Los nodos en el segundo nivel representan las elecciones hechas para la segunda componente y así sucesivamente.

Un nodo en el espacio de estados se dice que es *promisorio* si corresponde a una solución parcialmente construida que aún puede conducir a una solución; de otra forma se llama *no promisorio*. Las hojas representan, o bien *nodos no promisorios finales* o *soluciones completas* encontradas por un algoritmo.

Ejemplo de un problema cuya solución puede abordarse con *backtrack* es el problema *de las n-reinas*.

Consiste en colocar *n* reinas en un tablero de *n* por *n*, de tal forma que ningún par de reinas se ataquen entre sí (no están en el mismo renglón, ni columna, ni diagonal).

Como cada reina se coloca en su propio renglón, todo lo que necesitamos hacer es asignarla a una columna apropiada.

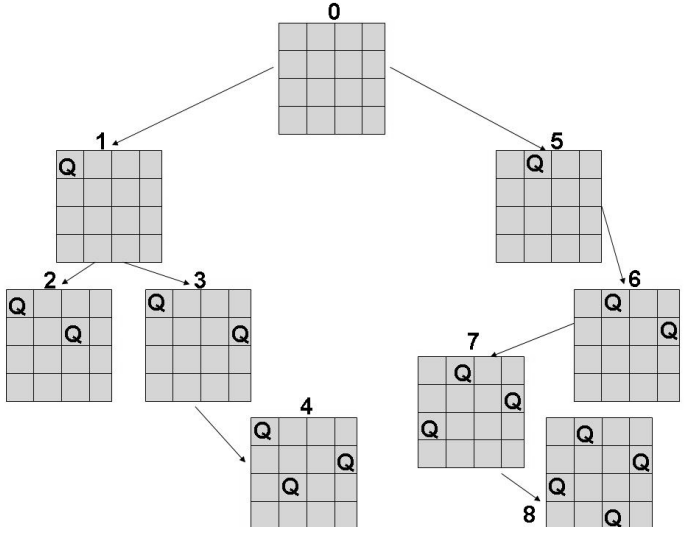


Figura 25. Ejemplo de una solución del problema de las 4-reinas usando *backtrack*.

La mayoría de los algoritmos de *backtrack* pueden pensarse que generan como salida *n*-tuplas $(x_1,\ldots x_n)$, donde cada coordenada x_i es un elemento de algún conjunto ordenado S_i.

Por ejemplo, para el problema de las reinas, cada S_i es un conjunto de enteros (números de columna) de 1 a *n*.

Las tuplas generalmente necesitan satisfacer restricciones adicionales, por ejemplo, el requerimiento de no atacarse para el problema de las *n*-reinas.

```
Algoritmo Backtrack(X[1..i])
// entrada: X[1..i] especifica las primeras i componentes promisorias de una solución
// salida: todas las tuplas que representan soluciones al problema
If X[1..i] es una solución write X[1..i]
else
 for cada elemento x de S_{i+1} consistente con X[1..i] y las restricciones do
 X[i+1] ← x
 Backtrack(X[1..i+1])
```

Tres puntos a favor de *backtrack*:

1. Se aplica típicamente a problemas combinatorios para los cuales no existen algoritmos eficientes para encontrar las soluciones exactas.
2. A diferencia de la búsqueda exhaustiva, en algunos casos, *backtrack* permite encontrar soluciones en una cantidad aceptable de tiempo.
3. Aún cuando *backtrack* no elimine ningún elemento y genere todos los elementos, proporciona una técnica específica.

Ramifica y acota (Branch and bound)

Recordemos que una de las ideas centrales del marcha atrás (*backtracking*) es la de cortar una rama del árbol de espacio de estados, tan pronto como deducimos que no puede conducir a una solución.

Esta idea puede ser fortalecida más si tratamos con un problema de optimización, que esté sujeto a algunas restricciones (longitud del camino, el valor de los objetos seleccionados, el costo de la asignación, etcétera).

Comparado con marcha atrás, ramifica y acota requiere dos puntos adicionales:

1. Una forma de proporcionar, para cada nodo de un árbol de espacio de estados, una cota en el mejor valor de la función objetivo (una cota inferior para un problema de minimización y una cota superior para un problema de maximización), en cualquier solución que puede ser obtenida agregando componentes adicionales a la solución parcial representada por el nodo.
2. El valor de la mejor solución vista hasta ahora.

Si esta información está disponible, podemos comparar el valor de la cota de un nodo con el valor de la mejor solución vista hasta ahora: si el valor de la cota no es mejor que la mejor solución vista hasta ahora (no más pequeña para un problema de minimización o más grande para un problema de maximización), el nodo es no promisorio y puede ser terminado (también se dice que la rama es cortada), porque no se puede obtener una mejor solución que la ya disponible. Esta es la idea principal de la técnica de ramifica y acota.

En general, terminamos un camino de búsqueda en el nodo actual de un espacio de búsqueda, en un algoritmo de ramifica y acota por alguna de las siguientes tres razones:

1. El valor de la cota del nodo no es mejor que el valor de la mejor solución vista hasta ahora.
2. El nodo representa soluciones no viables, porque las restricciones del problema no se satisfacen.
3. El subconjunto de soluciones factibles representadas por el nodo consiste en un solo punto (y por lo tanto, no se pueden hacer elecciones adicionales). En este caso se compara el valor de la función objetivo para esta solución viable con el de la mejor solución vista hasta ahora y se actualiza la mejor solución, si la nueva solución es mejor.

Una variante más elaborada de este algoritmo se emplea para mejorar el desempeño de la estrategia mínimax en juegos. En inteligencia artificial se le conoce como algoritmo de búsqueda con cortes alfa-beta, de forma que se maximiza la función objetivo del jugador

que usa el algoritmo y se minimiza la función objetiva del jugador oponente.

Algoritmos de aproximación para problemas NP

Si una instancia del problema es muy pequeña, podríamos resolverlo aplicando un algoritmo de búsqueda exhaustiva.

Un enfoque para resolver problemas de optimización difíciles, consiste en resolverlos aproximadamente con un algoritmo rápido. Este enfoque es atractivo para aplicaciones, en donde se busca una solución buena, pero no necesariamente la óptima.

La mayoría de los algoritmos de aproximación son algoritmos ambiciosos (*greedy*), basados en alguna heurística específica del problema. Por ejemplo, podemos ir a la siguiente ciudad más cercana no visitada en el problema del vendedor viajero.

Si usamos un algoritmo cuya salida es sólo una aproximación a la solución óptima, quisiéramos saber qué tan precisa es esta aproximación. Podemos cuantificar la precisión de una solución aproximada s_a, a un problema de minimizar alguna función f, por el tamaño del error relativo de esta aproximación.

$$\mathrm{re}(s_a) = [f(s_a) - f(s^*)] \ / \ f(s^*)$$

donde s^* es una solución exacta al problema. Alternativamente, como $\mathrm{re}(s_a) = f(s_a)/f(s^*) - 1$, podemos usar simplemente el cociente de precisión:

$$\mathrm{re}(s_a) = f(s_a) \ / \ f(s^*)$$

como una medida de la precisión de s_a.

Por uniformidad en la escala, el cociente de precisión de soluciones de aproximación a problemas de maximización a menudo se calcula como:

$$\mathrm{re}(s_a) = f(s^*) \ / \ f(s_a)$$

para hacer este cociente mayor o igual a 1. Mientras más cerca esté el cociente de 1, se tiene una mejor solución aproximada.

La mejor (menor) cota superior de posibles valores de re(s_a), tomados sobre todas las instancias del problema, es llamada la *razón de desempeño del algoritmo* y se denota por R_A.

Criptografía

La seguridad de un sistema criptográfico está directamente relacionada con la complejidad del algoritmo que puede encontrar las llaves. En esta sección presentamos el tamaño del espacio de llaves, para diferentes longitudes, con el objetivo de cuantificar el esfuerzo que requeriría un algoritmo por búsqueda exhaustiva en encontrar la llave.

Un esquema de encripción se dice que es *seguro computacionalmente* si se cumplen los siguientes dos criterios:

- El costo de romper el cifrado excede el valor de la información encriptada
- El tiempo requerido para romper el cifrado excede la vida útil de la información

Tamaño de llave (bits)	*Número alternativo de llaves*	*Tiempo requerido a 1 generación y verificación/μs*	*Tiempo requerido a 10^6 generación y verificación/μs*
32	$2^{32} = 4.3 \times 10^9$	$2^{31}\ \mu s = 35.8$ min	2.15 ms.
56 (DES)	$2^{56} = 7.2 \times 10^{16}$	$2^{55}\ \mu s = 1142$ años	10.01 hrs.
128	$2^{128} = 3.4 \times 10^{38}$	$2^{127}\ \mu s = 5.4 \times 10^{24}$ años	5.4×10^{18} años
168 (3 DES)	$2^{168} = 3.7 \times 10^{50}$	$2^{167}\ \mu s = 5.9 \times 10^{36}$ años	5.9×10^{30} años
26 caracteres (permutación)	$26! = 4 \times 10^{26}$	$2 \times 10^{26}\ \mu s = 6.4 \times 10^{12}$ años	6.4×10^6 años

Figura 26. Cálculo del espacio de búsqueda para diferentes longitudes de llaves y el tiempo que llevaría encontrar una llave por búsqueda exhaustiva, para diferentes capacidades de cómputo.

Ejercicios

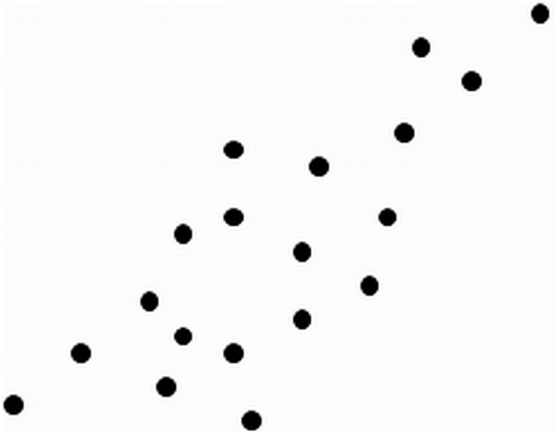
1. Investigue cómo la solución de problemas en inteligencia artificial, se formula como la búsqueda de la ruta en una gráfica que minimiza una función objetivo.
2. Investigue el algoritmo de cortes alfa–beta, para recorridos en gráficas y que se usa para encontrar rutas óptimas en problemas de juego, y en cuánto disminuye la complejidad este algoritmo, con respecto a una búsqueda ciega.
3. Encuentre la solución óptima para el problema del saco con un peso límite de 15 kg, para los siguientes objetos con sus respectivos pesos y valores: *a)* peso = 3, valor = 5; *b)* peso = 5, valor = 3; *c)* peso = 10, valor = 8; *d)* peso = 13, valor = 11;
4. Resuelva el siguiente sistema de ecuaciones por eliminación Gaussiana:

$$\begin{aligned} x_1 + x_2 + x_3 &= 2 \\ 2x_1 + x_2 + x_3 &= 3 \\ x_1 - x_2 + 3x_3 &= 8 \end{aligned}$$

5. Aplicando el algoritmo de Johnson-Trotter, genere todas las permutaciones para $n = 4$.
6. Resuelva el problema de la asignación, encontrando la asignación óptima de personas a trabajos, de acuerdo a la siguiente matriz de costos:

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

7. Encuentre la cubierta convexa para el siguiente conjunto de puntos:



8. Para el siguiente alfabeto de 6 caracteres, con las siguientes probabilidades de ocurrencia, obtenga sus códigos de Huffman y el número esperado de bits por caracter.

Caracter	**A**	**B**	**C**	**D**	*	-
Probabilidad	0.35	0.21	0.15	0.22	0.05	0.02

9. Encuentre una solución al problema de colocar 5 reinas en un tablero de 5 x 5, sin que se ataquen entre sí.
10. Programe el algoritmo de Horspool y pruebe su funcionamiento en diversos textos y con distintos patrones.
11. Dé una solución para el problema del cambio de un monto de $1 357 pesos, si se cuenta con billetes de denominaciones de 1 000, 500, 100, 50 y un peso.

7. Respuestas de los ejercicios

Capítulo 1

1. Un primer tipo de dato, es el *Booleano*, con valores, *v* (*verdadero*), y *f* (falso). Las operaciones que se pueden definir en este caso son la *y* y *o* exclusiva, con sus definiciones lógicas usuales.

   Un segundo tipo de datos es el *Conjunto*, que a su vez, sus valores se pueden tomar de cualquier otro tipo enumerado. Las operaciones usuales en este caso son la unión y la intersección.

   Un tercer tipo de datos es el *String*, sus valores son cadenas de caracteres. Una operación sobre el tipo de datos es la concatenación de cadenas.

2. Un problema es el ordenamiento de una lista de números o palabras. Existen muchos algoritmos de ordenamiento, como el *quicksort*, *bubble sort* o el *help sort*, con diferentes grados de desempeño, según las circunstancias. A su vez, estos algoritmos podemos programar en distintos lenguajes, como *C*, *C++* o Java.

3. Un primer tipo de datos abstractos es el tipo de los números enteros, con las operaciones de suma y multiplicación. Como estructura de datos para representar los enteros pueden usarse grupos de bytes, según el rango numérico que quiera manejarse para los enteros.

Un segundo TDA es una pila de caracteres, con las operaciones de insertar, sacar y consultar el elemento en el tope de la pila. Como estructura de datos para su implantación, podemos usar arreglos o lista ligadas.

Un tercer TDA es una cola de enteros, con las operaciones de insertar y sacar y consultar el primer elemento de la cola. Aquí también pueden usarse arreglos o listas ligadas para su implantación.

4. Una primera forma de cambiar una llanta es: *a)* sacar la herramienta y llanta de refacción de la cajuela. *b)* poner el gato en la posición correcta para levantar el auto, *c)* levantar el auto, *d)* quitar las tuercas de la llanta, *e)* quitar la llanta ponchada, *f)* poner la llanta de refacción, *g)* poner las tuercas y apretarlas, *h)* bajar el auto, *i)* guardar la herramienta y la llanta ponchada para repararla posteriormente.

Una segunda forma es: *a)* orillarse a un lugar seguro, *b)* sacar la herramienta, indicadores y llanta de refacción de la cajuela, *c)* colocar los indicadores de emergencia a unos 50 m antes del auto y poner las intermitentes, *d)* colocar el gato en la posición correcta para levantar el auto, *e)* aflojar algunas tuercas, *f)* levantar el auto, *g)* terminar de quitar las tuercas, *h)* quitar la llanta ponchada, *i)* colocar la llanta de refacción, *j)* colocar y apretar las tuercas, sin forzar, *k)* bajar el auto, *l)* volver a apretar las tuercas para asegurar la llanta, *m)* guardar la llanta de refacción y herramienta en la cajuela, *n)* quitar el indicador de emergencia y guardarlo en la cajuela, *ñ)* quitar intermitentes y partir.

Claramente el primer algoritmo, lleva menos pasos y es más eficiente. Por otra lado, el segundo, aunque lleva más pasos, toma más en cuenta, cuestiones de seguridad y calidad en el cambio de la llanta.

Capítulo 2

1. Movimientos para pasar 5 discos de una torre de Hanoi, del poste A al C. Los discos se numeran del 1 al 5, en orden de menor a mayor tamaño. Se usa el poste B como intermedio:

Mover el disco 1 del poste A al C.
Mover el disco 2 del poste A al B.
Mover el disco 1 del poste C al B.
Mover el disco 3 del poste A al C.
Mover el disco 1 del poste B al A.
Mover el disco 2 del poste B al C.
Mover el disco 1 del poste A al C.
Mover el disco 4 del poste A al B.
Mover el disco 1 del poste C al B.
Mover el disco 2 del poste C al A.
Mover el disco 1 del poste B al A.
Mover el disco 3 del poste C al B.
Mover el disco 1 del poste A al C.
Mover el disco 2 del poste A al B.
Mover el disco 1 del poste C al B.
Mover el disco 5 del poste A al C.
Mover el disco 1 del poste C al A.
Mover el disco 2 del poste B al C.
Mover el disco 1 del poste A al C.
Mover el disco 3 del poste C al A.
Mover el disco 1 del poste C al B.
Mover el disco 2 del poste C al A.
Mover el disco 1 del poste B al A.
Mover el disco 4 del poste B al C.
Mover el disco 1 del poste A al C.
Mover el disco 2 del poste A al B.
Mover el disco 1 del poste C al B.
Mover el disco 3 del poste A al C.
Mover el disco 1 del poste B al A.
Mover el disco 2 del poste B al C.
Mover el disco 1 del poste A al C.
$31 = 2^5 - 1$ movimientos

2. Para demostrar que $\sqrt{2}$ no es racional, primero supongamos que sí es racional. Si fuera racional, podría escribirse como cociente de 2 enteros, expresados como productos de primos, sin factores

comunes, como sigue: $\sqrt{2} = p/q$. Esto nos lleva a la conclusión que $p^2 = 2\,q^2$. Que a su vez implica, $p^2/q^2 = 2$. Lo cual conduce a que p^2 es de la forma $2\,q^2$. Esto significa que en su expansión original en números primos, p tendría que contener un entero primo que elevado al cuadrado fuera 2, lo cual es imposible para un entero. Finalmente esto es una contradicción, originada por la suposición de racionalidad de $\sqrt{2}$. Así que lo opuesto es que debe ser verdad, que $\sqrt{2}$ no es racional.

3. Verifiquemos que se cumple para $n = 1$. $1 = 1(1 + 1)/2$. Ahora supongamos que es válido para n, y a partir de ello, probemos que se cumple para $n + 1$. $1 + 2 + \ldots + n + (n + 1) = n(n + 1)/2 + (n + 1) = (n + 1)(n/2 + 1) = (n + 1)(n + 2)/2$. Por lo tanto, se ha probado que si es válido para n, también lo será para $n + 1$. Como también se probó para el caso base, $n = 1$, esto significa que la fórmula es válida para todos los naturales.

4. Supongamos que existirá un entero N más grande que todos. Esto nos lleva a que $N + 1$ también es un entero, pero $N + 1 > N$, lo cual es una contradicción con la suposición original. Por lo tanto, es imposible que exista un entero máximo.

5. Para probar que Q es numerable, se genera una tabla infinita en donde estarán todos los racionales positivos. En el primer renglón se ponen los números naturales y con denominador de 1. Luego en el segundo renglón, en el numerador van de nuevo los números naturales, pero se les pone como denominador el número 2. Se continúa este proceso, de tal forma, que el racional p/q, se encuentra en la columna p, renglón q. A continuación se inicia un proceso de asociación con los números naturales iniciando en $1/1$ y luego continuando por las diagonales. De esta forma, todo racional positivo se asigna con un natural. Los racionales negativos se pueden asociar de igual manera con los enteros negativos. Como los enteros Z, tienen la misma cardinalidad que los naturales, N, se concluye, que los racionales tienen la misma cardinalidad que N y son por lo tanto numerables.

6. Como $R = Q \cup I$, y Q es numerable, si I fuera numerable, R sería numerable. Sin embargo, R no es numerable, por lo tanto, I debe ser no numerable.

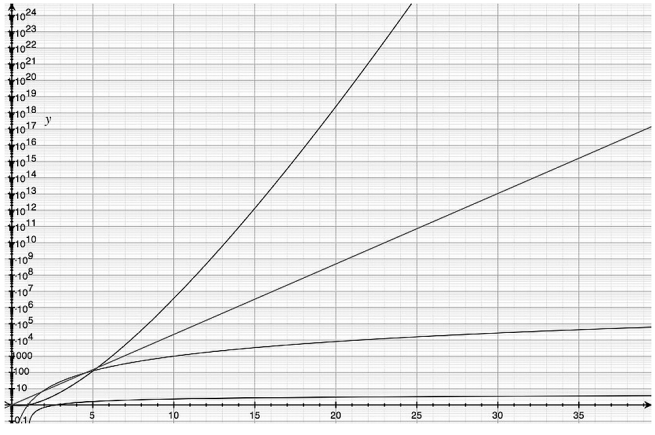
Capítulo 3

1. Un ejemplo es programar el algoritmo de la burbuja para ordenar números. Es de orden cuadrático. Por lo cual, al graficar, conforme n crece, debe tener la forma de una parábola.
   Una cota inferior sería $n \log n$, y una cota superior n^3.

2. El algoritmo tradicional de multiplicación de matrices es de orden n^3, por lo cual, una cota superior, es n^4. Una cota inferior, sería n^2, o bien $n^{2.807}$ como el algoritmo de Strassen, o $n^{2.376}$ como el algoritmo de Coppersmith-Winograd.

3. El número de movimientos óptimo para resolver una torre de Hanoi de n discos es $2^n - 1$.

4. De menor a mayor orden de crecimiento, las funciones propuestas, quedan como sigue: ln n, n^3, e^n, $n!$ Se anexa una gráfica de las funciones. Para poder desplegar las funciones y notar su tendencia, se usa una escala logarítmica en el eje vertical.



5. Una cota superior en la mayoría de los algoritmos de ordenamiento es n^2.

6. El orden del mejor algoritmo de ordenamiento conocido hasta el momento es $n \log n$.

7. El orden de la búsqueda secuencia es N.

8. El orden de la búsqueda binaria es $\log_2 N$.

9. Si usamos un algoritmo a partir de la definición de número primo, para encontrar todos los números primeros menores o iguales a N, el algoritmo sería de orden N^2. Una siguiente mejora, sería sólo probar la divisibilidad con los números primos menores a $N^{1/2}$.

10. Se representa con $\pi(n)$ la función que el número de primos menores o iguales a n. Se ha encontrado que una aproximación y cota inferior es la expresión: $n/\ln(n)$. Una mejor aproximación se obtiene con la función logaritmo integral: $Li(n) = \int_2^n \frac{1}{\ln x} dx$

De la Vallée-Poussin demostró que el error es: $\pi(n) - Li(n) = O\left(ne^{-a\sqrt{\ln n}}\right)$ para una constante a, $a > 0$.

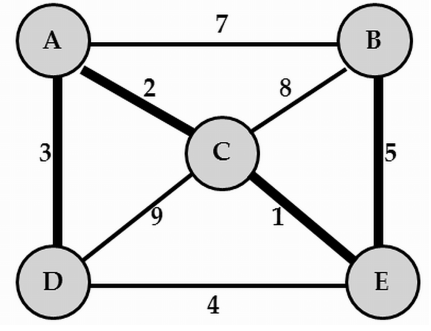
11. A continuación presentamos una tabla que calcula el número de elementos n, que podemos procesar con una computadora de 10 000 operaciones por segundo, en función de la tasa de crecimiento $f(n)$ del algoritmo. También se muestra el número de elementos n' que se puede procesar con una computadora 3 veces más rápida que procese 30 000 elementos por segundo.

f(n)	*n*	*n´*	*cambio*	*n´/n*
$10n$	1 000	3 000	$n' = 3n$	3.00
$20n$	500	1 500	$n' = 3n$	3.00
$5n \log_2 n$	251	645	$n' = 2.57n$	2.57
$2n^2$	70	122.47	$n' = 1.75n$	1.75
2^n	13	14.94	$n' = 1.15n$	1.15

12. En este ejercicio compare la mejora teórica obtenida al pasar de un cpu1 a un cpu2, Ghz_cpu2/Ghz_cpu1, y aislando factores como diferentes sistemas operativos, los tiempos experimentales, se espera que sigan una proporción similar. Es decir, por ejemplo, si el algoritmo es lineal, y al pasar de un cpu1 a un cpu2 se obtiene una mejora en performance X, se esperaría una reducción en ese factor en el tiempo de procesamiento. Si el orden del algoritmo no es lineal, debe ponderarse este factor para interpretación de los resultados de los tiempos de procesamientos experimentales.

Capítulo 4

1. Se muestra con líneas más anchas las conexiones que forman el árbol expansor de mínimo costo. En este caso, el costo total es de 11.



2. Como es una gráfica de 6 nodos, para resolver el problema del viajero a partir de uno de los nodos, necesitamos analizar 5! = 120 permutaciones para encontrar la ruta óptima. Como hay rutas que son equivalentes, porque sólo cambie el sentido de recorrido, únicamente tendríamos que analizar 120/2 = 60 permutaciones.

Fuente	*Destino*	*Ruta*	*Costo*
A	*B*	*A-B*	7
A	*C*	*A-C*	2
A	*D*	*A-D*	3
A	*E*	*A-C-E*	3
B	*A*	*B-A*	7
B	*C*	*B-E-C*	6
B	*D*	*B-E-D*	9
B	*E*	*B-E*	5
C	*A*	*C-A*	2
C	*B*	*C-E-B*	6
C	*D*	*C-A-D*	5
C	*E*	*C-E*	1
D	*A*	*D-A*	3
D	*B*	*D-E-B*	9
D	*C*	*D-E-C*	5
D	*E*	*D-E*	4
E	*A*	*E-C-A*	3
E	*B*	*E-B*	5
E	*C*	*E-C*	1
E	*D*	*E-D*	4

Capítulo 5

1. Las torres de Hanoi porque su tiempo de ejecución es exponencial, al menos es de $2^n - 1$.

El problema del saco, ya que hay examinar todos los subconjuntos de un conjunto de n elementos, los cuales son 2^n.

El vendedor viajero, ya que implica examinar $(n - 1)!/2$ rutas.

La generación de todas las permutaciones de un conjunto de n elementos lleva al menos $n!$

2. La secuencia de Collatz es muy sencilla de programar. Aún no se ha probado si existe un valor para el cual no se detenga. Se ha probado

con los números de 1 a menores que 2^{58}, y siempre se ha detenido. Para los números que son suma de potencias de 2, con exponente par, se han encontrado que siempre se detiene. Por ejemplo, con $5 = 1 + 4 = 2^0 + 2^2$. $21 = 1 + 4 + 16 = 2^0 + 2^2 + 2^4$. $53 = 1 + 4 + 16 + 32 = 2^0 + 2^2 + 2^4 + 2^5$.

3. Un ejemplo es la aplicación de filtros dígitales mediante la operación de convolución, y usando la equivalencia, que la transformada de Fourier de una convolución, es igual a la multiplicación de las transformadas de cada una de las funciones que intervienen en la convolución. $F\{f \text{ o } g\} = F\{f\} * F\{g\}$.

   Otro ejemplo, es el uso de la transformada de Laplace para resolver ecuaciones diferenciales lineales, para transformar en otro espacio, donde la solución es más sencilla y pueden aplicarse métodos algebraicos.

4. El método de Strassen para la multiplicación de matrices tiene orden: $N^{2.807}$.

5. La computación cuántica es un enfoque novedoso basado en propiedades cuánticas. La unidad básica de una computadora cuántica es el qubit (*quantum bit*) y puede representar el cero, uno o cualquier superposición cuántica de estos dos estados. Más aún, si tenemos n qubits, podemos tener cualquier superposición cuántica de 2^n estados simultáneamente. La mayoría de los algoritmos que se han propuesto para las computadoras cuánticas funcionan de manera probabilística. Por lo cual, se aplican más a problemas, que pueden formularse como una simulación, en donde mediante ensayos repetidos, la probabilidad de obtener una respuesta correcta se incrementa. Un tipo de problemas donde la computación cuántica puede tener ventaja respecto a las computadoras tradicionales es la factorización de enteros en sus primos. Se ha propuesto el algoritmo de Shor´s para resolver este problema en tiempo polinomial en una computadora cuántica. La computación cuántica continúa siendo un área de investigación, ya que sólo se han fabricado computadoras cuánticas de unos cuantos

qubits y es muy complejo incorporar datos, aplicar algoritmos y obtener resultados.

Capítulo 6

1. Muchos de los problemas en inteligencia artificial se plantean como la búsqueda en un espacio de estados. El nodo de donde parte la búsqueda es la configuración inicial del problema, y nodos adyacentes corresponden con configuraciones relacionadas mediante la aplicación de algún operador. Cada vez que se llega a un nodo, se evalúa por si es la solución o su valor relativo, como una mejor configuración. Al presentarse múltiples opciones de exploración se busca por la ruta más factible.

2. En el peor de los casos, el corte alfa-beta es de la misma complejidad que el algoritmo minimax. Por otro lado, en el mejor de los casos, si b es el factor de ramificación y d es el número de niveles, la complejidad del corte alfa-beta es $(b^d)^{1/2}$, que es menor que b^d, que le lleva al minimax.

3. Como se trata de 4 objetos, para resolver el problema del saco, necesitamos considerar todos los subconjuntos de un conjunto de 4 elementos, estos son: $2^4 = 16$.

Subconjunto	*Peso total*	*Valor total*
Ø	0	$0
{1}	3	$5
{2}	5	$3
{3}	10	$8
{4}	13	$11
{1, 2}	8	$8
{1, 3}	13	$13 ÓPTIMA
{1, 4}	16 > 15	NA
{2, 3}	15	$11

{2, 4}	18 > 15	NA
{3, 4}	23 > 15	NA
{1, 2, 3}	18 > 15	NA
{1, 2, 4}	21 > 15	NA
{1, 3, 4}	16 > 15	NA
{2, 3, 4}	28 > 15	NA
{1, 2, 3, 4}	31 > 15	NA

4.

$$\begin{pmatrix} 1 & 1 & 1 & 2 \\ 2 & 1 & 1 & 3 \\ 1 & -1 & 3 & 8 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & -2 & 2 & 6 \end{pmatrix}$$

Renglón 2 -2 renglón 1
Renglón 3 -1 renglón 1

$$\begin{pmatrix} 1 & 1 & 1 & 2 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 4 & 8 \end{pmatrix}$$

Renglón 3 -2 renglón 2

Ahora obtenemos la solución por sustitución hacia atrás:

$$x_3 = (8)/4 = 2\,,\ x_2 = 1 - x_3 = -1 \text{ y } x_1 = 2 - x_2 - x_3 = 1$$

5. Para representar la dirección de movilidad de cada elemento, usaremos < para flecha izquierda y > para flecha derecha. Las permutaciones para $n = 4$ son:

< 1	< 2	< 3	< 4
< 1	< 2	< 4	< 3
< 1	< 4	< 2	< 3
< 4	< 1	< 2	< 3
4 >	< 1	< 3	< 2
< 1	4 >	< 3	< 2
< 1	< 3	4 >	< 2
< 1	< 3	< 2	4 >

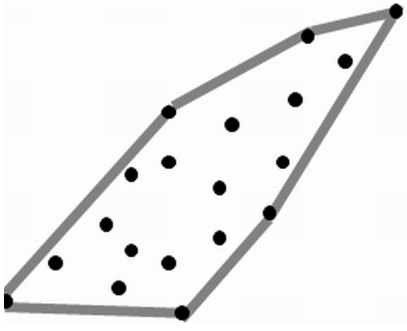
< 3	< 1	< 2	< 4
< 3	< 1	< 4	< 2
< 3	< 4	< 1	< 2
< 4	< 3	< 1	< 2
4 >	3 >	< 2	< 1
3 >	4 >	< 2	< 1
3 >	< 2	4 >	< 1
3 >	< 2	< 1	4 >
< 2	3 >	< 1	< 4
< 2	3 >	< 4	< 1
< 2	< 4	3 >	< 1
< 4	< 2	3 >	< 1
4 >	< 2	< 1	3 >
< 2	4 >	< 1	3 >
< 2	< 1	4 >	3 >
< 2	< 1	3 >	4 >

6. Para resolver el problema de la asignación generamos todas las permutaciones de 4 elementos, 4! = 24, calculamos los costos y seleccionamos la permutación con costo mínimo:

Permutación	*Costo*
(1, 2, 3, 4)	3 + 8 + 13 + 3 = 27
(1, 2, 4, 3)	3 + 8 + 12 + 5 = 28
(1, 3, 2, 4)	3 + 1 + 11 + 3 = 18
(1, 3, 4, 2)	3 + 1 + 12 + 2 = 18
(1, 4, 2, 3)	3 + 9 + 11 + 5 = 28
(1, 4, 3, 2)	3 + 9 + 13 + 2 = 27
(2, 1, 3, 4)	5 + 2 + 13 + 3 = 23
(2, 1, 4, 3)	5 + 2 + 12 + 5 = 24
(2, 3, 1, 4)	5 + 1 + 6 + 3 = 15 ÓPTIMO
(2, 3, 4, 1)	5 + 1 + 12 + 10 = 28
(2, 4, 1, 3)	5 + 9 + 6 + 5 = 25
(2, 4, 3, 1)	5 + 9 + 13 + 10 = 37
(3, 1, 2, 4)	4 + 2 + 11 + 3 = 20

Permutación	*Costo*
(3, 1, 4, 2)	4 + 2 + 12 + 2 = 20
(3, 2, 1, 4)	4 + 8 + 6 + 3 = 21
(3, 2, 4, 1)	4 + 8 + 12 + 10 = 34
(3, 4, 1, 2)	4 + 9 + 6 + 2 = 21
(3, 4, 2, 1)	4 + 9 + 11 + 10 = 34
(4, 1, 2, 3)	7 + 2 + 11 + 5 = 25
(4, 1, 3, 2)	7 + 2 + 13 + 2 = 24
(4, 2, 1, 3)	7 + 8 + 6 + 5 = 26
(4, 2, 3, 1)	7 + 8 + 13 + 10 = 38
(4, 3, 1, 2)	7 + 1 + 6 + 2 = 16
(4, 3, 2, 1)	7 + 1 + 11 + 10 = 29

7. Se muestra la cubierta convexa del conjunto de puntos:

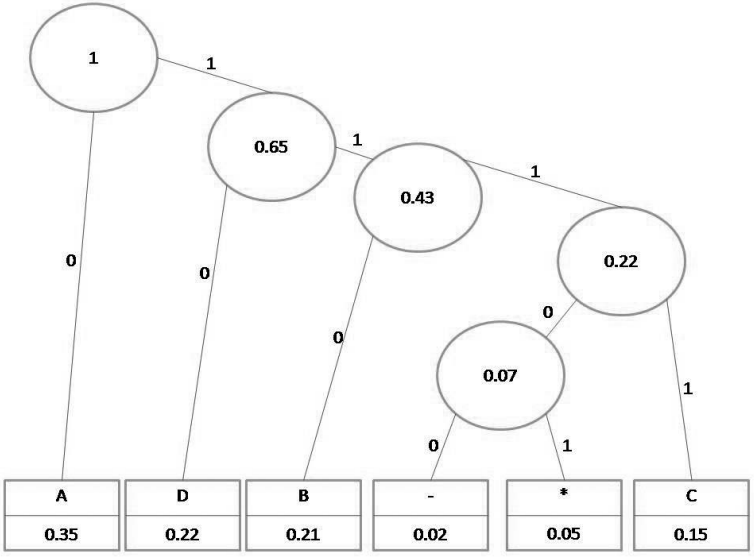


8. Se anexa la tabla de los códigos de Huffman para los símbolos considerados:

Símbolo	*Código*
-	11100
*	11101
C	1111
B	110
D	10
A	0

La longitud promedio de los códigos es: 5*0.02 + 5*0.05 + 4*0.15 + 3*0.21 + 2*0.22 + 1*0.35 = 2.37 bits.

Se muestra el árbol de Huffman:



9. Se anexa una solución por el problema de las 5 reinas:

	Q			
			Q	
Q				
		Q		
				Q

10. Se incluye el código fuente y ejemplos de su funcionamiento en la sección de programas.

11. Aplicando el algoritmo para dar cambio, el monto $1 357 se puede descomponer de la siguiente forma, utilizando el menor número de unidades monetarias. $1 357 = 1 * $1 000 + 0 * $500 + 3 * $100 + 1* $50 + 7* $1.

Bibliografía

Binstock, A. y J. Rex. *Practical Algorithms for Programmers*. 1a. ed., Addison-Wesley Proffesional, 1995, 592 p.

Chabert, Jean Luc *et al. A History of Algorithms: From the Pebble to the Microchip*. Springer, 1999, 533 p.

Cormen, T. H. *et al. Introduction To Algorithms.* 3a. ed., MIT Press, 2009, 1312 p.

Lee R., C. T. *et al. Introduction to the Design and Analysis of Algorithms, a Strategic Approach*. McGraw Hill, 2005, 752 p.

Levitin, A. *Introduction to the Design and Analysis of Algorithms*. 3a. ed., Addison-Wesley, 2011, 592 p.

Neapolitan, R. y K. Naimipour. *Foundations of algorithms*. 4a. ed., Jones and Barlett Publishers, 2009, 627 p.

Nilsson, N. J. *Artificial Intelligence: A New Synthesis*. Morgan Kauffman Publishers, EUA, 1998, 513 p.

PAPADIMITRIOU, C. H. y K. Steiglitz. *Combinatorial Optimization Algorithms and Complexity.* Dover Publications, 1998, 528 p.

RUSSELL, S. y P. Norvig. *Artificial Intelligence: A Modern Approach*. 3a. ed., Prentice-Hall, 2011, 1152 p.

SCHNEIER, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 2a. ed., John Wiley, 1996, 758 p.

Índice

Siendo rector de la Universidad Veracruzana
el doctor Raúl Arias Lovillo,
Análisis de algoritmos, de Homero Vladimir Ríos Figueroa,
Fernando Martín Montes González y Víctor Ricardo Cruz Álvarez
se terminó de imprimir en mayo de 2013
en en los talleres de Proagraf, S. A. de C. V.,
Av. 20 de Noviembre núm. 649, col. Badillo, CP 91190,
Xalapa, Veracruz, México. Tel. 2288906204.
La edición, impresa en papel cultural de 90 g,
consta de 500 ejemplares más sobrantes para reposición.
En su composición se usaron tipos Palatino de 10/11, 10/12, 12/14 y 16 puntos.
Formación: Aída Pozos Villanueva.
Edición: Martha Judith Vásquez Fernández.
La revisión técnica estuvo a cargo de Homero Vladimir Ríos Figueroa.

textos universitarios

En este texto los autores ofrecen al lector un enfoque innovador al realizar un balance entre la teoría y la práctica, al sintetizar la parte esencial del análisis de algoritmos y unirla a ejercicios y proyectos de diversos grados de dificultad; en un anexo además proporcionan las respuestas a las preguntas planteadas en los ejercicios así como el código en Java de los principales algoritmos estudiados.

Un aspecto que queda claro en ANÁLISIS DE ALGORITMOS es el carácter práctico de éstos, ya que se utilizan en la mayoría de las actividades cotidianas, por ejemplo para determinar caminos óptimos para repartir mercancías, con lo que los autores quieren demostrar a los lectores la ubiquidad de los algoritmos en el quehacer de la humanidad, además de mostrar cómo han evolucionado y cómo han incidido en otras ciencias fundamentales como las matemáticas.

Cabe destacar que el contenido de este texto se ha impartido en varios cursos de posgrado en distintas ocasiones, y que gracias a las aportaciones de los estudiantes se ha mejorado significativamente.

ISBN: 978-607-502-239-0