INTRODUCCIÓN AL ANÁLISIS DE ALGORITMOS

Cenid Editorial

INTRODUCCIÓN AL ANÁLISIS DE ALGORITMOS

Joel Ayala de la Vega
Irene Aguilar Juárez
Farid García Lamont
Hipólito Gómez Ayala

Si desea publicar un libro o un artículo de investigación contáctenos.

Av. México #2798. Piso 5-B, Torre Diamante
Circunvalación Vallarta
C.P. 44680 Guadalajara, Jalisco, México
Teléfono: 01 (33) 1061 8187
ww.cenid.org.mx
redesdeproduccioncenid@cenid.org

Edición y Diagramación:
Orlanda Patricia Santillán Castillo

INTRODUCCIÓN AL ANÁLISIS DE ALGORITMOS

ISBN: 978- 607- 8435-78-4

Primera edición 2019

Miembro de la Cámara Nacional de la Industría Editorial Mexicana Socio #3758

Impreso en México / Printed in Mexico

ÍNDICE

PROCESO DE REVISIÓN POR PARES

Los trabajos recibidos pasan en todos los casos por un proceso de arbitraje *(peer review)* por parte de los evaluadores designados por el consejo editorail. Los evaluadores emiten un juicio sobre las propuestas de publicación, con las observaciones que consideran pertinentes. Cuando la evaluación es positiva, las observaciones de los evaluadores se envian a los autores mediantes los editores.

En la presente publicación del consejo Editoral designó al siguiente grupo de evaluadores:

Georgina Guadalupe Oyervides Zapata
Benito León Corona
Gabriel de Sousa Torcato David
Christine Riegel
Kathryn Haselden
Joaquín García Carrasco
Paula Bock
Yolanda Arias Menéndez

Publicación financiada con recursos PFCE 2019

COMPLEXITY

A physician, a civil engineer, and a computer scientist were arguing about what was the oldest profesión in the world. The physician remarked, "Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profesión in the world." The civil engineer interrupted, and said, "But even earlier in the book of Genesis, it states that God created the order of the heavens and the earth from out the chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong: mine is the oldest profesión in the world." The computer scientist leaned back in her chair, smiled, and then said confidently, "Ah, ¿but who do you think created the chaos?" (Booch *et al.,* 2007)*.*

EL NACIMIENTO DE LA COMPUTACIÓN

…Su nombre era Kurt Gödel (1906-1978). El primer teorema lo expuso abiertamente el 7 de septiembre. El artículo con el desarrollo de la demostración fue enviado a la revista Monatshefte für Mathematik and Physik *en noviembre y apareció en el volumen 38 (1931),* ***una publicación cuya relevancia para la lógica es solo comparable con la Metafísica de Aristóteles****. La exposición de la demostración fue tan clara que no generó ni la más mínima controversia. Este fue el embrión de las ciencias de la computación* (Piñeiro, 2017).

AGRADECIMIENTOS

Este libro es el resultado del análisis de textos académicos relacionados con la programación de computadoras, tarea desarrollada durante varios años de trabajo en la enseñanza en el nivel superior. Agradecemos el apoyo de la Universidad Autónoma del Estado de México que permite, mediante la asignación de recursos, generar materiales académicos para fortalecer el perfil profesional de su comunidad. En particular, este libro fue financiado con recursos PFCE 2019 con el objetivo de proveer a los alumnos de ingeniería en Computación y a los profesionales de carreras afines una fuente de información especializada para sus planes de estudio.

PRESENTACIÓN

Los avances en el estudio de la computación no solo determinan las posibles tecnologías que se podrán emplear en el futuro, sino que también definen el ámbito de aplicación de los programas y los límites de la solución de problemas mediante el uso de computadoras. Esto se ha conseguido gracias al aprovechamiento de los conocimientos matemáticos logrados durante el siglo XX, los cuales sentaron las bases para manipular los equipos digitales de nuestro tiempo y para impulsar el desarrollo exponencial de la ciencia.

Debido a lo anterior, en la actualidad es cada vez más importante que los alumnos de nivel superior (especialmente los vinculados con las matemáticas y carreras afines al cómputo, como la psicología o la filosofía) comprendan los temas de dicha área. Por este motivo, en la presente obra se explican algunos de los principales conceptos de la teoría de la computación con el fin de intentar familiarizar a los referidos estudiantes con el campo de los algoritmos y de su eficiencia.

En concreto, el libro se compone de nueve capítulos organizados es cuatro secciones. En la sección I se abordan los conceptos sobre la complejidad algorítmica y la metodología para calcularla; asimismo, se describen algunos programas implementados en lenguaje C con los que se ejemplifica la solución de problemas mediante algoritmos de distinta complejidad, los cuales sirven para aumentar la lógica computacional de los estudiantes.

En la sección II (la más extensa) se explican los diferentes paradigmas de solución de problemas (como divide y conquistarás), la estrategia del avaro, la programación dinámica, la programación lineal, el retorno atrás, y la ramificación y acotamiento, todas estas usadas para resolver problemas de ordenamientos, búsquedas, optimización o decisión.

En la sección III se comentan las tesis matemáticas más relevantes del siglo XX, las cuales determinaron el avance de las matemáticas, la ciencia y la computación en las últimas décadas. Igualmente, se narra la evolución de las reflexiones matemáticas y sus implicaciones en la ciencia, y se plantean las preguntas matemáticas que aún se hallan a la espera de una respuesta, lo cual seguramente contribuirá al desarrollo de la computación.

Por último, en la sección IV se explican los fundamentos matemáticos requeridos para iniciarse en la complejidad algorítmica, para lo cual se usan ejemplos y ejercicios que permitirán poner en práctica los conceptos tratados. Esta sección es importante como capítulo introductorio y se recomienda para estudiantes que no han tratado el tema previamente.

En definitiva, se sugiere usar este material como libro de texto para cursos de programación avanzada o de complejidad algorítmica, con el estudio de las secciones I, II y III, mientras que se recomienda iniciar en la sección IV solo si los estudiantes necesitan fortalecer su conocimiento sobre los prerrequisitos.

SECCIÓN I

Complejidad algorítmica

En la medida en que se refiere a la realidad, las proposiciones de las matemáticas no son seguras, y viceversa, en la medida en que son seguras, no se refieren a la realidad.

Albert Einstein

INTRODUCCIÓN

Dos ideas cambiaron el mundo. En 1448, en la ciudad de Mainz, un orfebre llamado Johann Gutenberg descubrió el camino para imprimir libros colocando juntas dos piezas metálicas móviles. En ese momento se inició la disipación de la edad obscura, lo cual sirvió para liberar al intelecto humano. En otras palabras, la ciencia y la tecnología triunfaron y estimularon la fermentación de una semilla que culminó con la revolución industrial. Por eso, entre muchos historiadores persiste la idea de que todos esos cambios se lo debemos a la tipografía, pues antes de ella solo una élite podía tener acceso al conocimiento bibliográfico. Otros historiadores, sin embargo, consideran que la llave de ese desarrollo no lo produjo la tipografía, sino la algorítmica.

En nuestros días estamos acostumbrados a escribir números en notación decimal, de ahí que sea sencillo olvidar que Gutenberg utilizaba la notación romana para escribir, por ejemplo, el número 1448 (es decir, MCDXLVIII). Pero, en esa época, ¿cómo se podían sumar dos números romanos? Para cantidades pequeñas se solían emplear los dedos de las manos, pero para cifras más complejas se debía utilizar el ábaco.

El sistema decimal —inventado en la India alrededor del año 600 a. C.— produjo una revolución en el razonamiento cualitativo, pues únicamente usando diez símbolos se podían representar números de gran tamaño sin ninguna complicación. De este modo, cualquier operación aritmética se podía concretar de forma sencilla. No obstante, estas ideas tardaron bastante en expandirse debido a las tradiciones, la distancia y las barreras del lenguaje. De hecho, solo fue hasta el siglo noveno de nuestra era, aproximadamente, cuando Al Khwarizmi —un matemático, astrónomo y geógrafo persa— dio a conocer los métodos básicos para la suma, la resta, la multiplicación, la división e incluso para determinar la raíz cuadrada y el cálculo de . Sus procedimientos fueron precisos, eficientes y correctos, y se les llamó *algoritmos*, un término acuñado en honor al hombre sabio.

Si bien el trabajo de Al Khwarizmi solo se pudo establecer en Europa varios siglos después (especialmente por el esfuerzo de un matemático italiano del siglo XIII, conocido como Leonardo Fibonacci, quien descubrió el potencial del sistema posicional y trabajó bastante para su desarrollo y futura propagación), a medida que se fue implementado el sistema decimal se pudo generar un desarrollo más acelerado en distintos campos de la tecnología, la ciencia, el comercio, la industria y, posteriormente, el cómputo. De hecho, científicos de todo el mundo crearon algoritmos cada vez más complejos para todo tipo de problemas e investigaron acerca de novedosas aplicaciones que han cambiado el mundo de forma radical.

CONCEPTOS

El *Diccionario* de la Real Academia Española define el vocablo *algoritmo* en los siguientes términos: "Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema", mientras que para la palabra *cómputo* se otorga la siguiente definición: "Método preciso usado por una computadora para la solución de problemas".

Ahora bien, un algoritmo está compuesto por un conjunto finito de pasos, cada uno de los cuales puede requerir de una o más operaciones que deben estar definidas. Cada paso debe ser hecho, al menos, por una persona usando lápiz y papel en un tiempo finito. Este procedimiento no se puede confundir con lo sugerido por la palabra *receta,* pues en esta última muchas de las acciones pueden estar orientadas por la subjetividad (en determinadas preparaciones se puede recomendar, por ejemplo, agregar sal al gusto); en cambio, en el algoritmo cada uno de los pasos se debe indicar y realizar con exactitud.

Otra frase importante en este campo es *proceso computacional* (un ejemplo es el sistema operativo). Diseñado para controlar la ejecución de trabajos, este proceso, en teoría, nunca termina, ya que se queda en estado de espera hasta la llegada de solicitud de otro trabajo.

El estudio de algoritmos incluye varias y activas áreas de la investigación, entre las que se destacan las siguientes:

a. Cómo elaborar algoritmos.
b. Cómo expresarlos.
c. Cómo validarlos.
d. Cómo analizarlos.
e. Cómo probarlos.

El *debugging* (depuración) solo indica la presencia de errores, no la ausencia de ellos. Al analizar un algoritmo se debe tomar en cuenta tanto el aspecto temporal como el espacial, pues debe procesar la información de forma rápida y correcta. Pero ¿cómo se puede determinar si un algoritmo es eficiente o no?

Para responder esta interrogante se deben tomar en cuenta los conceptos *tiempo de ejecución y complejidad del algoritmo*, los cuales se vinculan con la cantidad de pasos que realiza un algoritmo para generar un conjunto de datos como salida a partir de un conjunto de datos de entrada.

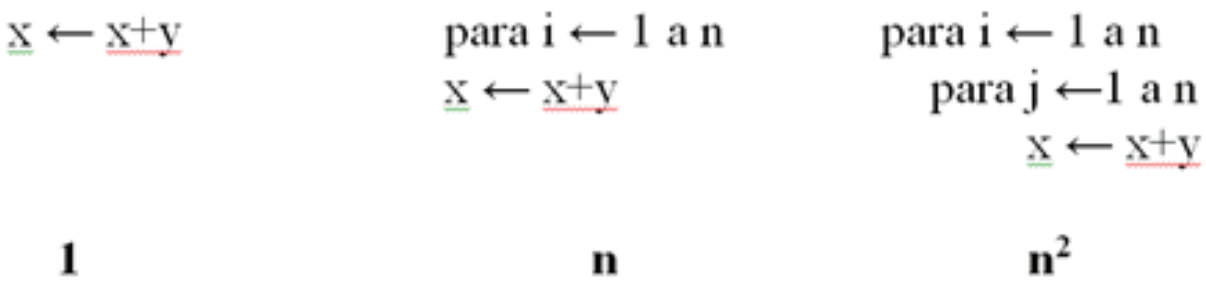
Así, para analizar un algoritmo, lo primero que se debe hacer es verificar cuáles operaciones son empleadas y su costo.

Estas operaciones se pueden acotar en un tiempo constante (p. ej., la comparación entre caracteres se puede hacer en un tiempo fijo), mientras que la comparación de las cadenas depende del tamaño de estas (no se puede acotar en tiempo).

Para cada problema se determina una medida *N* de su tamaño (por número de datos) y se intentan hallar respuestas en función de dicha *N*. El concepto exacto que mide N depende de la naturaleza del problema. Así, para un vector se suele utilizar como *N* su longitud: por ejemplo, para una matriz, el número de elementos que la componen; para un grafo puede ser el número de nodos (aunque a veces es más importante considerar el número de arcos, según el tipo de problema que se intenta resolver), mientras que en un archivo se suele usar el número de registros, etc. En estos casos, es imposible ofrecer una regla general, pues cada problema tiene su propia lógica de coste.

Una medida que suele ser útil es conocer el tiempo de ejecución de un programa en función de N, lo cual se denomina T(*N*). Esta función se puede medir físicamente (ejecutando el programa con reloj en mano) o se puede calcular sobre el código contando las instrucciones que se deben ejecutar y multiplicando el tiempo requerido por cada instrucción.

Para ver los tiempos de un algoritmo se requiere un análisis a *priori*, y no a *posteriori.* En un análisis a *priori* se obtiene una función que acota el tiempo del algoritmo. En un análisis a *posteriori* se colecta una estadística sobre el desarrollo del algoritmo en tiempo y espacio al momento de su ejecución. El siguiente es un ejemplo de análisis a *priori*:



El análisis a *priori* ignora qué tipo de máquina es, así como el lenguaje, pues solo se concentra en determinar el orden de magnitud de la frecuencia de ejecución.

Prácticamente todos los programas reales incluyen alguna sentencia condicional, lo que hace que las sentencias efectivamente ejecutadas dependan de los datos concretos presentados. Por ello, se debe hablar de un rango de valores, y no solo de un valor T(*N*):

$$T_{min(}N) \leq T(N) \leq T_{max}(N)$$

Los extremos son habitualmente conocidos como *peor caso* y *mejor caso*, entre los cuales se hallará algún *caso promedio* o más frecuente.

Cualquier fórmula T(*N)* incluye referencias al parámetro *N* y a una serie de constantes K que dependen de factores externos al algoritmo, como pueden ser la calidad del código generado por el compilador y la velocidad de ejecución de instrucciones del ordenador que lo ejecuta. Dado que es fácil cambiar de compilador y que la potencia de los ordenadores crece a un ritmo vertiginoso, se deben intentar analizar los algoritmos con algún nivel de independencia de estos factores; es decir, buscar estimaciones generales ampliamente válidas.

A la cantidad de operaciones básicas de un algoritmo que se representa como una función f(*n*) —cuyo argumento es el tamaño de la entrada— se le denomina *complejidad del algoritmo*.

Se pueden emplear supercomputadoras para algoritmos altamente concurrentes; esto, en principio, permite suponer que el problema se puede resolver *n* veces más rápido porque se tienen n procesadores. Desafortunadamente, este no es el caso, ya que existen diferentes conflictos, como el acceso a la memoria, el conflicto sobre la comunicación entre los trayectos de procesadores y procesadores a memoria, así como la ineficiencia por la implementación de la concurrencia de los algoritmos. Por eso, la estimación de la velocidad es mucho menor a *n*.

Una cota inferior estimada es la conjetura de Minsky, conocida como $\log_2$ (***n)***. Para la cota superior, depende de si el programa incluye la porción de entrada/salida (usualmente código secuencial). Si es así, se considera la cota superior como n/ln(***n***).

La tabla 1.1 muestra una estimación para la cota superior y la cota inferior para un sistema de n procesadores.

Tabla 1.1. Estimación de cota superior e inferior

Número de procesadores	*Cota inferior* $Log_2(n)$	*Cota superior* $n/ln(n)$
2	1	2.89
4	2	2.89
8	3	3.85
16	4	5.77
32	5	9.23

Como se puede observar, agregar más procesadores no necesariamente hace más rápida a la computadora. Para cada procesador añadido, la velocidad es cada vez más lenta, lo que hace más complejo el manejo de los procesadores. Por esta razón, los esfuerzos van encaminados no solo a explotar la concurrencia en algoritmos, sino también a aprovechar de la mejor manera un número pequeño de procesadores veloces, en lugar de depender de más procesadores, lo cual disminuye la velocidad de ejecución (Langholz, Francioni y Kandel, 1989). Para expresar los algoritmos se tienen dos formas:

- El seudocódigo que es similar al lenguaje C y Pascal, con palabras reservadas.
- Lenguaje de alto nivel, como cualquiera de los utilizados en programación.

En este trabajo la implementación de los códigos será en lenguaje C estándar. Para determinar la complejidad de un algoritmo es necesario analizar el comportamiento y ejecución de cada uno de sus pasos. Por ejemplo, ¿cuál es la complejidad del algoritmo 1.1?

Algoritmo 1.1. Ejemplo de un ciclo simple

Entrada: n	
Salida: c	

Repetición(n){
1: $c = 0$;
2: **for** $i = 1$ **to** n **do**
3: $c++$;
4: **end for**
5: **return** c;
6: }
	int Repeticion(int n){
	1. int c=0;
	2. for(i=1; i<=n; i++)
	3. c++;
	4. return c;
	}

En el caso anterior se debe contar la cantidad de veces que cada línea se ejecuta. Por ejemplo, las líneas 1 y 5 se ejecutan solo una vez; las líneas 2 y 3 se ejecutan n+1 y n veces, respectivamente. Al sumarse se tiene f(n)=1+n+1+n+1, y finalmente tenemos f(n)=2n+3; por lo tanto, se dice que la complejidad de este algoritmo es un polinomio de primer grado.

Algoritmo 1.2. Ejemplo de dos ciclos anidados con la misma cantidad de repeticiones cada uno

Entrada: n	
Salida: c	

Repetición(n){
1: $c = 0$;
2: **for** $i = 1$ **to** n **do**
3: **for** $j = 1$ **to** n **do**
4: $c++$;
5: **end for**
6: **end for**
7: **return** c;
8: }
	int Repeticion(int n){
	int i,j;
	int c=0;
	for(i=1; i<=n; i++)
	for(j=1; j<=n; j++)
	c++;
	
	return c;
	}

En el algoritmo 1.2, al contar la cantidad de veces que se ejecuta cada línea, se ve que las líneas 1 y 7 lo hacen una vez cada una. La línea 2 se ejecuta *n*+1 veces, en la línea 3 se encuentra un ciclo anidado dentro del ciclo controlado por la variable i; entonces, por cada iteración del primer ciclo, la línea 3 se repite *n*+1 veces. Pero por cada vez que se repita el ciclo de la línea 2, la línea 4 se ejecuta n veces. Por lo tanto, la complejidad es f(*n*)=1+*n*+1+*n*(*n*+1)+*n*(*n*)+1; luego, reduciendo términos, se obtiene f(*n*)=2*n*^2+2*n*+3. En consecuencia, se dice que la complejidad es de un polinomio de segundo grado.

Diferentes algoritmos pueden tener complejidades iguales o similares, como se enseña a continuación en el algoritmo 1.3.

Algoritmo 1.3. Ejemplo de dos ciclos anidados con diferentes números de repeticiones cada uno

Entrada: n	
Salida: c	

Repetición(n){
1: c = 0;
2: for i = 1 to n do
3: for j = 1 to i do
4: c++;
5: end for
6: end for
7: return c;
8: }
	int Repetición(int n){
	int i,j;
	int c=0,
	
	for(i=1; i<=n; i++)
	for(j=1; j<=i; j++)
	c++;
	return c;
	}

Del mismo modo que en el algoritmo, las líneas 1 y 7 se ejecutan una vez. La línea 2 se ejecuta *n*+1 veces. Para el ciclo controlado por j, sus repeticiones están en función del valor de i. De esta manera, cuando *i*=1, entonces la línea 4 se ejecuta una vez; cuando *i*=2, entonces la línea 4 se ejecuta 2 veces, pero hay que sumarle la cantidad de veces que se ejecutó cuando *i*=1; por lo tanto, el total es 1+2=3. Cuando *i*=3, entonces la línea 4 se ejecuta 3 veces, pero sumando la cantidad de veces que se ejecutó cuando *i*=1 y *i*=2, el total es 1+2+3=6.

Por inducción se establece que la línea 4 se repite $\sum_{j=1}^{i} j = \frac{i(i+1)}{2}$. pero como i=n, entonces $\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$.

Así, la suma se establece como $f(n) = 1 + n + 1 + n(n+1) + \frac{n(n+1)}{2} + 1;$ reduciendo términos, se obtiene $f(n) = \frac{3}{2}n^2 + \frac{5}{2}n + 3.$

Por lo tanto, la complejidad de este algoritmo es un polinomio de segundo grado. Véase que los polinomios del y del tienen el mismo grado, aunque sean diferentes los polinomios. Esto significa que los dos tienen la misma complejidad, mientras que en el Algoritmo 1.1 la complejidad es menor porque su polinomio es de inferior orden (es decir, realiza menos pasos u operaciones).

En estos casos, interesa conocer cuál es la cantidad de pasos que realiza un algoritmo para procesar una entrada de tamaño *n*, y no necesariamente cuál corre más rápido. Es decir, lo importante no es saber si un mismo algoritmo corre más rápido en C/C++, en Java, en algún lenguaje especial o en algún *hardware* específico, sino la cantidad de operaciones que realiza el algoritmo.

Esta tarea, por supuesto, no es sencilla, pues lo que se busca obtener es la complejidad, es decir, establecer funciones que representen el orden de crecimiento de las operaciones en función del tamaño de la entrada. Por ejemplo, en el caso del algoritmo 1.1, el orden de crecimiento es el de una función lineal o de primer orden, mientras que el orden de crecimiento del algoritmo 1.2 y del algoritmo 1.3 es el de una función cuadrática o de segundo orden. Como se verá en los siguientes capítulos —en términos de análisis de algoritmos—, aunque los polinomios son distintos, se dice que ambos algoritmos tienen la misma complejidad.

Informalmente, un algoritmo se define como un procedimiento computacional que toma un conjunto de datos como entrada y genera otro conjunto de datos como salida. Por ende, un algoritmo es una secuencia de pasos computacionales que transforma la entrada en salida. El algoritmo describe un procedimiento computacional específico para encontrar la relación entrada-salida. Un ejemplo tradicional para presentar el tema del diseño de los algoritmos se basa en ordenar una lista de números de forma no creciente.

Entrada: Una lista de número $(c_1, \dots, c_n)$.

Salida: Reordenamiento de la secuencia de numero tal que $c'_1 \leq c'_2 \leq \cdots \leq c'_n$.

METODOLOGÍA

Para enfocar la comparación de algoritmos se seguirán los siguientes pasos:

1. Averiguar la función f(*n*) que caracteriza los recursos requeridos por un algoritmo en función de tamaño n de los datos a procesar.
2. Dadas dos funciones f(*n*) y g(*n*), se define una relación de orden entre ellas, que llamaremos *complejidad*, lo cual determinará cuándo una función es más compleja que otra o cuándo son equivalentes.
3. Seleccionar una serie de funciones de referencia para situaciones típicas.
4. Definir conjuntos de funciones que llamaremos *órdenes de complejidad*.

En el paso uno se decide cómo medir *N*. Decidimos el recurso que nos interesa:

Tiempo de ejecución = *f(n)*.

A partir de aquí se analizará *f(n)*. Como se verá más adelante, calcular la fórmula analítica exacta que caracteriza a un algoritmo puede ser bastante laborioso. Habitualmente, no es necesario conocer el comportamiento exacto, sino que basta con conocer una cota superior, es decir, alguna función que se comporte "aun peor". De esta forma, se puede afirmar que el programa práctico nunca superará una cierta cota.

En el paso dos se decide la complejidad:

Dadas dos funciones *f(n)* y *g(n)*, se dirá que *f(n)* es más compleja que *g(n*) cuando

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Asimismo, se determinará que *f(n)* es menos compleja que *g(n)* cuando

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Por último, se afirmará que f(n) es equivalente a g(n) cuando

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = K, \text{ siendo } \mathrm{K} \neq 0 \text{ y } \mathrm{K} \neq \infty.$$

Nótese que esta definición permite realizar no solo un análisis algorítmico conociendo la formulación de la función, sino también un análisis experimental observando los recursos consumidos para valores crecientes de *N*.

En el paso tres se suelen manejar las siguientes funciones:

$f(n) = 1$	constante
$f(n) = \log(n)$	logaritmo
$f(n) = n$	lineal
$f(n) = n \log(n)$	
$f(n) = n^2$	cuadrática
$f(n) = n^a$	polinomio (de grado $a > 2$)
$f(n) = a^n$	exponencial ($a > 1$)
$f(n) = n!$	factorial
$f(n) = n^n$	combinatorio

¿Por qué estas y no otras? Simplemente porque son sencillas y porque la experiencia ha demostrado que aparecen con mayor frecuencia, de ahí que sea poco habitual que se requieran otras.

En el paso cuatro (orden de complejidad), a un conjunto de funciones que comparten un mismo comportamiento asintótico se le denominará *orden de complejidad*. A estos conjuntos regularmente se les llama $\mathcal{O}$ (gran $\mathcal{O}$), y existe una infinidad de ellos; aun así, solo nos centraremos en algunos de uso frecuente. Para cada uno de estos conjuntos se suele identificar un miembro f(n) que se utiliza como representante de la clase (tabla 1.2).

Tabla 1.2. Conjuntos u órdenes de complejidad

Orden	Nombre	Comentario
O (1)	Constante	Todo aquel algoritmo que responde en un tiempo constante. Son los que aplican alguna fórmula sencilla.
O (log n)	Logarítmico	El tiempo crece con un criterio logarítmico, independientemente de cuál sea la base mientras esta sea mayor que 1. Esto implica que un bucle realiza menos iteraciones que la talla del problema. Por ejemplo, la búsqueda binaria en un vector ordenado (ver el capítulo *Divide y conquistarás*).
O (n log(n))	Linearítmico	Es un orden relativamente bueno. En este orden está el algoritmo de ordenamiento *merge sort* (ver el capítulo *Divide y conquistarás*).
O (n^c)	Polinómico	Cuando c = 2, se le conoce como cuadrático; cuando c = 3, se le llama cúbico. Intuitivamente, se podría decir que este orden es el último aceptable. A partir del siguiente, en la práctica los algoritmos son complicados de tratar.
O (c^n) c>1	Exponencial	Conjunto de algoritmos que crecen mucho más rápido que el polinomial. Es un problema intratable (ver torres de Hanói).
O (n!)	Factorial	El algoritmo prueba todas las combinaciones posibles (ver determinante por menores y cofactores).
O (n^n)		Tan intratable como el anterior. A menudo no se hace distinción entre ellos.

La definición matemática de estos conjuntos debe ser muy cuidadosa para involucrar los dos aspectos antes comentados:

- identificación de una familia y
- posible utilización como cota superior de otras funciones menos malas.

COTA SUPERIOR ASINTÓTICA

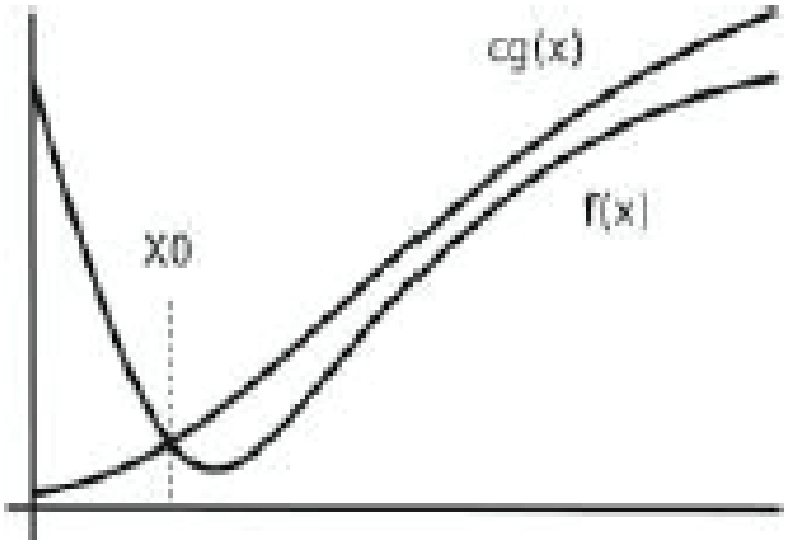
La notación $\mathcal{O}$ (gran $\mathcal{O}$), $\mathcal{O}$ (g(n)) es el conjunto de todas las funciones f_i para las cuales existen constantes enteras positivas c y n_o tales que para $n >= n_0$ se cumple lo siguiente:

$$f_i(n) \leq cg(n)$$

cg(n) es una "cota superior" de toda f_i para $n \geq n_0$.

En la figura 1.1 se presenta un ejemplo esquemático de cómo se comporta {\displaystyle cg(x)} con respecto a f(x) cuando x tiende a infinito. Nótese además que dicho conjunto es no vacío, pues *f(x) = O(-g(x)).*

Figura 1.1. f(x)EO(g(x))



Si *f(n) cg(n)* para todo $n > n_0$, implica que *f(n)* es $\mathcal{O}$ *(g(n))*. Se dice, entonces, que cuando el valor de n se hace grande, *f(n)* está acotada por *cg(n)*. Este es un concepto importante en la práctica, ya que cuando el tiempo de ejecución de un algoritmo está acotado por alguna función, se puede predecir un máximo de tiempo para que termine en función de *n*.

Si $A(n) = a_m n^m + ... + a_1 n + a_o$, entonces $A(n) \approx \mathcal{O}(n^m)$, siendo *A(n)* el número de pasos de un algoritmo determinado y $O(n^m)$, donde $m = máx \{m_i\}$, $1 \leq i \leq k$.

Como ejemplo, para comprender el orden de magnitud, supongamos que se tienen dos algoritmos para la misma tarea: uno del $\mathcal{O}$ (n^2) y otro del $\mathcal{O}$ (*n log n*). Si $n = 1024$ se requiere para el primer algoritmo 1 048 576 operaciones, mientras que para el segundo se necesitan 10 241 operaciones. Ahora bien, si la computadora toma un microsegundo para realizar cada operación, el algoritmo uno requerirá aproximadamente 1.05 segundos, mientras que el segundo necesitará aproximadamente

0.0102 segundos para la misma entrada. Así, el orden de complejidad queda de la siguiente manera:

$\mathcal{O}(1) < \mathcal{O}(\log n) < \mathcal{O}(n) < \mathcal{O}(n \log n) < \mathcal{O}(n^2) < \mathcal{O}(n^3) < \mathcal{O}(2^n) < \mathcal{O}(n!)$

Nota: La base del logaritmo es dos ($\text{Log}_b N = \text{Log}_a N / \text{Log}_b b$).

Un algoritmo exponencial es práctico solo con un valor pequeño de *n* (tabla 1.3).

Tabla 1.3. Complejidad algorítmica

Log(n)	n	n log (n)	N^2	N^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	6536
5	32	160	1024	32768	4294967296

Otra tabla comparativa es la 1.4, en la que se supone que la computadora puede hacer un millón de operaciones por segundo:

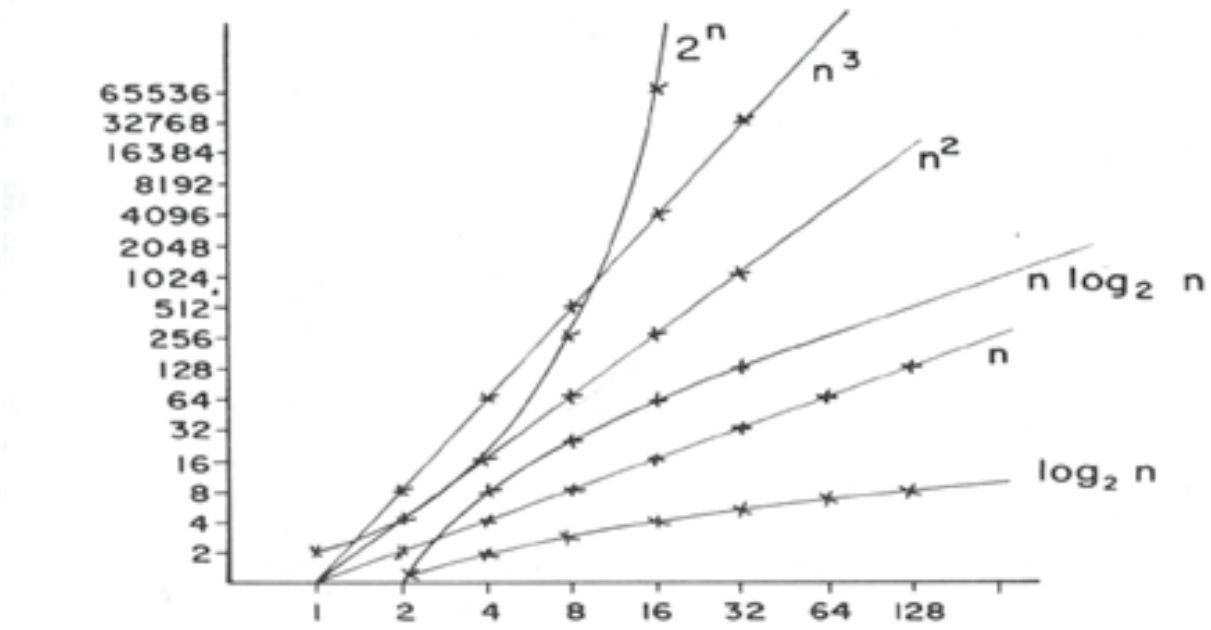
Tabla 1.4. Tiempo en segundos que tarda en realizar *f(n)* operaciones

f(n) \ n	10	20	30	40	50	70	100
n	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s	0.00007 s	0.0001 s
n log(n)	0.00003 s	0.00008 s	0.00014 s	0.00021 s	0.00028 s	0.00049 s	0.0006 s
n^2	0.0001 s	0.0004 s	0.0009 s	0.0016 s	0.0025 s	0.0049 s	0.01 s
n^3	0.001 s	0.008 s	0.027 s	0.064 s	0.125 s	0.343 s	1 s
n^4	0.01 s	0.16 s	0.81 s	2.56 s	6.25 s	24 s	1.6 min
n^5	0.1 s	3.19 s	24.3 s	1.7 s	5.2 min	28 min	2.7 horas
$n^{Log(n)}$	0.002 s	4.1 s	17.7 s	5 min 35 s	1 h 4min	2.3 días	224 días
2^n	0.001 s	1.04 s	17 min	12 dias	35.6 a.	37 Ma	2.6 MEU
3^n	0.059 s	58 min	6.5 a.	385495 a.	22735 Ma	52000 Ma	10^{18} MEU
n!	3.6 s	77054 a.	564 MEU	$1.6\ 10^{18}$ME	$6\ 10^{34}$	$2.4\ 10^{70}$M	2
n^n	2.7 horas	219 EU	$4.2\ 10^{14}$ME	$2.4\ 10^{28}$ME	$1.8\ 10^{67}$ME	$3\ 10^{111}$ M	$2\ 10^{182}$M

s: segundos min: minutos h: horas a: añosMa: millones de años
EU: edad del universo MEU: millones de veces la edad del universo

En la figura 1.2 se puede observar gráficamente el crecimiento de las principales funciones de complejidad temporal.

Figura 1.2. Gráfica de las principales funciones de complejidad temporal



COTA INFERIOR ASINTÓTICA

Si la notación $\mathcal{O}$ sirve para indicar una cota superior, también se puede definir una cota inferior.

Ω(g(n)) es el conjunto de todas las funciones f_i para las cuales existen constantes enteras positivas c y n_o tales que para cada $n \geq n_o$ se cumple lo siguiente:

$$f_i(n) \quad cg(n)$$

cg(n) es una "cota inferior" de toda f_i para toda $n \geq n_o$.

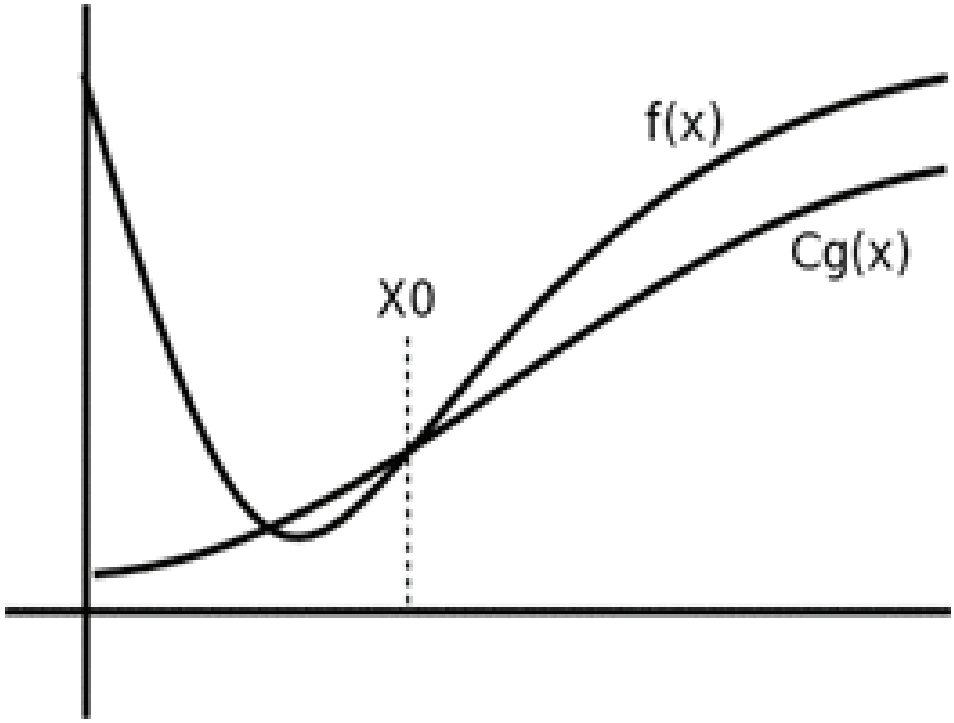
Si el tiempo de ejecución *T(n)* de un programa es $\Omega(n^2)$ implica lo siguiente:

$$T(n) \geq c\, n^2,$$

donde c y n_o son constantes enteras y positivas.

Entonces, $T(n)$ $\Omega(n^2)$ significa que el programa nunca tardará menos que cn^2. En la figura 1.3 se observa que *cg(x)* acota por debajo a *f(x)* a partir de x_o; *f(n)* crece asintóticamente más rápido que *g(n)* cuando $n \to \infty$.

Figura 1.3. Cota inferior asintótica

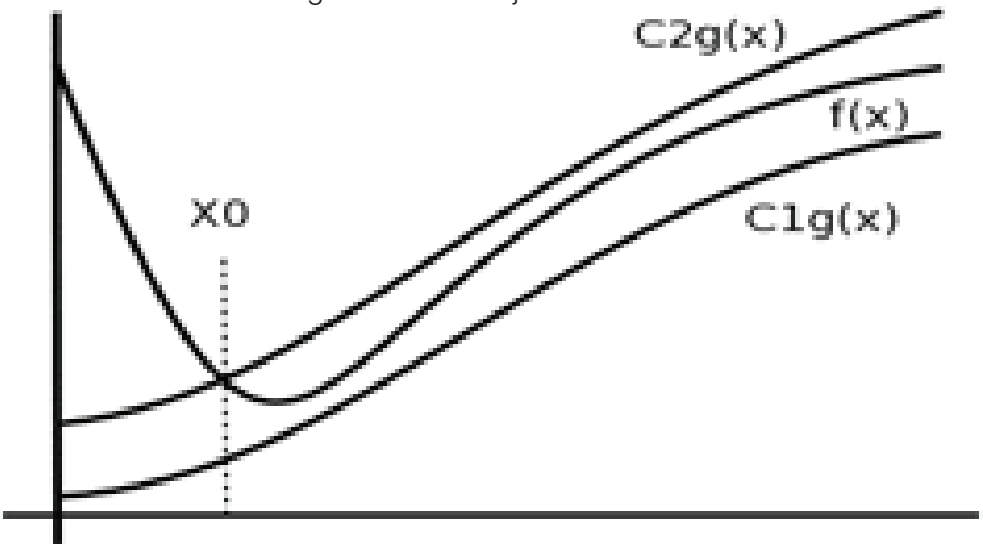


COTA AJUSTADA ASINTÓTICA

Si $f(n) = \Omega(g(n))$ y $f(n) = \mathcal{O}(g(n))$, entonces $f(n) = \Theta(g(n))$ si y solo si existen constantes positivas c y n_o tal que para toda $n > n_o$, $C_1 |g(n)| \leq |f(n)| \leq C_2|g(n)|$.

Esto indica que el peor y el mejor caso marcan la misma cantidad de tiempo (figura 1.4).

Figura 1.4. Cota ajustada asintótica



Por ejemplo, un algoritmo que busca el máximo en *n* elementos desordenados siempre realizará n-1 iteraciones, por lo tanto:

$$\Theta(n).$$

Ahora bien, un algoritmo que busque en un arreglo un valor determinado.

$\mathcal{O}(n)$ $\Omega(1)$

Aunque no existe una receta que funcione siempre para calcular la complejidad de un algoritmo, sí es posible tratar sistemáticamente una gran cantidad de ellos, ya que suelen estar bien estructurados y seguir pautas uniformes.

Los algoritmos bien estructurados combinan las sentencias según alguna de las siguientes formas:

- Sentencias sencillas

  Son las sentencias de asignación, entrada/salida, etc., siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño esté relacionado con el tamaño *N* del problema. La mayoría de las sentencias de un algoritmo requiere un tiempo constante de ejecución, siendo su complejidad $\mathcal{O}$ (1).

- Secuencia

  La complejidad de una serie de elementos de un programa es del orden de la suma de las complejidades individuales, aplicándose las operaciones arriba expuestas.

- Decisión

  La condición suele ser de $\mathcal{O}$ (1), complejidad a sumar con la peor posible, bien en la rama THEN, o bien en la rama ELSE. En decisiones múltiples (ELSIF, CASE) se tomará la peor de las ramas.

- Bucle

  En los bucles con contador explícito se pueden distinguir dos casos: el tamaño *N* forma o no forma parte de los límites.

  Si el bucle se realiza un número fijo de veces, independiente de *N*, entonces la repetición solo introduce una constante multiplicativa que puede absorberse.

for (int i= 0; i < K; i++)

algo_de_ $\mathcal{O}$ (1);

K * $\mathcal{O}$ (1) = $\mathcal{O}$ (1)

Si el tamaño *N* aparece como límite de iteraciones, surgen varios casos:

caso 1:

for (int i= 0; i < *N*; i++)

algo_de_ $\mathcal{O}$ (1);

$N * \mathcal{O}(1) = \mathcal{O}(n)$

caso 2:

for (int i= 0; i < *N*; i++)

for (int j= 0; j < *N*; j++)

algo_de_ $\mathcal{O}$ (1);

$N * N * \mathcal{O}(1) = \mathcal{O}(n^2)$

caso 3:

for (int i= 0; i < *N*; i++)

for (int j= 0; j < i; j++)

algo_de_ $\mathcal{O}$ (1)

El bucle exterior se realiza N veces, mientras que el interior se realiza 1, 2, 3,... *N* veces, respectivamente. En total, se consigue la suma de una serie aritmética:

$1 + 2 + 3 + ... + N = N * (1+N) / 2 > O(n^2)$

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores):

int c = 1;

while (c < N) {

algo_de_ O (1);

c*= 2;

}

El valor inicial de c es 1, siendo 2^k al cabo de k iteraciones. El número de iteraciones es tal que 2^k *N* => $k = \log_2(N)$,

y, por tanto, la complejidad del bucle es $\mathcal{O}$ (log n).

```
c = N;
while (c > 1) {
algo_de_ O(1);
c/= 2;
}
```

Un razonamiento análogo nos lleva a log(*N*) iteraciones y, por tanto, a un orden O (log n) de complejidad.

```
for (int i = 0; i < N; i++) {
c = i;
while (c > 0) {
algo_de_ O(1);
c/= 2;
}
}
```

Tenemos un bucle interno de orden $O(\log_2 n)$ que se ejecuta *N* veces, luego el conjunto es de orden $O(n \log n)$.

- Llamadas a procedimiento

La complejidad de llamar a un procedimiento viene dada por la complejidad del contenido del procedimiento en sí. El coste de llamar no es sino una constante que se puede obviar inmediatamente dentro de los análisis asintóticos.

El cálculo de la complejidad asociada a un procedimiento puede complicarse notablemente si se trata de procedimientos recursivos.

- Relaciones de recurrencia

En el cálculo de la complejidad de un algoritmo a menudo aparecen expresiones recursivas para estimar el tiempo que se tarda en procesar un problema de un cierto tamaño N.

Por ejemplo, se puede hallar un programa que para resolver un problema de tamaño N resuelve dos problemas de tamaño N/2 y luego combina las soluciones parciales con una operación de complejidad $O(n)$.

La relación de recurrencia es esta:

$T(n) = 2\ T(n/2) + \mathcal{O}(n)$

$T(1) = \mathcal{O}(1)$

Y el problema es cómo deducir, a partir de esa definición, la complejidad del algoritmo completo. Para resolverlo, se van aplicando pasos sucesivos de recursión:

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

$T(n) = 2\ (2\ T(n/4) + n/2) + n = 4\ T(n/4) + 2n$

$T(n) = 8\ T(n/8) + 3n$

hasta que se vea un patrón

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

Esta fórmula es válida siempre, en particular cuando $n/2^k = 1$ y podemos escribir

$T(n) = 2^k\ T(1) + k\ n$

y como sabemos la condición de parada de la recursión

$T(n) = 2^k + k\ n$

El valor de k que termina la recursión es:

$n/2^k = 1 \Rightarrow k = \log2(n)$,

sustituyendo

$T(n) = 2^{\log_2(n)} + \log2(n)\ n$

$T(n) = n + n\ \log(n) \Rightarrow \mathcal{O}(n\ \log(n))$

El método descrito permite resolver numerosas relaciones de recurrencia. Algunas son muy habituales (tabla 1.5).

Tabla 1.5. Casos de complejidad

Relación	Complejidad	Ejemplos
$T(n) = T(n/2) + O(1)$	$O(\log n)$	Búsqueda binaria
$T(n) = T(n-1) + O(1)$	$O(n)$	Búsqueda lineal factorial *Bucles for, while*
$T(n) = 2\,T(n/2) + O(1)$	$O(n)$	Recorrido de árbol binario en preorden, inorden y postorden.
$T(n) = 2\,T(n/2) + O(n)$	$O(n \log n)$	Quick Sort
$T(n) = T(n-1) + O(n)$	$O(n^2)$	Ordenación por selección y ordenación por burbuja
$T(n) = 2\,T(n-1) + O(1)$	$O(2^n)$	Torres de Hanói

Nota: Para tener una mayor profundidad matemática del tema, ver capítulo 9, sección 5 en adelante.

APLICACIONES EN C

Un problema se puede resolver de distintas formas o con diferentes algoritmos, cada uno de los cuales tiene su complejidad específica, como se muestra en los siguientes ejemplos.

Ejemplo 1. Diseñar un programa para evaluar un polinomio P(x) de grado N

Sea coef el vector de coeficientes:

Opción A: Desarrollo del algoritmo en forma iterativa (algoritmo 1.4).

Algoritmo 1.4. Evaluación de un polinomio P(x) en forma iterativa

```
#include <stdio.h>
#include <stdlib.h>
float evaluaPolinomio(float*,int, float);
int main(){
float * poli, x;
int tam,j;
printf("Da el tamaño del polinomio=>");
scanf("%d",&tam);
poli=(float*)malloc(sizeof(float)*(tam+1));
for (j=0;j<=tam;j++){
   printf("Da el coeficiente de la potencia %d=>",tam-j );
   scanf("%f",&poli[j]);
}
printf("Da el valor de x a evaluar=>");
scanf("%f",&x);
printf("P(%f)=%f\n",x,evaluaPolinomio(poli,tam,x));
getchar();
getchar();
}

float evaluaPolinomio(float * coef, int tam, float x) {
float total = 0;
for (int i = tam; i >=0; i--) {
   float xn = 1.0;
   for (int j = 0; j <i; j++)
    xn *= x;
 total += coef[tam-i] * xn;
 printf("\nTotal=%f",total);
   }
return total;
}
```

Como medida del tamaño tomaremos para *N* el grado del polinomio, que es el número de coeficientes. Así pues, el bucle más exterior se ejecuta N veces. El bucle interior se emplea para elevar a X la potencia indicada por la variable i, respectivamente. Por lo que:

$$C = (n - 1) + (n - 2) + \ldots + 2 + 1$$

Ahora bien, utilizando el truco de Gauss para la suma de números naturales se tiene:

$$\begin{aligned} S_n &= 1 + 2 + 3 + 4 + \ldots + (n-3) + (n-2) + (n-1) + n \\ S_n &= n + (n-1) + (n-2) + (n-3) + \ldots + 4 + 3 + 2 + 1 \\ 2S_n &= (n+1) + (n+1) + (n+1) + (n+1) + \ldots + (n+1) + (n+1) + (n+1) + (n+1) \end{aligned}$$

$2S_n = n^*(n + 1)$, por lo tanto, $S_n = n^* (n + 1) /2$

Utilizando la misma idea, se tiene:

$$S_n = 1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$$
$$S_n = (n-1) + (n-2) + (n-3) + (n-4) + \ldots + 2 + 1$$
$$2S_n = (n-1) + (n-1) + (n-1) + (n-1) + \ldots + (n-1) + (n-1) + (n-1)$$

$2S_n = n^* (n - 1)$, por lo tanto, $S_n = n^* (n - 1) /2$. De esta forma se tiene:

$$C = n^* (n - 1) /2 = (n^2 - n) /2 \Rightarrow O (n^2)$$

Opción B: Utilizando un algoritmo recursivo para calcular la potencia de X^n (algoritmo 1.5).

Algoritmo 1.5. Evaluación de un polinomio P(x) en forma recursiva

```
#include <stdio.h>
#include <stdlib.h>

float evaluaPolinomio(float*,int, float);
float potencia(float , int );

int main(){      float * poli, x;
int tam,j;
printf("Da el tamaño del polinomio->");
scanf("%d",&tam);
poli=(float*)malloc(sizeof(float)*(tam+1));
for (j=0;j<=tam;j++){
   printf("Da el coeficiente de la potencia %d->",tam-j );
   scanf("%f",&poli[j]);
}
printf("Da el valor de x a evaluar->");
scanf("%f",&x);
printf("P(%f)=%f\n",x,evaluaPolinomio(poli,tam,x));
getchar();
getchar();
}

float evaluaPolinomio(float * coef, int tam, float x) {
float total = 0;
for (int I = tam; I >=0; i--) {
   total += coef[tam-i] * potencia(x,i);
   printf("\nTotal=%f",total);
   }
return total;
}

float potencia(float x, int y) {
float t;
if (y == 0)
   return 1.0;
if (y % 2 == 1)
   return x * potencia(x, y - 1);
else {
   t = potencia(x, y / 2);
   return t * t;
   }
}
```

El análisis del método potencia es delicado, pues si el exponente es par, el problema tiene una evolución logarítmica; en cambio, si es impar, su evolución es lineal. No obstante, como si *j* es impar, entonces *j*-1 es par; el caso peor es que en la mitad de los casos tengamos *j*"impar y en la otra mitad par. El caso mejor, por contra, es que siempre sea *j* par.

Un ejemplo de caso peor sería x ^ 31, que implica la siguiente serie para *j*:

31 30 15 14 7 6 3 2 1,

cuyo número de términos podemos acotar superiormente por

2 * Valor_Superior ($\log_2$ (j)),

donde Valor_Superior(x) es el entero inmediatamente superior (este cálculo responde al razonamiento de que en el caso mejor visitaremos Valor_Superior ($\log_2$ (j)) valores pares de *j*; y en el caso peor podemos encontrarnos con otros tantos números impares entremezclados).

Por tanto, la complejidad de potencia es de orden *O* (*log n*). Otra forma de calcularlo es replantear la función recursiva, como se muestra en el algoritmo 1.6.

Algoritmo 1.6. Evaluación del polinomio P(*x*) en forma recursiva

```
#include <stdio.h>
#include <stdlib.h>
float evaluaPolinomio(float*,int, float);
float potencia(float , int );
int main(){      float * poli, x;
int tam,j;
printf("Da el tamaño del polinomio->");
scanf("%d",&tam);
poli=(float*)malloc(sizeof(float)*(tam+1));
for (j=0;j<=tam;j++){
   printf("Da el coeficiente de la potencia %d->",tam-j );
   scanf("%f",&poli[j]);
   }
printf("Da el valor de x a evaluar->");
scanf("%f",&x);
printf("P(%f)=%f\n",x,evaluaPolinomio(poli,tam,x));
getchar();
getchar();
}

float evaluaPolinomio(float * coef, int tam, float x) {
float total = 0;
for (int i = tam; i >=0; i--) {
total += coef[tam-i] * potencia(x,i);
printf("\nTotal=%f",total);    }
return total;
}

float potencia(float x, int y) {
   if (y == 0)          return 1.0;
   if (y == 1)          return x;
   if (y % 2 == 1)             return x * potencia(x * x, y / 2);
else
return potencia (x * x, y / 2);
}
```

Visto así, tenemos una función de recurrencia

$$T(n) = \mathcal{O}(1) + T(n / 2),$$

cuya solución es

$$\mathcal{O}(\log n).$$

Insertado el método auxiliar potencia en el método evaluaPolinomio, la complejidad compuesta es del orden $\mathcal{O}(n \log n)$, al multiplicarse por N un subalgoritmo de $\mathcal{O}(\log n)$.

Opción C: División sintética

Para calcular el valor del polinomio se puede utilizar la división. El residuo de la división es el valor P(x). Por ejemplo, si el polinomio a calcular es $P(x) = 3x^3 - 2x^2 + x - 6$, para $x = 2$, la división será:

$$\begin{array}{r|l} & 3x^2 + x + 2 \\ \hline x - 2 & 3x^3 - 2x^2 + x - 6 \\ & \underline{-3x^3 + 3x^2} \\ & \quad x^2 + x \\ & \quad \underline{-x^2 + x} \\ & \quad\quad 2x - 6 \\ & \quad\quad \underline{-2x + 2} \\ & \quad\quad\quad -4 \end{array}$$

Por tanto, $P(2) = -4$.

Como se aprecia, se debe dividir frecuentemente el polinomio entre el binomio $(x - a)$ para conocer el residuo, es ineficiente calcular cada uno de los $k+1$ términos separadamente y luego sumarlos. Esto indica n adiciones y $n(n + 1)/2$ multiplicaciones. Por ende, para simplificar el proceso, se usará el criterio de la división sintética que consiste en lo siguiente:

Si

$$Q(X) = A_0 X^{n-1} + A X^{n-2} + A_2 X^{n-3} + \ldots A_{n-2} X + A_{n-1},$$

y R es el residuo que resulta de dividir el polinomio

$$P(X) = a_0 X^n + a_1 X^{n-1} + a_2 X^{n-2} + ... + a_n$$

entre el binomio X-a, entonces:

$$P(X) = (X - a) Q(X) + R$$

o sea,

$$a_0 X^n + a_1 X^{n-1} + a_2 X^{n-2} + ... a_n = A_o X^n + (A_1 - a A_0) X^{n-1} + ... + (R - a A_{n-1})$$

Siendo los polinomios de ambos miembros idénticos, sus coeficientes deben ser iguales entre sí; por lo tanto:

$a_0 = A_0$

$a_1 = A_1 - a\,A_0$

$a_2 = A_2 - a\,A_1$

.

.

$a_n = R - a\,A_1$

De esta forma:

$A_0 = a_0$

$A_1 = a_1 + a\,A_0$

.

.

$R = a_n + a\,A_{n-1}$

Pudiéndose arreglar los cálculos de la siguiente forma:

a	a_0	a_1	a_2	...	a_{n-1}	a_n
		aA_0	aA_1	...	aA_{n-2}	aA_{n-1}
	A_0	A_1	A_2	...	A_{n-1}	R

El algoritmo 1.7 en lenguaje C es el siguiente:

Algoritmo 1.7. Evaluación del polinomio P(x) por división sintética

```
#include <stdio.h>
#include <stdlib.h>
float divide(float*,int, float);
int main(){
float * poli, x;
int tam,j;
printf("Da el tamaño del polinomio=>");
scanf("%d",&tam);
poli=(float*)malloc(sizeof(float)*(tam+1));
for (j=0;j<=tam;j++){
   printf("Da el coeficiente de la potencia %d=>",tam-j );
   scanf("%f",&poli[j]);
}
printf("Da el valor de x a evaluar=>");
scanf("%f",&x);
printf("P(%f)=%f\n",x,divide(poli,tam,x));
getchar();
getchar();
}

float divide(float p[], int n, float x){
float a=p[0];
for(int i=1;i<=n;i++)
   a=a*x+p[i];
return(a);
}
```

Siendo la complejidad del algoritmo $\mathcal{O}$ (n).

EJEMPLO 2. CÁLCULO DE NÚMEROS DE FIBONACCI

Opción A: En forma recursiva

Para comprender un poco más la complejidad algorítmica se usará una famosa secuencia de números desarrollada por Fibonacci:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

Formalmente hablando, los números de Fibonacci F_n son generados por una regla simple:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{Si } n > 1 \\ 1 & \text{Si } n = 1 \\ 0 & \text{Si } n = 0 \end{cases}$$

No existe otra secuencia de números que haya sido estudiada de forma tan extensa o aplicada a más campos del conocimiento (biología, demografía, arte, arquitectura, música, entre otros). De hecho, en ciencias de la computación es una de las secuencias favoritas, junto con la potencia de dos.

Para estimar la complejidad algorítmica se utilizará el número de oro o número áureo. Este es conocido desde la época de los griegos y también tiene otros nombres como *proporción dorada, divina proporción*, etc. El número en cuestión se define así: teniendo un segmento recto, se divide en dos partes que cumplan lo siguiente:

$$a/b = (a+b)/a = 1.6180339887\ldots = \varphi$$

Otra forma de encontrar la proporción dorada es utilizando la sucesión de Fibonacci:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 37

Si uno divide a un número entre el anterior, se acerca cada vez más a la famosa proporción áurea.

$$144/89 = 1.617977\ldots$$

$$233/144 = 1.61865\ldots$$

$$377/233 = 1.618025\ldots$$

Ahora bien, la tarea para el cálculo de la complejidad no es fácil, pues la función del tiempo de ejecución del valor n es:

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

Podemos hacer una hipótesis e intentar corroborarla. Supongamos que T(N) responde a una función del siguiente tipo:

$$T(n) = k\, a^n.$$

Entonces, se debe satisfacer que

$$k\,a^n = k\,a^{n-1} + k\,a^{n-2}.$$

Realizando una simplificación, se tiene:

$$a^2 = a + 1,$$

de donde se despeja

$$a^2 – a -1 = 0,$$

siendo esta una ecuación característica y una de sus raíces se conoce como:

$$a = (1+\sqrt{5}) / 2 \sim 1{,}6180339887\ldots$$

o sea,

$$T(n) \sim 1{,}6180339887^n.$$

En otras palabras,

$$T(n) \in \mathcal{O}(1{,}618^n).$$

En realidad, la secuencia de Fibonacci crece tan rápido como la potencia de dos: por ejemplo, F_{30} rebasa el millón, y la secuencia F_{100} tiene aproximadamente veintiún dígitos de largo. Para colocar la complejidad en potencia de dos se realiza la siguiente trasformación:

$$1.618^n = 2^{x\,n}$$

$$\text{Log } (1.618)^n = \log (2)^{x\,n}$$

$$n \text{ Log } (1.618) = x\ n \log (2)$$

$$x = n \text{ Log } (1.618) / (n \log (2)) = 0.694.$$

En general, $F_n \approx 2^{0.694n}$.

Pero ¿cuál es el valor exacto de F_{100} o de F_{200}? Fibonacci nunca supo la respuesta. Para saberlo, necesitamos un algoritmo para computar el n-ésimo número de Fibonacci.

Una idea es utilizar la definición recursiva de F_n. El pseudocódigo se muestra enseguida:

```
Función fib(n){
Si n = 0 retorna 0
Si n = 1 retorna 1
Retorna fib(n-1) + fib(n-2)
}
```

El código del algoritmo 1.8 en C es el siguiente:

Algoritmo 1.8. La función Fibonacci escrita en forma recursiva

```
#include <stdio.h>
int fibo(int);
main(){
int i;
printf("Da el Numero 32ibonacci a calcular:");
scanf("%d",&i);
printf("Fibo(%d)=%d\n",i,fibo(i));
getchar();
}

int fibo(int n){
if(n<2)
   return n;
return fibo(n-1) + fibo(n-2);
}
```

Cada vez que se tiene un algoritmo surgen preguntas que se deben responder:

1. ¿Es correcto?
2. ¿Cuánto tiempo tarda en dar el resultado?
3. ¿Se puede mejorar?

La primera pregunta no se discute aquí, ya que el algoritmo es precisamente la definición de Fibonacci para F_n. Pero la segunda pregunta demanda una respuesta.

Sea T(*n*) el número de pasos computacionales necesarios para computar *fib(n)*, ¿qué podemos decir sobre esta función? Si $n < 2$, el procedimiento termina casi de forma inmediata, justo después de un par de pasos. Para números mayores existen invocaciones recursivas de *fib( )*, así tenemos lo siguiente:

El tiempo de ejecución del algoritmo crece tan rápido como los números de Fibonacci: *T(n)* es exponencial en *n*, lo cual implica que el algoritmo es impráctico, excepto para valores muy pequeños de *n*.

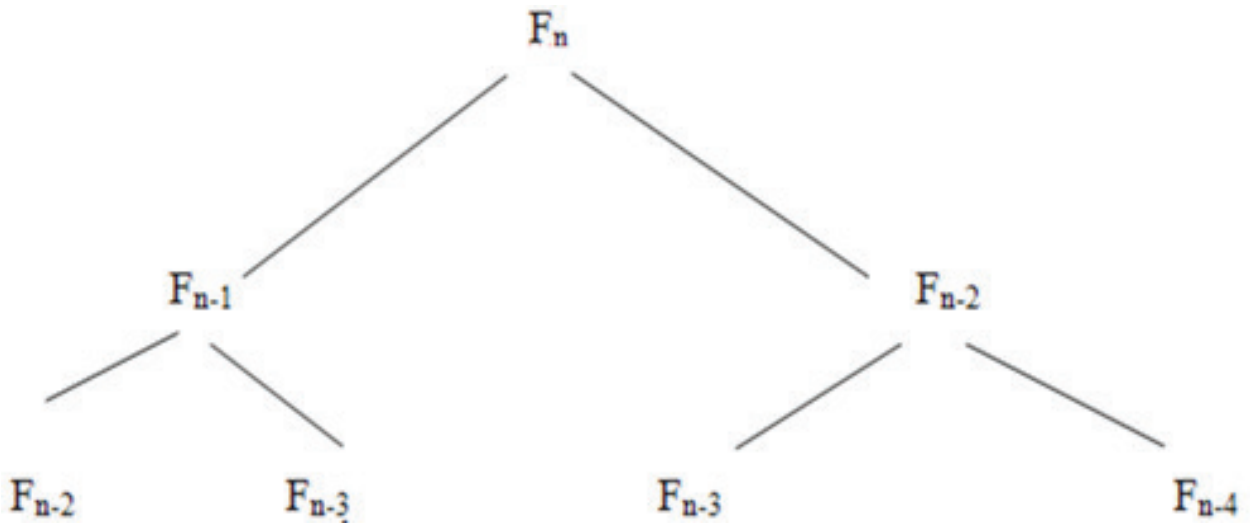
Por ejemplo, para calcular F_{200}, la función *fib( )* se ejecuta $T(200) \geq F_{200} \geq 2^{138}$ pasos elementales de cómputo. ¿Cuánto tiempo se requiere? Eso depende de la computadora usada. En este momento, una de las más veloces en el mundo es la NEC Earth Simulator, con 400 trillones de pasos por segundo. Pero aun para esta máquina, *fib*(200) tardará al menos 2^{92} segundos. Esto significa que si

iniciamos el cómputo hoy, estaría trabajando después de que el sol se tornara en una estrella roja gigante.

Pero la tecnología ha mejorado de tal forma que los pasos de cómputo se han duplicado aproximadamente cada 18 meses, fenómeno al cual se le conoce como ley de Moore. Con este extraordinario crecimiento, posiblemente la función fib se ejecutará más rápido para el próximo año. Verifiquemos este dato: el tiempo de ejecución de $fib(n) \approx 2^{0.694n} \approx (1.6)^n$, por lo que toma 1.6 veces más tiempo para computar F_{n+1} que F_n. Y desde la ley de Moore, el poder de cómputo crece aproximadamente 1.6 veces cada año.

Si nosotros podemos computar de forma razonable F_{100} con el crecimiento de la tecnología, el siguiente año se podrá calcular F_{101}, el próximo año F_{102} y así sucesivamente; es decir, solo un número de Fibonacci más cada año. Así, es el comportamiento de un tiempo exponencial. El algoritmo se vuelve ineficiente por la razón de que una llamada a *fib(n)* se tiene una cascada de llamadas recursivas en las cuales la mayoría de ellas son repetitivas (figura 1.5).

Figura 1.5. Árbol recursivo con la función de Fibonacci



Opción B: En forma lineal

En corto, nuestro algoritmo recursivo es correcto, pero sumamente ineficiente. ¿Podemos hacerlo mejor? El algoritmo 1.9 muestra en lenguaje C algo más sencillo:

Algoritmo 1.9. Programación de la función de Fibonacci en forma iterativa

```
#include <stdio.h>
main(){
int I, n, fibn_2,fibn_1, fibn;
printf("Da el valor a evaluar=>");
scanf("%d",&n);
fibn_2=0;
fibn_1=1;
for ( I =2;i<=n; i++){
   fibn=fibn_2+fibn_1;
   printf ("fibo (%d) =%d\n", I , fibn);
   fibn_2=fibn_1;
   fibn_1=fibn;
}
if (n<2)
printf ("fibo (%d) =%d\n", n , n);
getchar ();
}
```

¿Cuánto tiempo toma el algoritmo? El *loop* tiene solo un paso y se ejecuta n-1 veces, por lo que el algoritmo se considera lineal en n. De un tiempo exponencial, hemos pasado a un tiempo lineal: un gran progreso en tiempo de ejecución. Es ahora razonable calcular F_{200} o aun F_{200000}.

Opción C: En forma directa

Ahora bien, se puede realizar un algoritmo en forma directa, en este caso, la fórmula Binet (1786-1856) especifica que $Fib(n) = ((1+\sqrt{5})^n - (1-\sqrt{5})^n) / (2^n\sqrt{5})$, lo que el algoritmo 1.10 se puede programar de este modo:

Algoritmo 1.10. Programación de la función de Fibonacci utilizando la fórmula Binet

```
#include <stdio.h>
#include <math.h>
main(){
double x=sqrt(5);
int n;
```

```
printf("Da el valor de Fibo a calcular=>");
scanf("%d",&n);
double t1 = pow((1 + x) / 2, n);
double t2 = pow((1 - x) / 2, n);
int t = (t1 - t2) / x;
printf("Fibo(%d)=%d\n",n,t);
getchar();
}
```

Así, queda un algoritmo de tiempo constante:

$$T(n) \in \mathcal{O}(1).$$

EJEMPLO 3. TORRES DE HANÓI

El juego de las torres de Hanói involucra tres postes y n discos de diámetros 1, 2, 3,…, *n.* El juego inicia con todos los discos apilados sobre uno de los postes en orden creciente de la parte superior hacia abajo. El juego consiste en colocar todos los discos en otro poste obedeciendo las siguientes reglas:

1. Mover solo un disco a la vez.
2. Nunca colocar un disco sobre un disco de menor diámetro.

El origen del juego y la siguiente leyenda (quizá con alguna reserva) se atribuye a escritor inglés en matemáticas recreativas W. Rousse-Ball.

Cuenta la leyenda que en el gran templo de Benarés, bajo la cúpula que señala el centro del mundo, reposa una bandeja de cobre en la que están plantadas tres agujas cuyo diámetro es más fino que el aguijón de una abeja. En el momento de la creación, Dios colocó en una de las agujas 64 discos de oro puro ordenados por tamaño: desde el mayor que rebosa sobre la bandeja hasta el más pequeño, en lo más alto del montículo. Es la torre de Brahma. Incansablemente, día tras día, los sacerdotes del templo mueven los discos haciéndoles pasar de una aguja a otra, de acuerdo con las leyes fijas e inmutables de Brahma, las cuales evitan que el sacerdote en ejercicio mueva más de un disco al día o lo sitúe encima de un disco de menor tamaño. El día en que los 64 discos hayan sido trasladados desde la aguja en que Dios los puso al crear el mundo a cualquiera de las otras dos, ese día la torre, el templo y, con gran estruendo, el mundo desaparecerán (Dewdney, 1989).

Opción A: Solución recursiva

El pseudocódigo es el siguiente:

Hanói (N, Origen, Destino, Auxiliar)

Si N = 1

imprime “Mover disco de” Origen “a” Destino; si no

Hanói (N-1, Origen, Auxiliar, Destino)

imprime “Mover disco de” Origen “a” Destino

Hanói (N-1, Auxiliar, Destino, Origen) (Cairó y Guardati, 2000).

Y el código del algoritmo 1.11 se observa en lenguaje de programación C de la siguiente forma:

Algoritmo 1.11. Programación de las torres de Hanói

```
#include <stdio.h>
void hanoi(int, char *, char *, char *);
main(){ int n;
char *o,*d,*a;
o="origen";
d="destino";
a="auxiliar";
printf("Da el número de discos:");
scanf("%d",&n);
hanoi(n,o,d,a);
getchar();
getchar();
}

void hanoi(int n, char *o, char *d, char *a){
if (n>1){
   hanoi(n-1,o,a,d);
   printf("de %s a %s\n",o,d);
   hanoi(n-1,a,d,o);
}
else
   printf("de %s a %s\n",o,d);
}
```

Siendo *N* el número de anillos y *T* el tiempo para colocar los *N* anillos de la espiga *origen* a la espiga *destino*, se tiene:

$$T(N) = 2T(N-1) + 1$$

La estrategia es mover los *N-1* discos de una espiga a otra (con un costo *T(N-1)),* con lo cual queda en este momento el disco de mayor diámetro solo en una espiga, por lo que colocar tal disco a la espiga sin discos tiene un costo de una unidad. Posteriormente, se tienen que colocar los *N-1* discos sobre el disco de mayor diámetro (con un costo *T(N-1))* (ver figura 1.6). Por tanto, la condición de paro en la recursividad será la siguiente:

$$T(1) = 1.$$

De esta forma:

$T(N) = 2T(N-1) + 1 = 2(2T(N-2) + 1) + 1 = 4T(N-2) + 2 + 1$

$T(N-2) = 2T(N-3) + 1$

$T(N) = 4(2(T(N-3) + 1) + 2 + 1$

$= 8T(N-3) + 4+ 2+ 1$

Generalizando:

$T(N) = 2^K T(N-K) + \sum_{I=0}^{K-1} 2^i$

donde

$T(1) = 1$ y $K=N-1$

sabiendo que

$$\sum_{I=0}^{K-1} 2^i = 2^k - 1$$

se tiene:

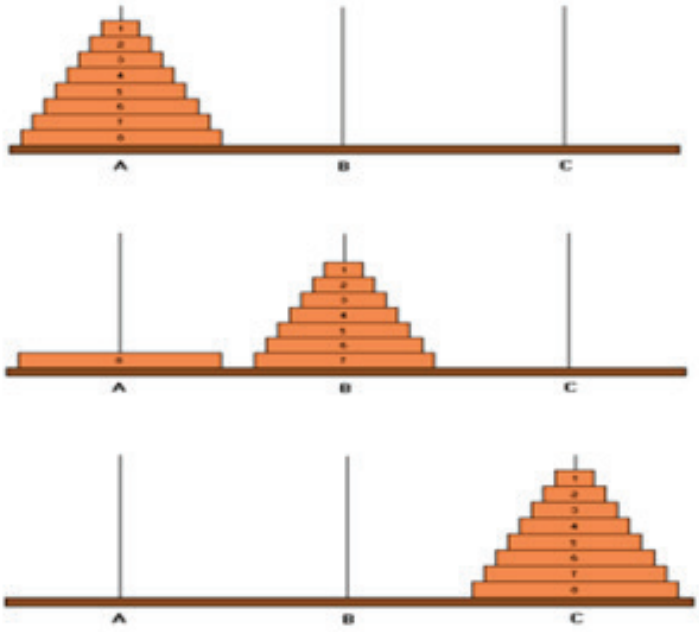
$T(N) = 2^{N-1} + 2^{N-1} - 1$

$T(N) = 2^{N-1}(1 + 1) -1 = 2^{N-1} (2) – 1$

$T(N) = 2^N - 1$

De esta forma, la complejidad del algoritmo es $\mathbb{Q}(2^N)$.

Figura 1.6. Torres de Hanói



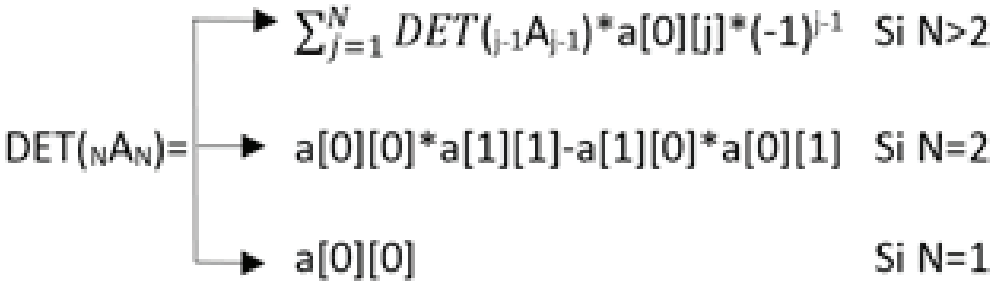
Si los monjes hicieran un movimiento por segundo, podrían pasar los 64 discos a la tercera espiga en unos 5849 millones de años. De acuerdo con las teorías científicas más creíbles, la tierra tiene aproximadamente 4400 millones de años, de ahí que reste muchísimo tiempo para disfrutar de las cosas de la vida. Hasta la fecha, este juego no tiene forma de mejorar su complejidad.

EJEMPLO 4. CÁLCULO DEL DETERMINANTE

Opción A: Por medio de menores y cofactores

El cálculo del determinante por menores y cofactores de una matriz se puede definir de forma recursiva, como se muestra en la figura 1.7:

Figura 1.7. Definición del determinante por menores y cofactores



En la llamada recursiva *DET($_{N-1}{}^{a}{}_{N-1}$)* los parámetros son *N-1*, por lo que en cada llamada en profundidad se elimina la primera hilera y la columna *j-1* de la matriz. Asimismo, se muestra el código que permite calcular el determinante en forma recursiva; en este caso, se utiliza la función *subm()*, que escoge para su eliminación la *primera* hilera y la *i-ésima* columna (la i-ésima columna la indica el comando *for* localizado en la función *determinante()*). El algoritmo 1.12 se muestra enseguida:

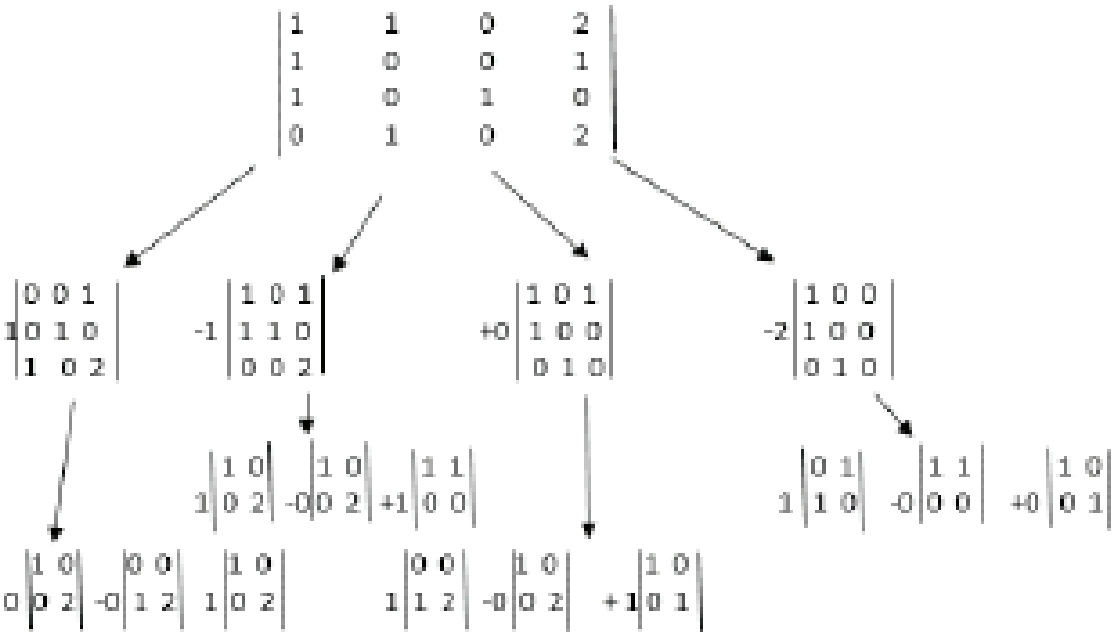
Algoritmo 1.12. Programación del determinante por cofactores y menores

```
#include <stdio.h>
#include <stdlib.h>
//PROTOTIPOS
int signo(int);
int determinante(int ** ,int);
int ** subm(int , int ** , int);
main(){
int f,i,j;
int **pm;
printf("Da el tamaño de la matriz=>");
scanf("%d",&f);
pm=(int **)malloc(sizeof(int *)*f);
for (j=0;j<f;j++)
   pm[j]=(int*)malloc(sizeof(int)*f);
for (i=0;i<f;i++)
   for (j=0;j<f;j++){
```

```
    printf("a[%d][%d]=",i,j);
 scanf("%d",&pm[i][j]);
   }
printf("LA MATRIZ ES:\n");
for (i=0;i<f;i++){
   for (j=0;j<f;j++)
    printf("%d\t",pm[i][j]);
putchar('\n');
}
printf("\nDeterminante(A)=%d\n", determinante (pm, f));
getchar();
getchar();
}

int determinante (int ** a, int tam){
if (tam==1)
   return a [0][0];
if (tam==2)
   return a [0][0] *a[1][1]-a [1][0] *a[0][1];
int m=0;
int tam1=tam-1;
for (int i=0;i<tam;i++)
   m=signo(i)*determinante(subm(i,a,tam1),tam1)*a[0][i]+m;
   return m;
}

int ** subm(int k, int ** a,int tam){
int ** b;
int i,j;
b=(int **)malloc(sizeof(int *)*tam);
for (j=0;j<tam;j++)
   b[j]=(int*)malloc(sizeof(int)*tam);
for (i=0; i<tam;i++){
   int m=0;
   for (j=0; j<tam+1;j++)
    if (j != k){
 b[i][m]=a[i+1][j];
 m++;
 }
}
return b;
}

int signo(int i){
if (i%2==0)
   return 1;
return -1;
}
```

Para una matriz de 4x4, en la primera etapa recursiva en profundidad se requieren cuatro llamadas recursivas con matrices de 3x3. En la segunda etapa, cada matriz de 3x3 requiere tres llamadas recursivas con matrices de 2x2. Siendo un criterio de paro, el cálculo de matrices de 2x2 con un costo de una unidad. Para una matriz de NxN, la primer etapa en profundidad tendrá N llamadas con matrices de tamaño (N-1)x(N-1); en la segunda llamada en profundidad, cada matriz de tamaño *(N-1)* x(N-1) tendrá *N-1* llamadas con matrices de tamaño (N-2)x(N-2), obteniéndose un orden de llamadas factorial (ver figura 1.8). Por eso, una matriz de tamaño NxN tendrá una complejidad $\mathcal{O}$ (N!), siendo N el número de hileras.

Figura 1.8. Árbol recursivo del cálculo del determinante de una matriz de 4x4



Opción B: Por el método de Gauss

Un método iterativo para calcular el determinante es el de Gauss, cuyo principio básico es el siguiente: uno puede cambiar una representación matricial A del conjunto original por una nueva, pero equivalente representación matricial según las siguientes reglas:

- Se puede multiplicar una hilera por una constante y la matriz es equivalente.
- Se podrá permutar una hilera por otra y el sistema es equivalente. Para el caso del cálculo del determinante, cada vez que se tenga una permutación existirá un cambio de signo al resultado. Por ejemplo, si existieron permutaciones impares, el resultado del cálculo del determinante se multiplicará por menos uno.

- Se puede sumar una hilera con otra y el resultado se puede ubicar en cualquiera de estas dos hileras y el sistema es equivalente.

Con estos tres criterios se puede cambiar el sistema a una forma triangular superior, de esta forma:

$$\begin{matrix} a_{00} & a_{01} & \dots & a_{0n} \\ & a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots & \dots \\ & & & a_{nn} \end{matrix}$$

Si todo valor de la diagonal principal es diferente de cero, el determinante será la multiplicatoria de la diagonal principal:

$$\text{Det(A)} = \left(\prod_{i=0}^{n} A_{ii}\right)^{*} \text{ signo de la permutación.}$$

Si no, el determinante vale cero y existe dependencia lineal. En el algoritmo 1.13 se muestra la programación del determinante por el método de Gauss.

Algoritmo 1.13. Programación del determinante por el método de Gauss

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
//PROTOTIPOS PARA EL CÁLCULO DEL DETERMINANTE
int gauss(float**, int, int*);
float **lee(float**, int*);
void imprime(float**, int);
void determinante(void);

main(){
determinante();
}

void determinante() {
//a-> matriz de la cual se calculará el determinante
//piv->pivote para realizar el metodo de gauss
// j-> bandera que indica si la matriz a tiene vectores linealmente dependientes
//k-> sirve para iterar
int i, j, k, piv = 1;
float **a;
a = lee(a,&i);
printf("El determinante a resolver es:\n");
imprime(a, i);
j = gauss(a, i, &piv);
if (j == 0) {
```

```
   printf("la matriz diagonalizada es:\n");
   imprime(a, i);
   float det = 1;
   for (k = 0; k < i; k++)
    det = det * (*(*(a + k) + k)); //producto de la diagonal
    printf("DETERMINANTE=%f\n", det * piv); //imprime el determinante
 }
else
   printf("DEPENDENCIA LINEAL");
}

int gauss(float **a, int n,int *p) {
int k, i, j,puedo = 0;
```

```
float m, M, temp;
for (k = 0; k < n - 1; k++) {//PRIMER CICLO
//busqueda del mayor abajo de la diagonal
int piv = 0;
M = fabs(*(*(a + k) + k));
for (i = k; i < n - 1; i++) {
   temp = fabs(*(*(a + i + 1) + k));
   if (M < temp) {
    *p = -1 * *p;
    piv = i + 1;
    M = temp;
   }
}

//cambio de hilera para el pivoteo parcial
if (M > fabs(*(*(a + k) + k))) {
   printf("ANTES DE PERMUTACION:\n");
   imprime(a, n);
   for (j = k; j <= n; j++) {
    temp = *(*(a + k) + j);
 *(*(a + k) + j) = *(*(a + piv) + j);
 *(*(a + piv) + j) = temp;
 }
   printf("PERMUTACION:\n");
   imprime(a, n);
}
if (*(*(a + k) + k) == 0) {
   puedo = 1;
   break;
```

```
}
//Transformando las hileras abajo del a hilera pivote
else {
   for (i = k + 1; i < n; i++) {//SEGUNDO CICLO
    m = -((*(*(a + i) + k)) / (*(*(a + k) + k)));
 for (j = k; j <=n; j++) //TERCER CICLO
 *(*(a + i) + j) = (*(*(a + i) + j)) + ((m) * (*(*(a + k) + j)));

 }
 }
}
printf("DESPUES DE LA TRANSFORMACIÓN\n");
imprime(a, n);
if (*(*(a + n - 1)+(n - 1)) == 0)
puedo = 1;
return (puedo);
}

float ** lee(float **matriz, int *num_hileras) {
int i, j, filas,k;
printf("DA EL NUMERO DE HILERAS DE LA MATRIZ:");
scanf("%d", &filas);
printf("da los elementos de la matriz A.\n");
```

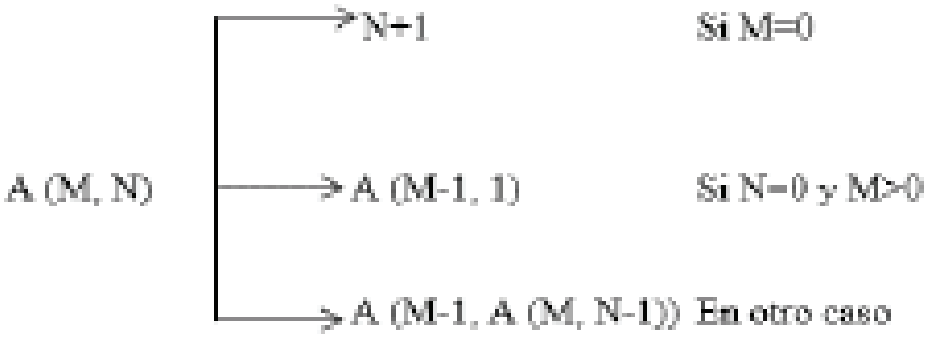
```
matriz = (float **) malloc(sizeof (float) * filas);
for (k = 0; k < filas; k++)
matriz[k] = (float *) malloc(sizeof (float) * filas);
for (i = 0; i < filas; i++) {
for (j = 0; j < filas; j++) {
printf("Dame el valor a[%d][%d]= ", i, j);
scanf("%f", (*(matriz + i) + j));
 }
 }
*num_hileras = filas;
return matriz;
}

void imprime(float **matriz, int n) {
int i, j;
for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
float temp = *(*(matriz + i) + j);
if (fabs(temp) < .000001 && j < i)
temp = 0;
 printf("%13G", temp); }
 printf("\n"); }
getchar();
```

Para el cálculo de la complejidad algorítmica se observará que la función de Gauss tiene máximo tres operadores de control for anidados, por lo que el algoritmo tiene una complejidad $O(N^3)$.

EJEMPLO 5. LA FUNCIÓN DE ACKERMANN

En teoría de la computación, la función de Ackermann es una función matemática recursiva encontrada en 1926 por Wilhelm Ackermann, la cual ha tenido un crecimiento extremadamente rápido y ha sido muy usada en las ciencias de la computación. En la actualidad, existe una serie de funciones con esta denominación, las cuales se asemejan a la ley original y tienen un comportamiento de crecimiento similar. Esta función toma dos números naturales como argumentos y devuelve un único número natural. Como norma general se define en la figura 1.9.

Figura 1.9. Algoritmo de Ackermann (Cairó y Guardati, 2000)



El algoritmo 1.14 muestra la función programada en C.

Algoritmo 1.14. Función de Ackermann

```
#include <stdio.h>
int ackermann(int, int);
main(){
int m,n;
printf("Da el valor de m=>");
scanf("%d",&m);
printf("Da el valor de n=>");
scanf("%d",&n);
printf("Ackerman(%d,%d)=%d\n",m,n,ackermann(m,n));
getchar();
}

int ackermann(int n, int x){
if (n==0)
return x+1;
else if (x==0)
return ackermann(n-1,1);
else
return ackermann(n-1,ackermann(n,x-1));
}
```

Para comprender mejor la complejidad del algoritmo se revisa la llamada recursiva A(4,x). Sabemos que

$A(0,x) = x+1$

$A(n,0) = A(n-1,1)$

$A(n, x) = A(n-1, A(n,x-1))$

Por lo que

$A(1,x)=A(0,A(1,x-1))=1+A(1,x-1))=1+A(0,A(1,x-2))=1+1+A(1,x-2)=1+1+1+A(0,A(1,x-3))$

$A(1,x)=1+1+1+\ldots$ (x-1 veces uno) $+ A(0,A(1,0))=(x-1)+1+A(1,0)=x+A(0,1)=x+1+1=x+2$

Apoyándose en el valor de A(1,x) se obtiene:

$A(2,x)=A(1,A(2,x-1))=2+A(2,x-1)=2+2+A(2,x-2)=2+2+2+A(2,x-3)=2x+A(1,1) = 2x+3$

Partiendo de estos dos valores se tratará de obtener A(3,x) y recordando que

$A(n, x)=A(n-1, A(n,x-1))$, se tiene:

$A(3,x) = A(2, A(3, x-1)) = 3+2^a(3,X-1) = 3+2^a(2, A(3, x-2)) =3+2(3+2^a(3,X-2))$

$= 3+2(3+2(A(2,A(3,x-3)))$

$= 3+2(3+2(3+2(A(2,A(3,x-4))))$

= 3+2(3+2(3+2(3+...2,A(3,0)))…) //A(3,0)=A(2,1)=2(1)+3=5

= 3+2(3+2(3+2(3+...2(5)))))…)

Se observa que se tiene un anidamiento con una profundidad x (desde x-1 hasta 0).

Abriendo un paréntesis, primero entenderemos la siguiente operación algebraica equivalente a A(3,x) para comprender el comportamiento de la anidación:

$x(y+x(y+x(y+xz))) = x(y+x(y+xy+x^2 z)) = x(y+xy+ x^2 y+ x^3 z) = xy+ x^2 y+ x^3 y+ x^4 z$

Por lo tanto,

$A(3,x) = 3+3*2+3*2^2+3*2^3+3*2^4+...+ 5*2^x$

$= 3(2^0+2^1+2^2+2^3+...+2^{x-1}) +5*2^x$

$= 3(2^x -1) + 5*2^x$

$= 3*2^x -3+5*2^x$

$= 8*2^x-3$

$= 2^3*2^x-3$

$= 2^{x+3} - 3$

El valor de A(3,x) se utilizará para obtener

A(4,x) =A(3, A(4,X-1))

$$=2^{A(4,x-1)+3} -3$$

$$=2^{2^{A(4,A(x-2))+3}-3}-3$$

obteniéndose como resultado x potencias sucesivas de 2.

Algunos valores interesantes de la función de Ackermann son:

A(0,0) = 1 A(0,n) = n+1;

A(1,0) = 2

A(1,n) = n+1

A(2,0) = 3

A(2,n) = 2n+3

A(3,0) = 5

A(3,1) = 13

A(3,2) = 29

A(3,3) = 61

A(3,4) = 125

$A(3,n) = 8 \cdot 2^n - 3$

$A(4,0) = 13$

$A(4,1) = 65533$

$A(4,2) = 0\,2^{65536} - 3 \approx 2 \cdot 10^{19728}$

$A(4,3) = A(3,2^{65536}\text{-}3)$

$A(4,4) = A(3,A(4,3))$

$A(4,n) = 2^{2^{\cdot^{\cdot^{2}}}} - 3\,(n+3 \text{ términos})$

$A(5,0) = 65533$

$A(5,1) = A(4,65533)$

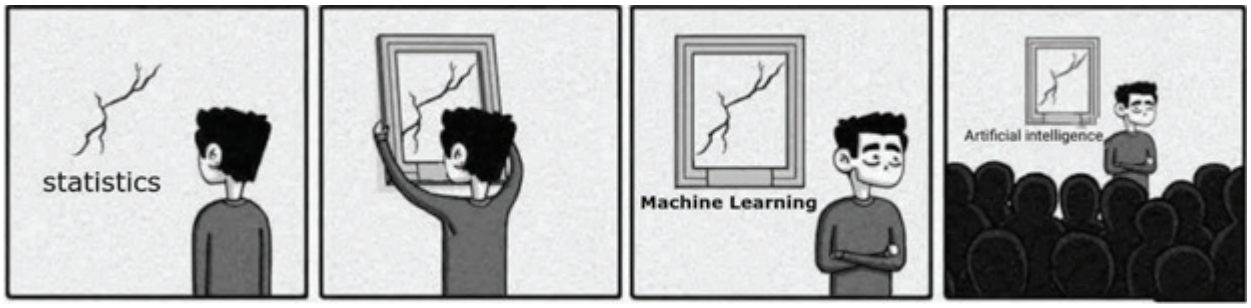
$A(5,2) = A(4,A(5,1))$

etc.

Para darse una idea de la magnitud de los valores que aparecen, se puede destacar que, por ejemplo, A(4, 2) es mayor que el número de partículas que forman el universo elevado a la potencia 200 y el resultado de A(5, 2) no se puede escribir dado que no cabría en el universo físico.

SECCIÓN II

Paradigmas de solución de problemas



INTRODUCCIÓN

Un problema se define como un asunto o una cuestión que requiere solución a nivel social. Se trata de alguna situación en concreto que, en el momento en que se logra resolver, aporta beneficios a la sociedad. Esa solución se puede apoyar en las siguientes estrategias:

- Criterio de error y acierto. Esta se puede apoyar en la heurística como procedimiento práctico o informal para resolver problemas (existen otras definiciones para la misma palabra). Además, permite resolver problemas viendo cómo otros lo hacen.
- Matemáticamente. Consiste en buscar una determinada entidad matemática de entre un conjunto de entidades del mismo tipo para satisfacer las llamadas *condiciones del problema.*
- Una solución algorítmica a un problema abstracto. Consiste de un algoritmo que por cada instancia del problema calcula al menos una solución correspondiente —en caso de haberla— o expide un certificado de que no existe solución alguna al algoritmo dado (el algoritmo se basa en una solución matemática o una solución heurística)

Ejemplo de un tipo de problema que no se puede resolver por medio de un algoritmo, aunque sí tuvo solución matemática, es el último teorema de Fermat, el cual indica que $x^n + y^n \neq z^n$ para $n > 2$. Fue conjeturado por Pierre de Fermat en 1637 y demostrado hasta 1995 por el matemático Andrew Wiles. En otras palabras, existirán problemas que tienen una solución matemática pura, aunque sin lograr presentar un algoritmo.

Si un problema no se puede resolver de forma matemática, es posible introducir una heurística para, por medio de la experiencia, intentar llegar a un espacio que se acerque a la solución. Apoyándose en la heurística, es posible formular un algoritmo que permita, en forma más sencilla, manejar la expertiz del individuo y conseguir un espacio cercano a la solución.

La palabra *heurística* proviene del griego *euriskein*, que significa 'encontrar'. Este vocablo se volvió muy popular gracias al libro Cómo resolverlo *(How to solve it)* de George Pólya publicado por la Universidad de Princeton en 1945, en el cual se proponen muchas recetas para tratar de encontrar solución a problemas complejos.

En optimización, la palabra *heurística* se utiliza para caracterizar a las técnicas por las cuales se encuentra o se mejora la solución de un problema intratable. Los algoritmos heurísticos permiten obtener "buenas soluciones" (soluciones subóptimas o cercanas al óptimo global) de problemas cuyos algoritmos exactos de solución no son factibles en tiempo polinomial.

En computación es fundamental encontrar algoritmos con buen tiempo de ejecución y que proporcionen "buenas soluciones"; una heurística, por tanto, es un algoritmo que trata de satisfacer al menos uno de estos requerimientos, cuando no ambos. Muchos algoritmos de inteligencia artificial son heurísticos por naturaleza o usan reglas heurísticas (De los Cobos Silva, Goddard, Gutiérrez Andrade y Martínez Licona, 2010).

Un ejemplo de esto se puede observar en el problema del agente viajero, del cual se explica en el apartado 4.2 una solución matemática exacta por medio de la estrategia "programación dinámica" (este problema es de naturaleza intratable), mientras que en el apartado 7.4 se muestra un algoritmo heurístico para resolver el mismo problema por medio de la estrategia de ramificación y acotamiento. Los problemas computacionales se pueden clasificar de forma general del siguiente modo:

1. Problemas de ordenación.
2. Problemas de búsqueda.
3. Problemas de optimización.
4. Problemas de decisión.
5. Problemas de clasificación.

En esta sección se estudiarán las estrategias clásicas para la resolución de la mayoría de los problemas en forma determinística, donde **se encuentra la solución al problema dado**. La mayoría de los problemas clasificados de la categoría uno a la cuatro se puede resolver con alguna de estas estrategias.

Existen problemas que tienen una solución en las estrategias conocidas como *máquinas de aprendizaje (heurísticas) o machine learning* y generalmente sirven para encontrar un espacio factible cerca-

no a la solución de un problema que no se puede resolver por medio de un método determinístico. Las estrategias determinísticas que se estudian a continuación son:

- Divide y conquistarás.
- La estrategia del avaro.
- Programación dinámica.
- Programación lineal.
- Retorno atrás.
- Ramificación y acotamiento.

Las estrategias que engloban las máquinas de aprendizaje son:

- Algoritmos genéticos y evolutivos.
- Redes neuronales artificiales.
- Regresión.
- Máquinas de soporte vectorial.
- Redes bayesianas.

Divide y conquistarás (divide and conquer DANDC).

INTRODUCCIÓN

Dada una función para computar n entradas, la estrategia divide y conquistarás sugiere dividir la entrada en k subconjuntos, $1 \leq k \leq n$, lo que genera *k* subproblemas, los cuales deben ser resueltos mediante un método que debe ser hallado para combinar las subsoluciones en una solución general. Si el subproblema sigue siendo demasiado grande, entonces puede ser reaplicada la estrategia. Frecuentemente, los subproblemas resultantes de un diseño divide y conquistarás son del mismo tipo que el problema original. Este principio se expresa de forma natural por un procedimiento recursivo, de ahí que se obtengan subproblemas de menor dimensión aunque de la misma clase; de hecho, eventualmente se producen subproblemas que son lo suficientemente pequeños como para ser resueltos sin la necesidad de dividirlos. Una forma general de ver este modelo se observa en el algoritmo 2.1.

Algoritmo 2.1. Forma general de la estrategia divide y conquistarás (Horowitz y Sahni, 1978)

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

En este caso, la función *pequeño* determina si se cumple una condición específica para que el algoritmo se detenga; si no es así, se crean dos subespacios en los cuales se subdivide el espacio en dos más pequeños. DANDC se puede describir por esta relación recurrente:

$$T(n)= \begin{cases} g(n) & \text{para n pequeño} \\ 2T(n/2) + f(n) & \text{de otra forma} \end{cases}$$

donde *f(n)* es el tiempo que se dedica para calcular las funciones divide y conquistarás. Dos algoritmos de ordenamiento y dos de búsqueda pertenecen a este paradigma:

- Búsqueda binaria *(binary search).*
- Búsqueda del máximo y mínimo.
- Mezcla *(merge sort).*
- Ordenación rápida *(quick sort).*

Antes de analizar estos algoritmos se dará una introducción a los más conocidos que no pertenecen a divide y conquistarás; estos son:

- Burbuja.
- Ordenación por selección directa.
- Inserción binaria.

MÉTODO DE LA BURBUJA

Es el algoritmo de ordenamiento más sencillo, ideal para empezar; consiste en ciclar repetidamente a través de una lista para comparar elementos adyacentes de dos en dos. Si un elemento es mayor al que está en la siguiente posición, se intercambian. El algoritmo 2.2 muestra en lenguaje de programación C:

Algoritmo 2.2. Método de la burbuja

```
#include <stdio.h>
#define tam      10
main(){
int lista[]={5,4,3,2,10,9,8,7,6,1};
for (int i=0;i<tam;i++)
for (int j=0;j<tam-1;j++)
if(lista[j]>lista[j+1]){
int temp;
temp=lista[j];
lista[j]=lista[j+1];
lista[j+1]=temp;
}
for (int i=0; i<10;i++)
printf("%d..",lista[i]);
getchar();
}
```

donde

lista: Es cualquier lista para ordenar.

TAM: Es una constante que determina el tamaño de la lista.

i, j: Contadores.

temp: Variable que permite realizar los intercambios de la lista.

Veamos un ejemplo. Esta es nuestra lista:

4 - 3 - 5 - 2 – 1.

Tenemos cinco elementos; por tanto, TAM toma el valor 5. Comenzamos comparando el primer elemento con el segundo: 4 es mayor que 3, así que intercambiamos. Ahora tenemos:

3 - 4 - 5 - 2 – 1.

Luego comparamos el segundo con el tercero: 4 es menor que 5, así que no hacemos nada. Seguimos con el tercero y el cuarto: 5 es mayor que 2, por tanto, intercambiamos y obtenemos:

3 - 4 - **2 - 5** – 1.

Comparamos el cuarto y quinto elemento: 5 es mayor que 1, así que intercambiamos nuevamente:

3 - 4 - 2 - **1 – 5.**

Repitiendo este proceso, vamos obteniendo los siguientes resultados:

3 - 2 - 1 - 4 - 5

2 - 1 - 3 - 4 - 5

1 - 2 - 3 - 4 - 5

Este es el análisis para la versión no optimizada del algoritmo:

- Estabilidad: Este algoritmo nunca intercambia *registros* con *claves iguales*. Por lo tanto, es *estable*.
- Requerimientos de memoria: Este algoritmo solo requiere de una variable adicional para realizar los intercambios.
- Tiempo de ejecución: El ciclo interno se ejecuta n veces para una lista de *n* elementos. El ciclo externo también se ejecuta *n* veces. Es decir, la complejidad

es $\mathcal{O}(n^2)$. El comportamiento del caso promedio depende del orden de entrada de los datos, pero es solo un poco mejor que el del peor caso, y sigue siendo $\mathcal{O}(n^2)$.

Ventajas

- Fácil implementación.
- No requiere memoria adicional.

Desventajas:

- Muy lento.
- Realiza numerosas comparaciones.
- Realiza numerosos intercambios.

ORDENACIÓN POR SELECCIÓN DIRECTA

La idea básica de este algoritmo consiste en buscar el menor elemento del arreglo y colocarlo en la primera posición. Luego se busca el segundo elemento más pequeño del arreglo y se coloca en la segunda posición. El proceso continúa hasta que todos los elementos del arreglo hayan sido ordenados. El método se basa en los siguientes principios:

- Seleccionar el menor elemento del arreglo.
- Intercambiar dicho elemento con el primero.
- Repetir los pasos anteriores con los *(n-1), (n-2)* elementos, y así sucesivamente hasta que solo quede el elemento mayor.

El algoritmo 2.3 muestra el código en C:

Algoritmo 2.3. Ordenación por selección directa

```
#include <stdio.h>
main(){
int MENOR, i, a[7]={10,7,6,4,9,8,1},j , k , m;
for (i=0;i<6;i++){
MENOR = a[i];
for (j=i+1; j<7;j++){
if (a[j] <MENOR){
MENOR=a[j];
k=j;
          }
   }
if (MENOR<a[i]){
a[k] =a[i];
a[i] =MENOR;
   }
}
for (i=0; i<7;i++)
printf ("%d...", a[i]);
getchar ();
}
```

El análisis del método de selección directa es relativamente simple. Se debe considerar que el número de comparaciones entre elementos es independiente de la disposición inicial de estos en el arreglo. En la primera pasada se realizan (n-1) comparaciones, en la segunda pasada (n-2) comparaciones y así sucesivamente hasta 2 y 1 comparaciones en la penúltima y última pasadas, respectivamente. Por esto:

$$C = (n-1) + (n-2) + \ldots + 2 + 1.$$

Ahora bien, utilizando el truco de Gauss para la suma de números naturales se tiene:

$$\begin{aligned} S_n &= 1 + 2 + 3 + 4 + \ldots + (n-3) + (n-2) + (n-1) + n \\ S_n &= n + (n-1) + (n-2) + (n-3) + \ldots + 4 + 3 + 2 + 1 \\ \hline 2S_n &= (n+1) + (n+1) + (n+1) + (n+1) + \ldots + (n+1) + (n+1) + (n+1) + (n+1) \end{aligned}$$

$2S_n = n^* (n+1)$, por lo tanto, $S_n = n^* (n+1) / 2$. Utilizando la misma idea, se tiene:

$$S_n = 1 + 2 + 3 + \ldots + (n-3) + (n-2) + (n-1)$$
$$S_n = (n-1) + (n-2) + (n-3) + (n-4) + \ldots + 2 + 1$$
$$2S_n = (n-1) + (n-1) + (n-1) + (n-1) + \ldots + (n-1) + (n-1) + (n-1)$$

$2S_n = n^* (n-1)$, por lo tanto, $S_n = n^* (n-1) / 2$. De esta forma se tiene:

$$C = n^* (n-1) / 2 = (n^2-n) / 2.$$

Entonces, el tiempo de ejecución del algoritmo es proporcional a $\mathcal{O}(n^2)$.

MÉTODO DE INSERCIÓN BINARIA

Este realiza una búsqueda binara, en lugar de una búsqueda secuencial, para insertar un elemento en la parte izquierda del arreglo, que ya se encuentra ordenado. El proceso se repite desde el segundo hasta el n-ésimo elemento (algoritmo 2.4).

Algoritmo 2.4. Método de inserción binaria

```
#include <stdio.h>
main(){
int a[]={10,8,7,2,1,3,5,4,6,9},I,aux,der,izq,m,j;
for (i=1;i<10;i++){
aux=a[i];
izq=0;
der=i-1;
while(izq<=der){
m=(izq+der)/2;
if (aux<=a[m])
der=m-1;
else
izq=m+1;
   }
j=i-1;
while(j>=izq){
a[j+1]=a[j];
j--;
   }
a[izq]=aux;
}
for (i=0;i<10;i++)
printf("%d..",a[i]);
putchar('\n');
getchar();
}
```

Al analizar el método de ordenación por inserción binaria se advierte la presencia de un caso antinatural. El método efectúa el menor número de comparaciones cuando el arreglo está totalmente desordenado, y el máximo cuando se encuentra ordenado.

Es posible suponer que mientras en una búsqueda secuencial se necesitan K comparaciones para insertar un elemento, en una binaria se necesita la mitad de ellas. Por lo tanto, el número de comparaciones promedio en este método se puede calcular así:

$$C = 1/2+2/2+3/2+\ldots+(n-1) / 2 = (n*(n-1)) / 4 = (n^2-n) / 4.$$

Por lo tanto, el tiempo de ejecución del algoritmo sigue siendo proporcional a $O(n^2)$.

MÉTODO DE MEZCLA (MERGE SORT)

Este algoritmo ordena elementos y tiene la propiedad de que el peor caso en complejidad será O (n log n). Los elementos van a ser ordenados de forma creciente. Dados n elementos, estos se dividirán en dos subconjuntos. Cada subconjunto será ordenado y el resultado será unido para producir una secuencia de elementos ordenados. El algoritmo 2.5 muestra el código en C:

Algoritmo 2.5. Método de mezcla

```
#include <stdio.h>
#define N 10
void mergesort(int [],int,int);
void merge(int [],int,int,int);
main(){
int i,a[N]={9,7,10,8,2,4,6,5,1,3};
mergesort(a,0,9);
for (i=0;i<10;i++)
printf("%d..",a[i]);
getchar();
 }

void mergesort(int a[],int low, int high){
int mid;
if (low<high){
mid=(low+high)/2;
mergesort(a,low,mid);
mergesort(a,mid+1,high);
merge(a,low,mid,high);
 }
}

void merge(int a[],int low, int mid, int high){
int b[N],h,i,j,k;
h=low;
i=low;
j=mid+1;
while(h<=mid && j<=high){
if (a[h]<=a[j]){ b[i]=a[h];
h++; }
else{b[i]=a[j]; j++; }
```

```
i++;
 }
if (h>mid)
   for (k=j;k<=high;k++){
   b[i]=a[k];
 i++;}
else
   for (k=h;k<=mid;k++){
   b[i]=a[k];
   i++;}
for (k=low;k<=high;k++) a[k]=b[k];
}
```

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

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

Las barras verticales indican el acotamiento de los subarreglos. A(1) y A(2) son fusionados, lo que produce:

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

Entonces, A(3) es fusionado con A(1:2), lo que genera:

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

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

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

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

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

En este punto, el algoritmo ha retornado a la primera invocación de *merge sort* y se realizará la segunda llamada recursiva. Las llamadas recursivas a la derecha producen los siguientes subarreglos:

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

A(6) y A(7) son fusionados y entonces A(8) se fusiona con A(6:7), lo que origina:

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

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

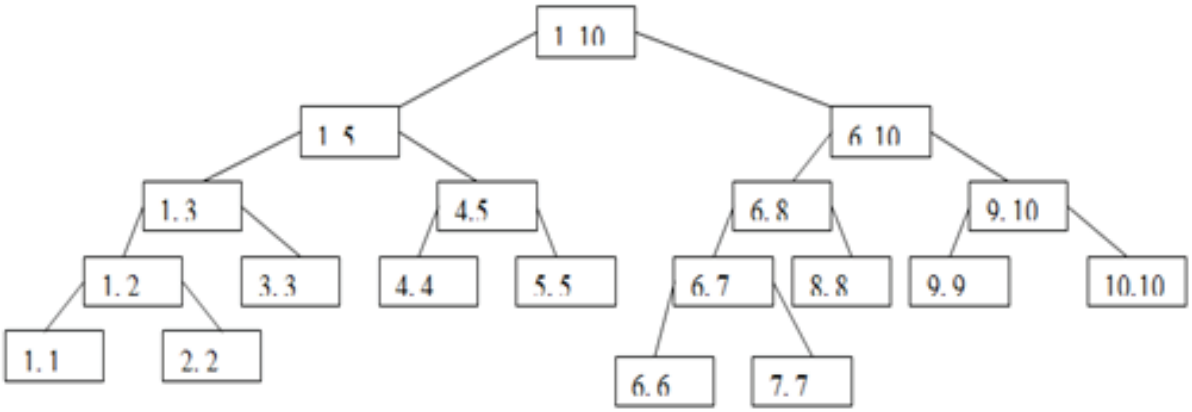
(179, 285, 310, 351, 652|254, 423, 450, 520, 861).

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

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

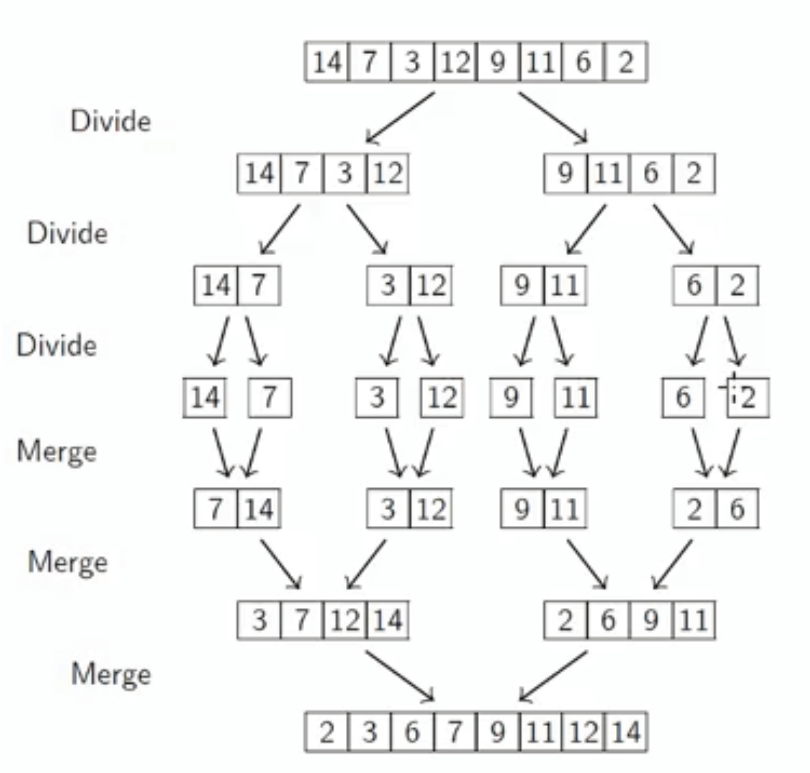
A continuación, la figura 2.1 enseña la secuencia de recursividades producidas por *merge sort* con los diez elementos, así como la forma en que se van a dividir los subarreglos. Véase que la división continúa hasta contener un simple elemento.

Figura 2.1. Árbol recursivo para el algoritmo *merge sort*



Ahora, en la figura 2.2 se aprecia la parte de división de un arreglo, la parte de ordenación de cada uno de los subarreglos, así como sus respectivas fusiones hasta conseguir el arreglo ordenado.

Figura 2.2. Método *merge sort* (University of Pennsylvania. Departament of Engineering)



El tiempo de cómputo para *merge sort* se describe a continuación:

$$T(n)= \begin{cases} a & n=1 \quad \text{a es una constante.} \\ 2(T(n/2) + C_n & n>1 \quad \text{c es una constante.} \end{cases}$$

La recursividad para *merge sort* requiere:

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

ya que

- Realizar la recursividad de los dos subarreglos requiere *2T(n/2).*
- La ordenación y fusión de los subarreglos requiere *2n* $\in$ *O(n).* El costo es C_n.
- La base de la recursión es *T(1) = 0*, ya que es trivial ordenar un arreglo de un elemento.

Si n es una potencia de 2, entonces $n = 2^k$. Así, resolviendo por sustituciones sucesivas se tiene:

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

$T(n) = 2(2T(n/4) + C_{n/2}) + C_n$

$T(n) = 2(2(2T(n/8) + C_{n/4}) + C_{n/2}) + C_n$

$T(n) = 8T(n/8) + 4C_{n/4} + 2C_{n/2} + C_n$

$T(n)=2^kT(n/2^k)+\underbrace{2^{k-1}C_{n/X}\ldots+ 2^2C_{n/4} +2^1C_{n/2} + 2^0C_n}_{\text{K elementos}}$ //Donde $X=2^{k-1}$

Observando que $4C_{n/4} \sim C_n$, y $T(n/2^k) = 0$, se tiene:

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

$T(n) = K\ C_n$.

Conociendo que $C_n \in O(n)$,

$T(n)=K\ O(n)$.

Observando que $n = 2^k$, $K= \log_2 n$, por lo tanto,

$T(n) = \mathcal{O}(n \log n)$.

MÉTODO DE ORDENACIÓN RÁPIDA (QUICK SORT)

Este es actualmente el más eficiente y veloz de los métodos de ordenación interna. Fue llamado de esta forma por su autor —C. A. Hoare—, y consiste en lo consiste:

1. Se toma un elemento X de una posición cualquiera del arreglo.
2. Se trata de ubicar a X en la posición correcta del arreglo, de modo que todos los elementos que se encuentran a la izquierda sean menores o iguales a X, y todos los que se hallan a la derecha sean mayores o iguales a X.
3. Se repiten los pasos anteriores, pero ahora para los conjuntos de datos que se encuentran a la izquierda y a la derecha de la posición X en el arreglo.
4. El proceso termina cuando todos los elementos se hallan en su posición correcta en el arreglo.

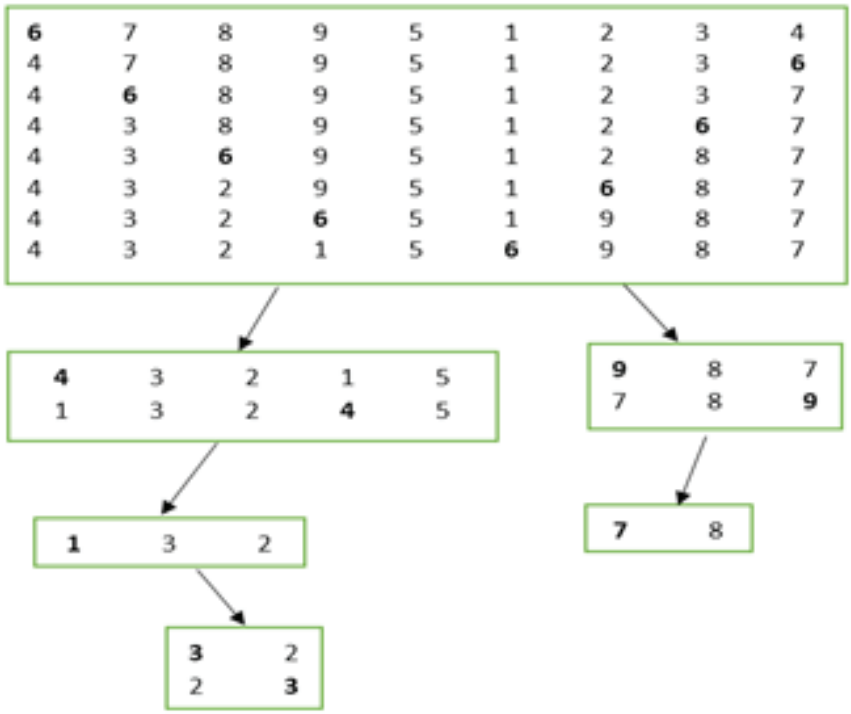
En este caso, para la programación del algoritmo, el elemento X será el primer elemento de la lista. Se empieza a recorrer el arreglo de derecha a izquierda comparando si los elementos son mayores o iguales a X. Si un elemento no cumple con esta condición, se intercambian aquellos y se almacena en una variable la posición del elemento intercambiado —se acota el arreglo por la derecha—. Se inicia nuevamente el recorrido, pero ahora de izquierda a derecha, comparando si los elementos son menores o iguales a X. Si un elemento no cumple con esta condición, entonces se intercambian aquellos y se almacena en otra variable la posición del elemento intercambiado —se acota el arreglo por la izquierda—. Se repiten los pasos anteriores hasta que el elemento X encuentra su posición correcta en el arreglo. En este momento, dependiendo del lugar que ocupe el valor de X, se hará una recursividad izquierda tomando solo los primeros elementos del subarreglo hasta el vecino a la izquierda de X o una recursividad a la derecha del vecino más cercano a la derecha de X hasta el final del subarreglo. A continuación, se muestra el algoritmo 2.6 en lenguaje de programación C.

Algoritmo 2.6. *Método quick sort*

```
#include <stdio.h>
#define N 10
void quicksort(int [], int, int);
main(){
int a[]={10,8,7,2,1,3,5,4,6,9},i;
quicksort(a,0,N-1);
for (i=0;i<10;i++)
printf("%d..",a[i]);
putchar('\n');
getchar();
}

void quicksort(int a[],int ini,int fin){
int izq=ini, der=fin, pos=ini,band=1,aux;
while(band==1){
band=0;
while(a[pos]<=a[der] && pos!=der) der--;
if (pos!=der){
aux=a[pos];
a[pos]=a[der];
a[der]=aux;
pos=der;
while(a[pos]>=a[izq] && pos!=izq)
izq++;
if (pos!=izq){
band=1;
aux=a[pos];
a[pos]=a[izq];
a[izq]=aux;
pos=izq;
          }
   }
}
if (pos-1>ini)
quicksort(a, ini, pos-1);
if (fin>pos+1)
quicksort(a,pos+1,fin);
}
```

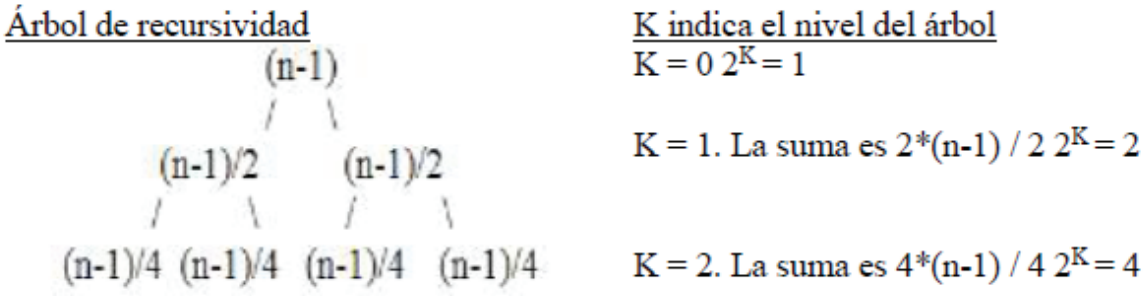
Para ejemplificar, en la figura 2.3, dentro de cada cuadro, se muestra la parte iterativa y cada llamada recursiva crea un nuevo cuadro. Los números con negrita son los que se utilizan como pivote para que sean los que se van a acomodar de tal forma que a su izquierda se coloquen los números menores a él y a la derecha los números mayores a él.

Figura 2.3. Árbol recursivo utilizando quick sort



El *método quick sort* es el más rápido de ordenación interna que existe en la actualidad, lo cual es sorprendente porque este tiene su origen en el método de intercambio directo, el peor de todos los directos.

Diversos estudios realizados sobre su comportamiento demuestran que si se escoge en cada pasada el elemento que ocupa la posición central del conjunto de datos a analizar, y si el tamaño del arreglo es **una potencia de 2**, en la primera pasada realiza *(n-1)* comparaciones; en la segunda *(n-1)/2* comparaciones, pero en dos conjuntos diferentes; en la tercera *(n-1)/4* comparaciones, pero en cuatro conjuntos diferentes, y así sucesivamente. Esto produce un árbol binario recursivo. Por lo tanto,



C = (n-1) + 2*(n-1) / 2 + 4*(n-1) / 4+... (n-1) *(n-1) / (n-1),

lo cual es lo mismo que

C=(n-1) +(n-1) +…+(n-1).

Si se considera a cada uno de los componentes de la sumatoria como un término y el número de términos de la sumatoria es igual a k (siendo k la profundidad del árbol binario recursivo), se tiene:

C = (n-1) *k.

Considerando que el número de términos de la sumatoria (k) es el número de niveles del árbol binario, el número de elementos del arreglo se puede definir como $2^k = n$, por esto,

$\text{Log}_2 2^k = \log_2 n$, $k \log_2 2 = \log_2 n$ (recordando que $\log_m m=1$), $k = \log_2 n$,

de ahí que la expresión anterior quede así:

$C = (n-1) * \log_2 n$

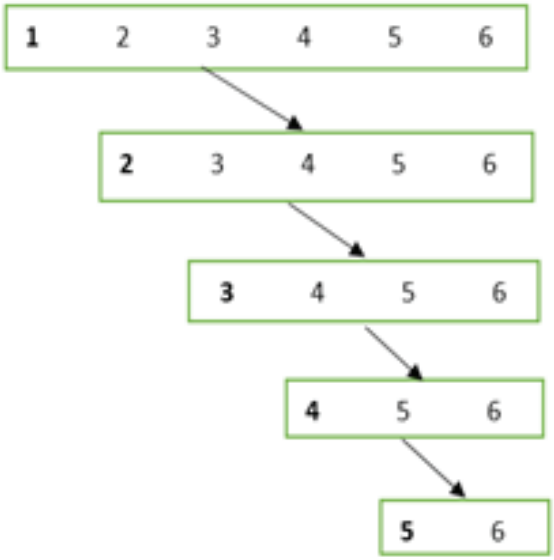
Otra forma de verlo es esta:

Cuando se realiza la llamada recursiva, una de las llamadas tomará el valor de k y la otra el valor de n – k -1, por lo tanto:

T(n) = T(K) + T(n-k-1) + O (n).

En el **peor caso** se tiene cuando el arreglo está ordenado, por lo que k = 1 y la función llamada será (n-1), (n-2)…2 (figura 2.4).

Figura 2.4. Peor caso para *quick sort*



Como se observa en la figura 2.4, no se tiene recursividad izquierda, y para la recursividad derecha cada llamada se realiza con un elemento menos:

$C_{máx}$ = n+(n-1) + (n-2) + (n-3) +…+2 ≃n*(n-1) /2 (observar el método de inserción binaria) o:

T(n) = T (1) + T(n-1) + n

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

T(n) = T (1) + T (1) + T (1) + T(n-3) + n + (n-1) + (n-2)

…

T(n) = T (1) + T (1) + T (1) + … T(1) + T(2) + n + (n-1) + (n-2) + … + 3

$$T(n)= \underbrace{T(1) + T(1) + T(1) + \dots T(1) + T(1)}_{\text{n-1 veces}} + n + (n-1) + (n-2) + \dots + 3 + 2$$

T(n)= n + (n-1) + (n-2) +… + 3 +2.

Utilizando una adaptación del truco de Gauss, se obtiene:

T(n) = $\mathcal{O}$ (n^2).

En el mejor caso, cuando existe una buena partición, se tiene:

K = n / 2

T(n) = 2T(n/2) + $\mathcal{O}$ (n)

T(n) = $\mathcal{O}$ (nLog(n)),

siendo el análisis análogo al *merge sort*.

Sin embargo, encontrar el elemento que ocupe la posición central del conjunto de datos que se va a analizar es una tarea difícil, ya que existen 1/n posibilidades de lograrlo. Además, el rendimiento medio del método es aproximadamente (2*ln 2) inferior al caso óptimo, por lo que Hoare, el autor del método, propone como solución que el elemento X se seleccione arbitrariamente o bien entre una muestra relativamente pequeña de elemento del arreglo.

Se puede afirmar que el tiempo promedio de ejecución del algoritmo es proporcional a $\mathcal{O}$ (n* log n). En el peor de los casos, es proporcional a $\mathcal{O}$ (n^2).

BÚSQUEDA SECUENCIAL

La búsqueda secuencial consiste en revisar elemento tras elemento hasta encontrar el dato indagado o hasta llegar al final del conjunto de datos disponible. Normalmente cuando una función de búsqueda concluye con éxito, interesa conocer en qué posición fue hallado el elemento que se estaba buscando. Esta idea se puede generalizar para todos los métodos de búsqueda.

A continuación, en el algoritmo 2.7 se presenta el método de búsqueda secuencial en arreglos desordenados en código C:

Algoritmo 2.7. Método de búsqueda secuencial

```
#include <stdio.h>
#define N 10
main(){
int x,i=0,a[N]={9,7,10,8,2,4,6,5,1,3};
scanf("%d",&x);
while (i<N && a[i]!=x)
i++;
if (i>N-1)
printf("dato no encontrado");
else
printf("El dato se encuentra en la posici[on %d\n",i);
getchar();
getchar(); }
```

Si hubiera dos o más ocurrencias del mismo valor, se encuentra la primera de ellas. Sin embargo, es posible modificar el algoritmo para obtener todas las ocurrencias de datos buscados.

En el algoritmo 2.8 se presenta una variante de este algoritmo, pero utilizando recursividad en lugar de interactividad.

Algoritmo 2.8. Método de búsqueda secuencial recursivo

```
#include <stdio.h>
#define N 10
void secuencial(int [], int,int,int);

main(){ int x,i=0,a[N]={9,7,10,8,2,4,6,5,1,3};
scanf("%d",&x);
secuencial(a,N,x,0);
getchar();getchar(); }

void secuencial(int a[],int n, int x, 64nti ){
if (i>n-1) printf("Dato no localizado\n");
else if (a[i]==x) printf("dato localizado en la posicion %d\n",i);
else secuencial(a,n,x,i+1); }
```

El número de comparaciones es uno de los factores más importantes para determinar la complejidad de los métodos de búsqueda secuencial; en este se deben establecer los casos más favorables o desfavorables que se presenten.

Al buscar un elemento en el arreglo unidimensional desordenado de N componentes, puede suceder que ese valor no se encuentre; por lo tanto, se harán N comparaciones al recorrer el arreglo. Por otra parte, si el elemento se encuentra en el arreglo, este puede estar en la primera posición, en la última o en alguna intermedia. Así,

$C_{min} = 1$ $C_{med} = (n+1)/2$ $C_{máx} = n$.

El algoritmo tiene un $\mathcal{O}(n)$.

BÚSQUEDA BINARIA (BINARY SEARCH)

El método de búsqueda binaria funciona exclusivamente con arreglos ordenados. No se puede utilizar con listas simplemente ligadas ni con arreglos desordenados. Con cada iteración del método el espacio de búsqueda se reduce a la mitad; por lo tanto, el número de comparaciones que se deben realizar disminuye notablemente, lo cual resulta favorable cuando es más grande el tamaño del arreglo.

La búsqueda binaria consiste en dividir el intervalo de búsqueda en dos partes, comparando el elemento buscado con el que ocupa la posición central en el arreglo. Cuando estos no son iguales, se redefinen los extremos del intervalo, tomando en cuenta si el elemento central es mayor o menor que el buscado para disminuir el espacio de búsqueda. El proceso concluye cuando el elemento es encontrado o cuando el intervalo de búsqueda se anula, es vacío. El algoritmo 2.9 muestra el método de búsqueda binaria:

Algoritmo 2.9. Método de búsqueda binaria

```
#include <stdio.h>
#define N 10
main(){
int x,low,high,mid,j,n,a[]={1,2,3,5,6,7,8,9,10,13};
low=j=0;
high=N-1;
scanf("%d",&x);
while(low<=high){
   mid=(low+high)/2;
   if (x<a[mid])
   high=mid-1;
   else if (x>a[mid])
   low=mid+1;
   else{
   j=mid;
   break; }
 }
if (j==0) printf("Elemento no encontrado");
else printf("Elemento localizado en el lugar %d.\n",j);
getchar(); getchar();
}
```

Como ejemplo se enseña este arreglo con los siguientes números:

A(n)	15	-6	0	7	9	23	54	82	101
Comparaciones	3	2	3	4	1	3	2	3	4

En el peor de los casos se requieren 4 comparaciones.

El promedio de las comparaciones es 2.77.

El óptimo es una comparación.

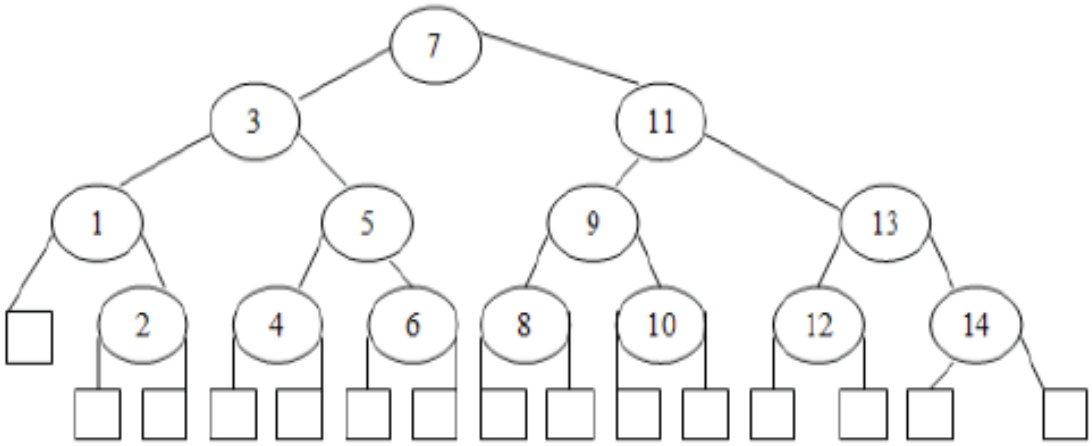
Si no se encuentra un elemento:

A(n)	15	-6	0	7	9	23	54	82	101
Comparaciones 3	3	3	4	4	3	3	3	4	4

El promedio es 3.4.

Este análisis trabaja para nueve elementos, pero ¿y para cualquier n? Por ejemplo, para n = 14 (figura 2.5) se tiene:

Figura 2.5. Algoritmo de búsqueda binaria con 14 elementos



Teorema. Si n está en el rango $[2^{k-1}, 2^k]$, el algoritmo de búsqueda binaria hace máximo k comparaciones, por lo que requiere máximo $\mathcal{O}$ (log n) para un éxito y para un fracaso Θ (log n).

BUSCANDO EL MÁXIMO Y MÍNIMO (FINDING THE MAXIMUN AND MINIMUN)

El problema es encontrar el máximo y el mínimo de un conjunto de n elementos en desorden.

El algoritmo 2.10 muestra un método directo:

Algoritmo 2.10. Método directo para el máximo y mínimo

```
#include <stdio.h>
main(){
int max,min,i,j,a[]={4,2,1,5,7,9,8,6,3,10};
max=min=a[0];
for(i=1;i<10;i++){
if (a[i]>max) max=a[i];
if (a[i]<min) min=a[i];
 }
printf("El maximo valor es %d y el minimo valor es %d\n",max,min);
getchar(); }
```

El procedimiento requiere 2(n-1) comparaciones en el mejor, en el promedio, así como en el peor de los casos. Puede existir una mejora al cambiar el ciclo de la siguiente forma:

if (a[i]>max)

max=a[i];

else if (a[i]<min)

min=a[i];

Ahora, el mejor caso surge cuando los elementos están en forma creciente, ya que en el mejor de los casos se requieren n-1 comparaciones, mientras que en el peor de los casos se necesitan 2(n-1) comparaciones. El promedio será:

$$[2(n-1) + n-1] / 2 = (3n-1) / 2 - 1.$$

A continuación, el algoritmo 2.11 muestra una forma recursiva para encontrar el máximo y el mínimo de un conjunto de elementos y maneja la estrategia de divide y conquistarás. Este algoritmo envía cinco parámetros: el primero es el arreglo para trabajar; los otros dos se manejan como paso de parámetros por valor, y los dos últimos se manejan como pase de parámetros por referencia. En este caso, el segundo y tercer parámetro indican el índice del subconjunto a analizar, y los dos últimos parámetros sirven para retornar el mínimo y máximo de un subconjunto determinado. Al término de la recursión se obtienen el mínimo y máximo del conjunto dado.

Algoritmo 2.11. Encontrando el máximo y mínimo en forma recursiva

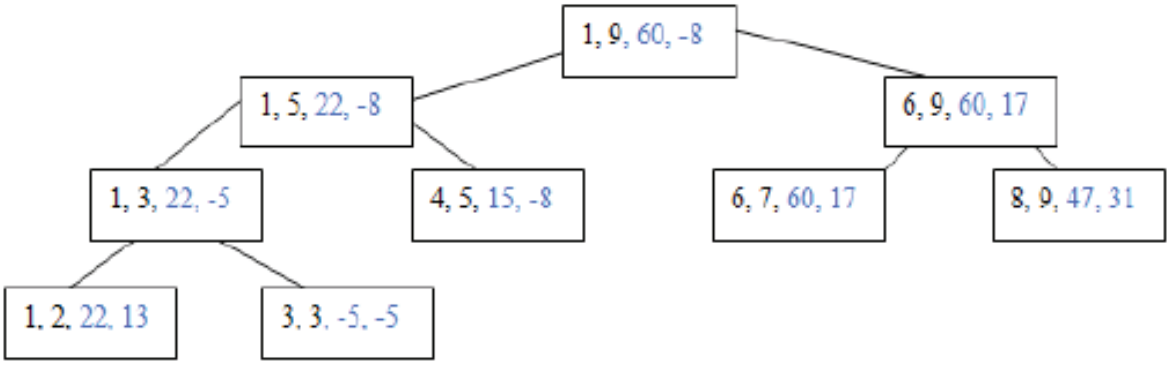
```
#include <stdio.h>
void MaxMin(int [],int, int, int *, int *);
int max(int, int);
int min(int, int);

main(){ int fmax,fmin,a[]={4,2,10,5,-7,9,80,6,3,1};
MaxMin(a,0,9,&fmax,&fmin);
printf("El maximo valor es %d y el minimo valor es %d\n",fmax,fmin);
getchar();
}

void MaxMin(int a[],int i, int j, int * fmax, int *fmin){
   int gmax,gmin,hmax,hmin,mid;
   if (i==j) *fmax=*fmin=a[i];
   else if (i==j-1)
   if (a[i]<a[j]){
   *fmax=a[j];
   *fmin=a[i]; }
   else{*fmax=a[i];
   *fmin=a[j]; }
   else{ mid=(i+j)/2;
   MaxMin(a,i,mid,&gmax,&gmin);
   MaxMin(a,mid+1,j,&hmax,&hmin);
   *fmax=max(gmax,hmax);
   *fmin=min(gmin,hmin); } }

int max(int g, int h){
if (g>h) return g;
return h; }

int min(int g,int h){
if (g<h) return g;
return h;
}
```

Por ejemplo:

A(n) 22 13 -5 -8 15 60 17 31 47.

En la figura 2.6 se muestra el árbol recursivo:

Figura 2.6. Árbol recursivo con el algoritmo máximo y mínimo



¿Cuántas comparaciones se requieren?

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) & n>2 \\ 1 & n=2 \\ 0 & n=1 \end{cases}$$

Cuando n es potencia de 2, $n = 2^k$. Por lo tanto,

$T(n) = 2T(n/2) +2 = 2(2T(n/4)+2)+2 = 4T(n/4) + 4 +2 = 4(2T(n/8)+2) + 4 + 2\ T(n) = 8T(n/8)+8 + 4 + 2 = 8(2T(n/16)+2) + 8 + 4 + 2 = 16T(n/16)+16+8+4+2 \ldots$

$T(n) = 2^{k-1} T(2) + \sum 2^i = 2^{k-1} + 2^k - 2 = 3n/2 - 2,$

donde $2^{k-1} = n / 2$ y la sumatoria corre de la siguiente forma: $1 \leq i \leq k-1$.

Si n es igual a 16 se tiene:

$T(16) = 2T(8) + 2 = 2(2T(4) + 2) + 2 = 4T(4) + 4 + 2 = 4(2T(2) + 2) + 4 + 2$

$T(16) = 8T(2) + 8 + 4 + 2,$

donde $n = 2^4$, $K = 4$ y $T(2) = 1$; por tanto,

$T(16) = 2^3 + 2^4 - 2 = 3*16 / 2 - 2 = 22.$

Véase que $3n / 2 - 2$ es el mejor promedio y peor caso si n es potencia de 2. Contrastado con 2n-2 comparaciones existe un ahorro de 25 %, por lo que es mejor que el secuencial. Pero ¿esto indica que sea mejor en la práctica? No necesariamente. En términos de almacenamiento es peor, ya que requiere una pila para guardar a *i, j, fmax, fmin.* Dados n elementos, se requieren log n +1 niveles de recursión. Se necesitan guardar cinco valores y el direccionamiento de retorno. Por ende, MaxMin es más ineficiente porque se maneja una pila y recursión.

Método codicioso (greedy)

INTRODUCCIÓN

Un juego como el ajedrez solo se gana pensando en las posibles jugadas futuras; por ende, un jugador que solo se focalice en la jugada presente es fácil de vencer. Pero en otros juegos, tales como Scrabble, es posible hacer un buen juego simplemente realizando una jugada que parezca apropiada en el momento, sin preocuparse demasiado en las consecuencias.

Este tipo de comportamiento miope es fácil y conveniente, lo cual genera una estrategia atractiva para algún tipo de problema. Los algoritmos avaros construyen una solución pieza por pieza, siempre escogiendo la siguiente pieza que ofrece el obvio e inmediato beneficio (Dasgupta, Papadimitriou y Vazirani, 2008).

La mayoría de estos problemas tienen *n* entradas y requieren un subconjunto que satisfaga ciertas restricciones. Cualquier subconjunto que satisfaga estas restricciones se conoce como solución factible. El problema demanda una solución factible que maximice o minimice una función objetivo dada. Existe una forma obvia de encontrar un punto factible, pero no necesariamente óptimo.

El método *greedy* sugiere que uno puede hacer un algoritmo que trabaje por pasos, considerando una entrada a la vez. En cada paso se realiza una decisión para seguir la ruta que en ese momento puede maximizar la ganancia o minimizar el costo (nótese que no garantiza el óptimo global, ya que sus decisiones son locales). El algoritmo 3.1 refleja en pseudocódigo la estrategia *greedy*:

Algoritmo 3.1. Estrategia general del algoritmo greedy (Horowitz y Sahni, 1978)

```
Procedimiento Greedy
//A (1: n) contiene n entradas
solución ← ϕ
para i ← 1 a n realiza
   x ← selecciona (A)
   Si es factible (x) entonces
        Solución ← Unión (Solución, x)
    Fin del si
Fin del para
Retorna (solución)
Fin Greedy
```

ALMACENAMIENTO ÓPTIMO EN CINTAS (OPTIMAL STORAGE ON TAPES)

Existen n archivos que son almacenados en una cinta de tamaño *L*. Asociado con cada archivo i hay un tamaño L_i, $1 \le i \le n$. Todos los archivos pueden ser guardados en cinta *iff* la suma del tamaño de los archivos es máximo L. Si los archivos son guardados en orden $I = i_1, i_1, i_2, i_3, \ldots, i_n$ y el tiempo requerido para guardar o recuperar el archivo i_j es $T_j = \sum_{1 \le k \le j} L_{i,k}$. Si todo archivo es leído con la misma frecuencia, entonces el tiempo de referencia medio (TRM) es $(1/n) \sum_{1 \le j \le n} t_j$. En el problema del almacenamiento óptimo, se requiere encontrar una permutación para n de tal forma que se minimice el TRM. Minimizar TRM es equivalente a minimizar $D(I) = \sum_{1 \le k \le j} \sum_{1 \le k \le j} I_{i,k}$. Por ejemplo:

N = 3 $(I_1, I_2, I_3) = (5, 10, 3)$.

Existen N! = 6 posibles ordenaciones.

1, 2, 3	$5 + 5 + 10 + 5 + 10 + 3 = 38$
1, 3, 2	$5 + 5 + 3 + 5 + 3 + 10 = 31$
2, 1, 3	$10 + 10 + 5 + 10 + 5 + 3 = 43$
2, 3, 1	$10 + 10 + 3 + 10 + 3 + 5 = 41$
3, 1, 2	$3 + 3 + 5 + 3 + 5 + 10 = 29$
3, 2, 1	$3 + 3 + 10 + 3 + 10 + 5 = 34$

El orden óptimo es (3, 1, 2). En este algoritmo, el método greedy requiere que los archivos sean almacenados en forma creciente Esta ordenación puede realizarse por medio de un algoritmo de ordenación como *merge sort*, por lo que requiere $\mathcal{O}$ (n log n).

EL PROBLEMA DE LA MOCHILA (KNAPSACK PROBLEM)

Se tienen n objetos y una mochila. El objeto i tiene un peso W_i y la mochila tiene una capacidad *M*. Si una fracción X_i, $0 \le Xi \le 1$ del objeto i se introduce, se tendrá una ganancia $P_i X_i$. El objetivo es llenar la mochila de tal forma que maximice la ganancia. El problema es:

$$Max \sum_{1 \le i \le n} P_i X_i \qquad (1)$$

S. A.

$$\sum_{1 \le i \le n} W_i X_i \le M \qquad (2)$$

$$0 \le X_i \le 1 \quad 1 \le i \le n \qquad (3)$$

Una posible solución es cualquier conjunto ($X_1, X_2, \ldots, X_n$) que satisfaga (2) y (3). Una solución óptima es una solución factible en la cual maximice la ganancia (1). Por ejemplo:

$$n = 3,\ M = 20,\ (P_1, P_2, P_3) = (25, 24, 15)$$

$$(W_1, W_2, W_3) = (18, 15, 10).$$

Cuatro posibles soluciones son:

		$\sum W_i X_i$	$\sum P_i X_i$
i.	(1/2,1/3,1/4)	16.5	24.25
ii.	(1,2/15,0)	20	28.2
iii.	(0,2/3,1)	20	31
iv.	(0,1,1/2)	20	31.5

De las cuatro soluciones, la cuarta produce un máximo.

Toda solución óptima llena la mochila al máximo.

Se pueden tener tres estrategias:

a. Escoger los elementos con mayor ganancia (ii).
b. Escoger los elementos con menor peso (iii).
c. Un radio entre P_i / W_i (iv).

La estrategia c produce un óptimo y requiere solo $\mathcal{O}(n)$.

El algoritmo 3.2 localiza el óptimo siempre y cuando $P(i+1) / W(i+1) \leq P(i) / W(i)$:

Algoritmo 3.2. Método *greedy* para el problema de la mochila

```
#include <stdio.h>
#define N 3
main(){
   float Cu,M=20, P[]={24,15,25},W[]={15,10,18},X[]={0,0,0},ganancia=0;
   float R[N];
   int i;
   Cu=M;
   for (i=0;i<N;i++){
   if(W[i]>Cu)
   break;
   X[i]=1;
   Cu=Cu-W[i];
    }
   if (i<N)
   X[i]=Cu/W[i];
   for (i=0;i<N;i++){
   printf("X[%d]=%f..",i,X[i]);
   ganancia=ganancia+X[i]*P[i]; }
   printf("\nGanancia=%f\n", ganancia);
   getchar();
}
```

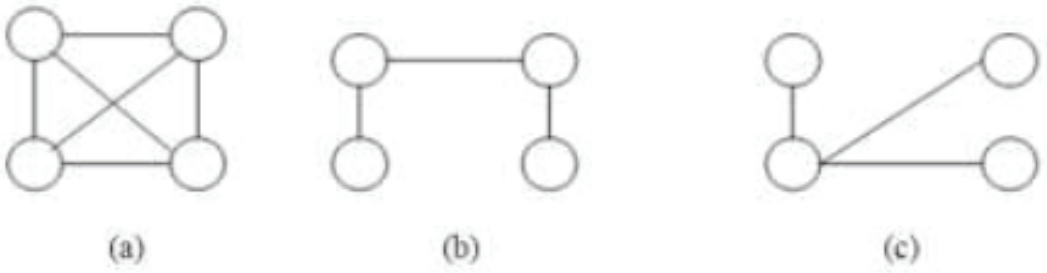
Mientras que las dos primeras estrategias no garantizan la solución óptima para el problema de la mochila, el siguiente teorema muestra que la tercera estrategia siempre obtiene una solución óptima.

Teorema: *Si* $P_1/W_1 \geq P_2/W_2 \geq P_3/W_3 \geq \ldots \geq P_n/W_n$, entonces el algoritmo de la mochila genera un óptimo a la instancia dada por el problema.

ÁRBOL DE EXPANSIÓN MÍNIMA (MINIMUN SPANNING TREE)

Definición: Sea G (*V*, E) (donde V es el conjunto de vértices y E el conjunto de aristas, arcos o segmentos que unen a los nodos), por lo que una gráfica es la forma en que se unen los vértices mediante las aristas. En la figura 3.1 se muestran ejemplos de gráficas no direccionadas. Un árbol es una gráfica que no tiene ciclos. Una subgráfica T (*V*, E‘) es un árbol de extensión de G si y solo si T es un árbol. Por ende, las figuras 3.1b y 3.1c son árboles.

Figura 3.1. Ejemplo de gráficas



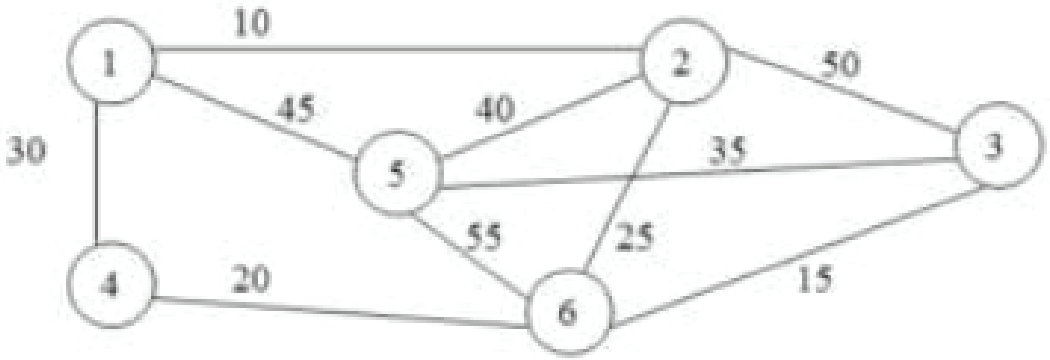
Si los nodos representan ciudades y el segmento que las une simboliza una posible unión entre ambas ciudades, entonces el mínimo número de segmentos para empalmar las *n* ciudades es *n-1*. El árbol de extensión representa todas las posibles combinaciones.

En una situación práctica, los segmentos tendrán pesos asignados, los cuales pueden representar costos de construcción, distancias, etc. Ahora, dado un peso, lo que se desea es conectarse a todos los nodos con el mínimo costo. En cualquier caso, los segmentos seleccionados formarán un árbol (suponiendo todos los costos positivos). El interés será localizar un árbol de extensión en G con el costo mínimo.

Un método *greedy* para obtener un mínimo costo será ir edificando el árbol segmento por segmento. El siguiente segmento para escoger será aquel en que se minimice el incremento en costos.

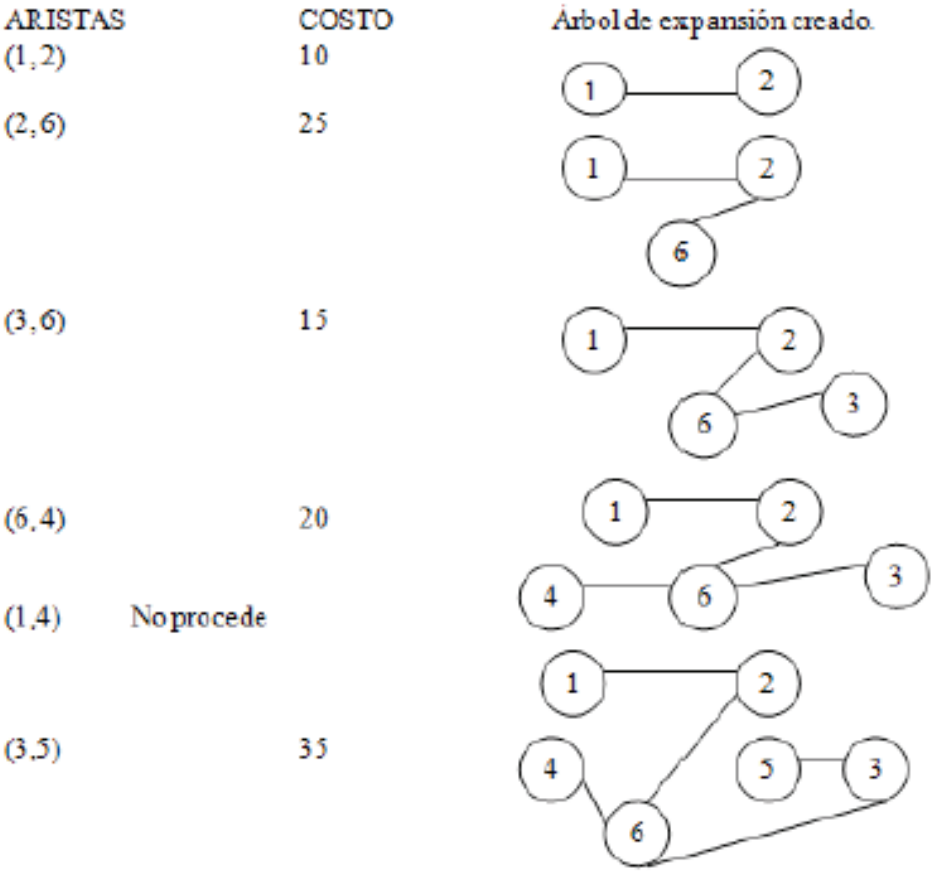
Si A es el conjunto de segmentos seleccionados hasta el momento, entonces *A* forma un árbol. El siguiente segmento (u, v) a ser incluido en A es un segmento con costo mínimo no en *A* con la propiedad de que A U {u, v} también es un árbol. Este se conoce como el algoritmo de Prim.

Figura 3.2. Un ejemplo de grafo no dirigido con costos (Horowitz y Sahni, 1978)



Utilizando como grafo a la figura 3.2, en la siguiente imagen se muestra la forma en que trabaja el método Prim (figura 3.3). El árbol de expansión tiene un costo de 105.

Figura 3.3. Solución paso a paso del árbol de expansión mínima (Horowitz y Sahni, 1978)



El algoritmo inicia con un árbol que incluye solo un borde con costo mínimo de *G*. Entonces, las aristas serán adicionadas al árbol una por una. La siguiente arista *(i, j)* a ser adicionada es tal que el vértice i se encuentra incluido en el árbol, *j* es el vértice aún no incluido y el costo *(i, j)* es mínimo sobre todas las aristas *(k, l)*, donde el vértice *k* es parte del árbol y el vértice *l* no se encuentra en el árbol.

Para determinar este vértice *(i, j)* en forma eficiente, nosotros asociamos por cada vértice j aún no incluido en el árbol un valor NEAR(j). NEAR(j) es un vértice en el árbol tal que costo (j, NEAR(j)) es mínimo sobre todas las elecciones para NEAR(j). Se define NEAR(j) = 0 para todos los vértices j que se encuentran en el árbol. La siguiente arista a incluir es definida por el vértice j, tal que NEAR(j) ≠ 0 (j no se encuentra aún en el árbol) y costo (j, NEAR(j)) es mínimo. El algoritmo 3.3 muestra la estrategia Prim para el árbol de expansión mínima:

Algoritmo 3.3. Estrategia Prim

```
#include <stdio.h>
#define TAM 6
main(){
//costo (n, n) es la matriz de costos del trayecto del grafo
//El costo (i, i) es infinito, el costo(i,j), donde i!=j, es positivo.
//El trayecto se guarda en el arreglo T(n,2)
//El costo final se asigna a minicost.
//Se requiere un vector que indique el nodo más cercano al nodo j, ese
vector
//NEAR se guarda en la variable ne.
Int
cost[TAM][TAM]={{999,10,999,30,45,999},{10,999,50,999,40,25},{999,50,999,9
99,35,15},{30,999,999,999,999,20},{45,40,35,999,999,55},{999,25,15,20,55,9
99}};
int ne[6], n, I, j, k, l, t[6][2], min=999,mincost;
for (i=0;i<TAM;i++)
   for (j=0;j<TAM;j++)
   if (min>cost[i][j]){
   min=cost[i][j];
   k=I;
   l=j; }
mincost=cost[k][l];
t[0][0]=k;
t[0][1]=l;
for (i=0;i<TAM;i++)
if (cost[i][l]<cost[i][k])
   ne[i]=l;
else
   ne[i]=k;
   ne[k]=ne[l]=-1;
for (i=1;i<TAM-1;i++){
   min=999;
for (k=0;k<TAM;k++){
   if (ne[k]!=-1 && cost[k][ne[k]]<min){
   min=cost[k][ne[k]];
   j=k;
    }
}
t[i][0]=j;
t[i][1]=ne[j];
mincost=mincost+cost[j][ne[j]];
ne[j]=-1;
for (k=0;k<TAM;k++)
if (ne[k]!=-1 && cost[k][ne[k]]>cost[k][j]) ne[k]=j;
 }
printf("Costo del árbol de expansión mínima=%d\n",mincost);
for (i=0;i<TAM-1;i++)
printf("Trayecto de %d a %d\n",t[i][0]+1,t[i][1]+1);
getchar(); getchar();
}
```

El tiempo para ejecutar el algoritmo Prim es del (n^2) donde n es el número de nodos. La matriz costo del ejemplo anterior junto con una simulación de los valores históricos del vector NEAR quedan como se observa en la tabla 3.1.

Tabla 3.1. Simulación de la estrategia Prim utilizando el algoritmo 3.3 para la figura 3.2

	1	2	3	4	5	6	NEAR
1	∞	10	∞	30	45	∞	0
2	10	∞	50	∞	40	25	0
3	∞	50	∞	∞	35	15	2, 6
4	30	∞	∞	∞	∞	20	1, 6, 0
5	45	40	35	∞	∞	50	2, 3, 0
6	∞	25	15	20	55	∞	2, 0

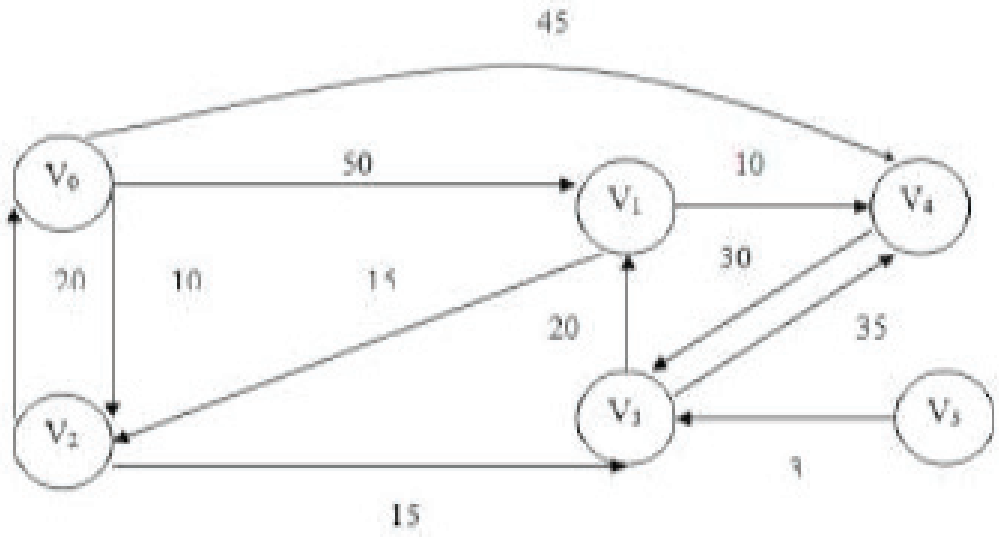
LA RUTA MÁS CORTA A PARTIR DE UN ORIGEN (SINGLE SOURCE SHORTEST PATHS)

Los grafos pueden ser utilizados para representar carreteras donde los vértices simbolizan ciudades y los segmentos que las unen son la carretera. Los segmentos pueden tener asignados pesos que marcan una distancia entre dos ciudades conectadas.

La distancia de un trayecto es definida por la suma del peso de los segmentos. El vértice de inicio se definirá como el origen y el último como el destino. El problema para considerar se basará en una gráfica dirigida *G = (V, E),* una función de peso c(e) para los segmentos de *G* y un vértice origen v_0. El problema es determinar el trayecto más corto de v_0 a todos los demás vértices de *G*. Se asume que todos los pesos son positivos.

Ejemplo: Considere la gráfica dirigida de la figura 3.4. El número de trayectos también es el número de pesos. Si v_0 es el vértice de origen, entonces el trayecto más corto desde v_0 *a* v_1 *es* $v_0\, v_2 v_3 v_1$. La distancia del trayecto es 10 + 15 + 20 = 45. En este caso, recorrer tres caminos es más económico que transitar en forma directa $v_0\, v_1$, el cual tiene un costo de 50.

Figura 3.4. Un ejemplo para el algoritmo del trayecto más corto (Horowitz y Sahni, 1978)



Para formular un algoritmo greedy y generar el trayecto más corto, debemos concebir una solución multietapa. Una posibilidad es construir el trayecto más corto uno por uno. Como una medida de optimización se puede usar la suma de todos los trayectos generados hasta el momento. En orden de que la medida sea mínima, cada trayecto individual debe ser de tamaño mínimo. Si se han construido i trayectos mínimos, el siguiente que debe ser construido debería ser el próximo trayecto con mínima distancia.

El camino *greedy* para formar los trayectos cortos desde v_0 a los vértices remanentes se logra generando los trayectos en orden creciente. Primero, se crea el trayecto más corto al vértice más cercano. Entonces el trayecto más corto al segundo vértice más cercano se genera, y así sucesivamente. En la figura 3.4, el trayecto más cercano para v_0 es v_2 *(c* $(v_0, v_2) = 10)$, por lo que el trayecto $v_0\ v_2$ será el primero generado.

El segundo trayecto más cercano de v_0 *es* v_2, v_3 con una distancia de 25. Para generar los siguientes trayectos se debe determinar i) el siguiente vértice con el cual deba generar un camino más corto y ii) un camino más corto para cada uno de los vértices partiendo del vértice original. Sea S el conjunto de vértices (incluyendo V_0) en el cual el trayecto más corto ha sido generado.

Sea W el conjunto de vértices no en S, sea DIST(w) la distancia del trayecto más corto desde v_0 yendo solo a través de estos vértices que están en *S* y terminando en *w*. Se observa lo siguiente:

I. Si el siguiente trayecto más corto es al vértice u, el trayecto inicia en v_0, termina en u y va a través de algunos de los vértices localizados en *S*.

II. El destino del siguiente trayecto generado debe de ser aquel vértice u tal que forme la mínima distancia —DIST(u)— sobre todos los vértices, no en S.

III. Habiendo seleccionado un vértice u en II y generado el trayecto más corto de v_0

a u, el vértice u viene a ser miembro de S. En este punto la dimensión del trayecto más corto iniciando en v_0 irá en los vértices localizados en S y terminando en w no en S puede decrecer. Esto es, el valor de la distancia *DIST(w)* puede cambiar. Si cambia, se debe a que existe un trayecto más corto iniciando en v_0 y posteriormente va a u y entonces a w. Los vértices intermedios de v_0, *a u* y de u a w deben estar todos en *S*. Además, el trayecto v_0 a u debe ser el más corto; de otra manera, DIST(w) no está definido en forma apropiada. También, el trayecto u a w puede no contener vértices intermedios.

Las observaciones arriba indicadas crean un algoritmo simple, el cual fue desarrollado por Dijkstra (este algoritmo es la base para el ruteo en internet). De hecho, solo determina la magnitud de la trayectoria del vértice v_0 *a* todos los vértices en *G.*

Se asume que los n vértices en G se numeran de 1 a *n*. El conjunto se mantiene *S* con un arreglo con *S(i)* = 0 si el vértice i no se encuentra en *S y S(i) = 1* si pertenece a S. Se asume que la gráfica se representa por una matriz de costos. Horowitz y Sahni (1978) muestran el algoritmo general 3.4 en formato pseudocódigo.

Algoritmo 3.4. Método de Dijkstra para la ruta más corta

```
Procedure SHORTEST-PATHS (v, COST, DIST, n)

   //DIST(j) es el conjunto de longitudes del trayecto más corto del vértice v
     al
   //vértice j en la gráfica G con n vértices.
   //G es representada por la matriz de costos COST (n, m)
   Bolean S (1: n); real COST (1: n, 1:n), DIST(1:n)
   Integer u, v, n, num, i, w
   For i ← 1 to n do
       S(i) ← 0; DIST(i) ← COST (v, i)
   Repeat
   S(v) ← 1 DIST(v) ← 0 // colocar el vértice v en S.
   For num← 2 to n-1 do //determina n - 1 trayectos desde el vértice v.
       Escoger u tal que DIST(u) = min {DIST(w)}
       S(w) =0
              S(u) ← 1       //Coloca el vértice u en S
              For all w con S(w) =0 do
                     DIST(w) ← min (DIST(w), DIST(u) + COST (u, w))
              Repeat
    Repeat
End SHORTEST-PATH.
```

En el algoritmo 3.5 se muestra el código en lenguaje de programación C con matrices declaradas en forma dinámica:

Algoritmo 3.5. Programación del método de Dijkstra

```
#include<stdio.h>
#include<stdlib.h>
//La ruta más corta a partir de un origen
void generar(int **,int);
void path(int **,int *,int,int);

main(){ int **cost,*dist,tam,I,j,v;
printf(" ********** Ruta mas corta a partir de un origen **********\n");
printf("\nIntroduce el número de 77értice77: "); scanf("%d",&tam);
cost=(int **)malloc(sizeof(int *)*tam);
for(i=0;i<tam;i++)
cost[i]=(int *)malloc(sizeof(int)*tam);
dist=(int *)malloc(sizeof(int)*tam);
for(i=0;i<tam;i++)
for(j=0;j<tam;j++)
cost[i][j]=9999;
generar(cost,tam);
printf("Introduce el 77értice de origen: "); scanf("%d",&v);
printf("La matriz generada es: \n");
for(i=0;i<tam;i++){
for(j=0;j<tam;j++)
printf("%6d",cost[i][j]);
printf("\n");
 }
path(cost,dist,tam,v-1);
printf("\n\nCosto de los caminos\n");
for(i=0;i<tam;i++)
printf("Camino %d a %d: %d\n",v,i+1,dist[i]);
system("PAUSE");
}

void generar(int **cost,int tam){
int i,nv=0,p,ad;
printf("Para terminar de introducir un 77értice introduce 99\n");
while(nv<tam){
   printf("Vertice %d a... \n",nv+1);scanf("%d",&ad);
   if(ad==99){ printf("Termino vertice %d\n",nv+1);
   nv++;
}
else if(ad>tam) printf("El vertice no existe \n");
else{ printf("Intorduce el peso del Vertice:");
scanf("%d",&p);
cost[nv][ad-1]=p;
 }
 }
}

void path(int **cost,int *dist,int tam,int v){
int u,w,I,num,s[tam],min;
for(i=0;i<tam;i++){
   s[i]=0;
   dist[i]=cost[v][i];
 }
   s[v]=1;
   dist[v]=0;
```

```
#include<stdio.h>
#include<stdlib.h>
//La ruta más corta a partir de un origen
void generar(int **,int);
void path(int **,int *,int,int);

main(){ int **cost,*dist,tam,I,j,v;
printf(" ********** Ruta mas corta a partir de un origen **********\n");
printf("\nIntroduce el número de 77értice77: "); scanf("%d",&tam);
cost=(int **)malloc(sizeof(int *)*tam);
for(i=0;i<tam;i++)
cost[i]=(int *)malloc(sizeof(int)*tam);
dist=(int *)malloc(sizeof(int)*tam);
for(i=0;i<tam;i++)
for(j=0;j<tam;j++)
cost[i][j]=9999;
generar(cost,tam);
printf("Introduce el 77értice de origen: "); scanf("%d",&v);
printf("La matriz generada es: \n");
for(i=0;i<tam;i++){
for(j=0;j<tam;j++)
printf("%6d",cost[i][j]);
printf("\n");
 }
path(cost,dist,tam,v-1);
printf("\n\nCosto de los caminos\n");
for(i=0;i<tam;i++)
printf("Camino %d a %d: %d\n",v,i+1,dist[i]);
system("PAUSE");
}

void generar(int **cost,int tam){
int i,nv=0,p,ad;
printf("Para terminar de introducir un 77értice introduce 99\n");
while(nv<tam){
   printf("Vertice %d a... \n",nv+1);scanf("%d",&ad);
   if(ad==99){ printf("Termino vertice %d\n",nv+1);
   nv++;
}
else if(ad>tam) printf("El vertice no existe \n");
else{ printf("Intorduce el peso del Vertice:");
scanf("%d",&p);
cost[nv][ad-1]=p;
 }
 }
}

void path(int **cost,int *dist,int tam,int v){
int u,w,I,num,s[tam],min;
for(i=0;i<tam;i++){
   s[i]=0;
   dist[i]=cost[v][i];
 }
   s[v]=1;
   dist[v]=0;
```

El tiempo que tarda el algoritmo con n vértices es $\mathcal{O}(n^2)$. Esto se ve fácilmente, ya que el algoritmo contiene dos for anidados. Un ejemplo se aprecia en la figura 3.5 y su matriz de costos en la tabla 3.2.

Figura 3.5. Un ejemplo para el trayecto más corto (Horowitz y Sahni, 1978)

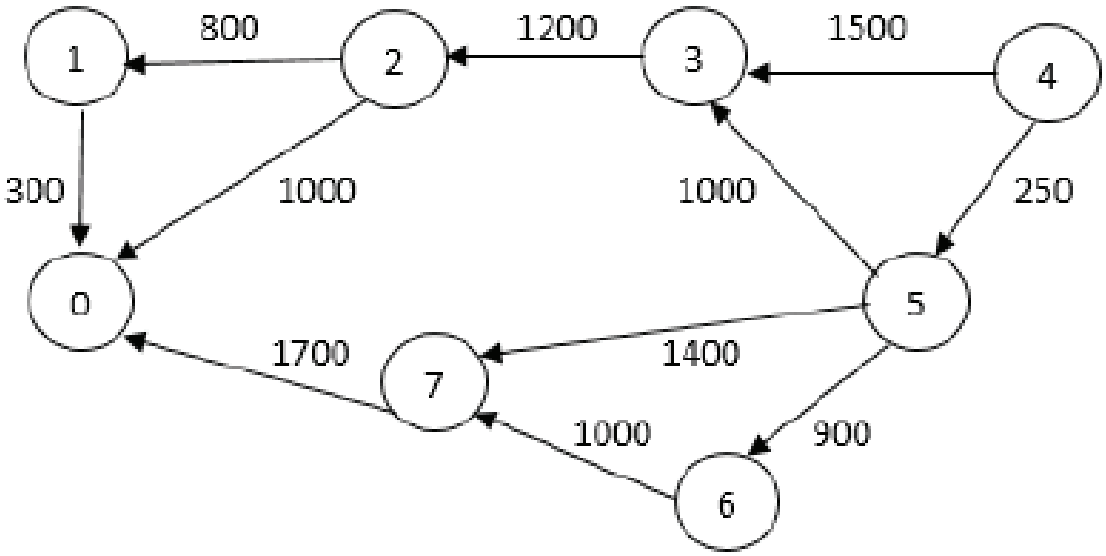


Tabla 3.2. Matriz de costos

	0	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞	∞
1	**300**	**0**	∞	∞	∞	∞	∞	∞
2	1000	**800**	**0**	∞	∞	∞	∞	∞
3	∞	∞	**1200**	**0**	∞	∞	∞	∞
4	∞	∞	∞	**1500**	**0**	**250**	∞	∞
5	∞	∞	∞	**1000**	∞	**0**	**900**	**1400**
6	∞	∞	∞	∞	∞	∞	**0**	**1000**
7	**1700**	∞	∞	∞	∞	∞	∞	**0**

Si *v = 4*, indica que se busca el trayecto de mínimo costo a todos los nodos desde el nodo 4. En la tabla 3.3 se evidencia una prueba de escritorio para la función *path ()*; véase que cada columna representa una iteración dada por la variable *num*:

Tabla 3.3. Simulación del algoritmo 3.5 utilizando la tabla 3.2

V = 4

S

num	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	**0**	**0**	**0**	**0**	**0**	0	1
2	0	0	0	0	0	1	0
3	**0**	0	0	1	0	0	**0**
4	0,1	0	**0**	0	**0**	**0**	0
5	0	1	0	**0**	0	0	0
6	**0**	0	1	0	0	0	**0**
7	**0**	**0**	0	0	1	0	0

DIST

num	0	1	2	3	4	5	6
u	**4**	**4**	**4**	**5**	**7**	**2**	**1**
0	∞	0	0	0	3350	0	0
1	∞	**0**	**0**	**0**	**0**	3250	1
2	∞	0	0	2450	0	0	0
3	**1500**	1250	0	0	0	0	**0**
4	0	0	**0**	0	**0**	**0**	0
5	250	0	0	**0**	0	0	0
6	∞	1150	0	0	0	0	**0**
7	∞	**1650**	0	0	0	0	0

Programación dinámica (dynamic programming)

INTRODUCCIÓN

Con la estrategia del avaro o codicioso se intenta buscar en forma local el siguiente punto para mejorar la función (maximizar la ganancia o minimizar el costo). La técnica de la programación dinámica busca un óptimo global; su estrategia es localizar todos los subóptimos de un grafo dado.

PRINCIPIO DE OPTIMALIDAD DE BELLMAN

Cuando hablamos de *optimizar* nos referimos a buscar la **mejor** solución (pueden ser varias) de entre muchas alternativas posibles. Este proceso de optimización puede ser visto como una secuencia de decisiones que nos proporcionan la solución correcta. Si dada una subsecuencia de decisiones siempre se conoce cuál es la decisión que debe tomarse a continuación para obtener la secuencia óptima, el problema es elemental y se resuelve trivialmente tomando una decisión detrás de otra, lo que se conoce como estrategia voraz. En otros casos, aunque no sea posible aplicar la estrategia voraz, puede cumplirse el principio de optimalidad de Bellman enunciado en 1957, el cual señal que "dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima".

En este caso, sigue siendo posible ir tomando decisiones elementales, pues se confía que la combinación de ellas seguirá siendo óptima, aunque será necesario explorar muchas secuencias de decisiones para dar con la correcta, punto en el cual interviene la programación dinámica. Aunque este principio parece evidente, no siempre es aplicable y, por tanto, es necesario verificar que se cumpla para el problema en cuestión.

Contemplar un problema como una secuencia de decisiones equivale a dividirlo en problemas más pequeños y, en consecuencia, más fáciles de resolver como hacemos en divide y vencerás, técnica similar a la de programación dinámica, la cual se aplica cuando la subdivisión de un problema conduce a lo siguiente:

- Una enorme cantidad de problemas.
- Problemas cuyas soluciones parciales se solapan.
- Grupos de problemas de muy distinta complejidad.

Para que un problema pueda ser abordado por esta técnica debe cumplir dos condiciones:

- La solución al problema ha de ser alcanzada a través de una secuencia de decisiones, una en cada etapa.
- Dicha secuencia de decisiones ha de cumplir el principio de optimalidad de Bellman.

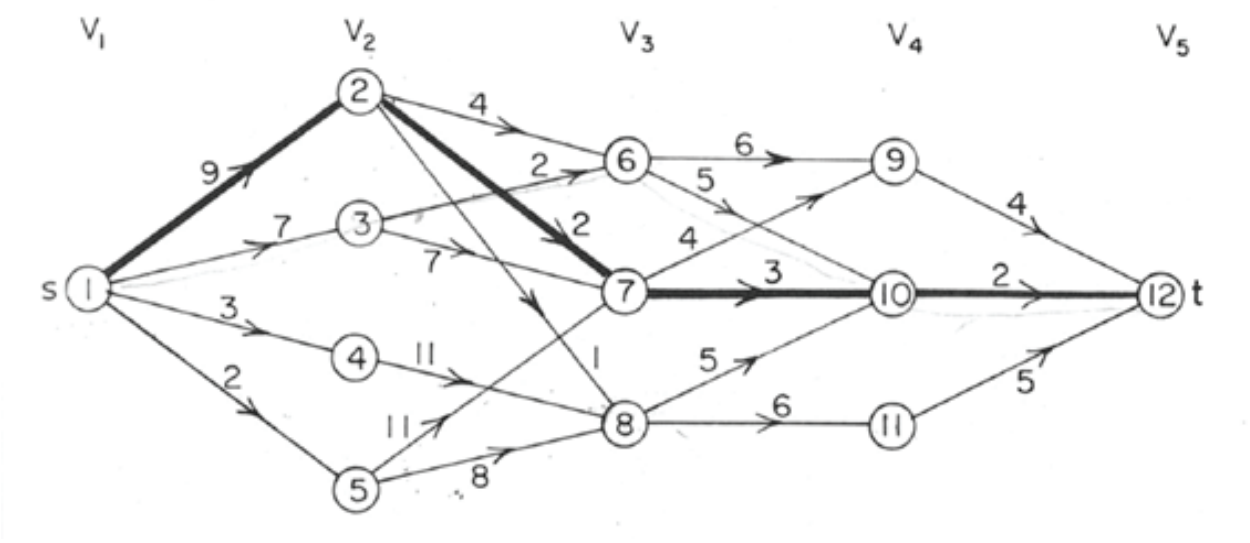
En general, el diseño de un algoritmo de programación dinámica consta de los siguientes pasos:

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que esta cumple el principio de optimalidad de Bellman.
2. Definición recursiva de la solución.
3. Cálculo del valor de la solución óptima mediante una tabla en la cual se almacenan soluciones de problemas parciales para reutilizar los cálculos.
4. Construcción de la solución óptima usando la información contenida en la tabla anterior.

GRÁFICAS DE MÚLTIPLES ETAPAS (MULTISTAGE GRAPHS)

Es una gráfica dirigida en el cual los vértices son particionados en K ≥ 2 conjuntos disjuntos V_i, $1 \leq i \leq k$ (cada V_i es una de las etapas con una determinada cantidad de nodos). En adición, si < u, v > son una arista en E, entonces u V_i y v V_{i+1} para algún i, $1 \leq i < k$. Los conjuntos V_1 y V_k son tales que $|V_1| = |V_k| = 1$. Sea s y t respectivamente el vértice en V_1 y V_k, donde s es la fuente y t es la meta para llegar. Sea c *(i, j)* el costo de la arista *<i, j>.* El costo del trayecto de s a t iniciando en el estado 1, va la etapa 2, posteriormente a la etapa 3, a la etapa 4, etc. Y eventualmente termina en la etapa k. En la figura 4.1 se muestra una gráfica de cinco etapas. El trayecto de mínimo costo de s a t se muestra en negrita.

Figura 4.1. Gráfica de cinco etapas (Horowitz y Sahni, 1978)



Una formulación de programación dinámica para un problema gráfico de k etapas se obtiene, primero, al darse cuenta de que todos los caminos de s a t son el resultado de una secuencia de k-2 decisiones. La i-ésima decisión consiste en determinar cuál vértice en V_{i+1}, $1 \le i \le k-2$ va a estar en el trayecto. Es fácil de observar que el principio de optimización se cumple. Sea COSTO(i, j) (donde i indica la etapa y j determina el número de nodo) el costo de este trayecto. Usando el enfoque hacia adelante, se obtiene:

$$\text{COSTO}(i,j) = \min_{\substack{L \in Vi+1 \\ (j,L) \in E}} \{C(j,L) + \text{COSTO}(i+1,L)\}$$

Resolviendo la gráfica de cinco etapas indicada en la figura 4.1, se obtienen los siguientes valores:

COSTO (3,6) = min {6 + COSTO (4,9), 5 + COSTO (4,10)} = min {6+4, 5+2} = 7

COSTO (3,7) = min {4 + COSTO (4,9), 3 + COSTO (4,10)} = min {4+4, 3+2} = 5

COSTO (3,8) = min {5 + COSTO (4,10), 6 + COSTO (4,11)} = min {5+2, 6+5} = 7

COSTO (2,2) = min {4 + COSTO (3,6), 2+COSTO (3,7), 1+ COSTO (3,8)} = 7

COSTO (2,3) = 9

COSTO (2,4) = 18

COSTO (2,5) = 15

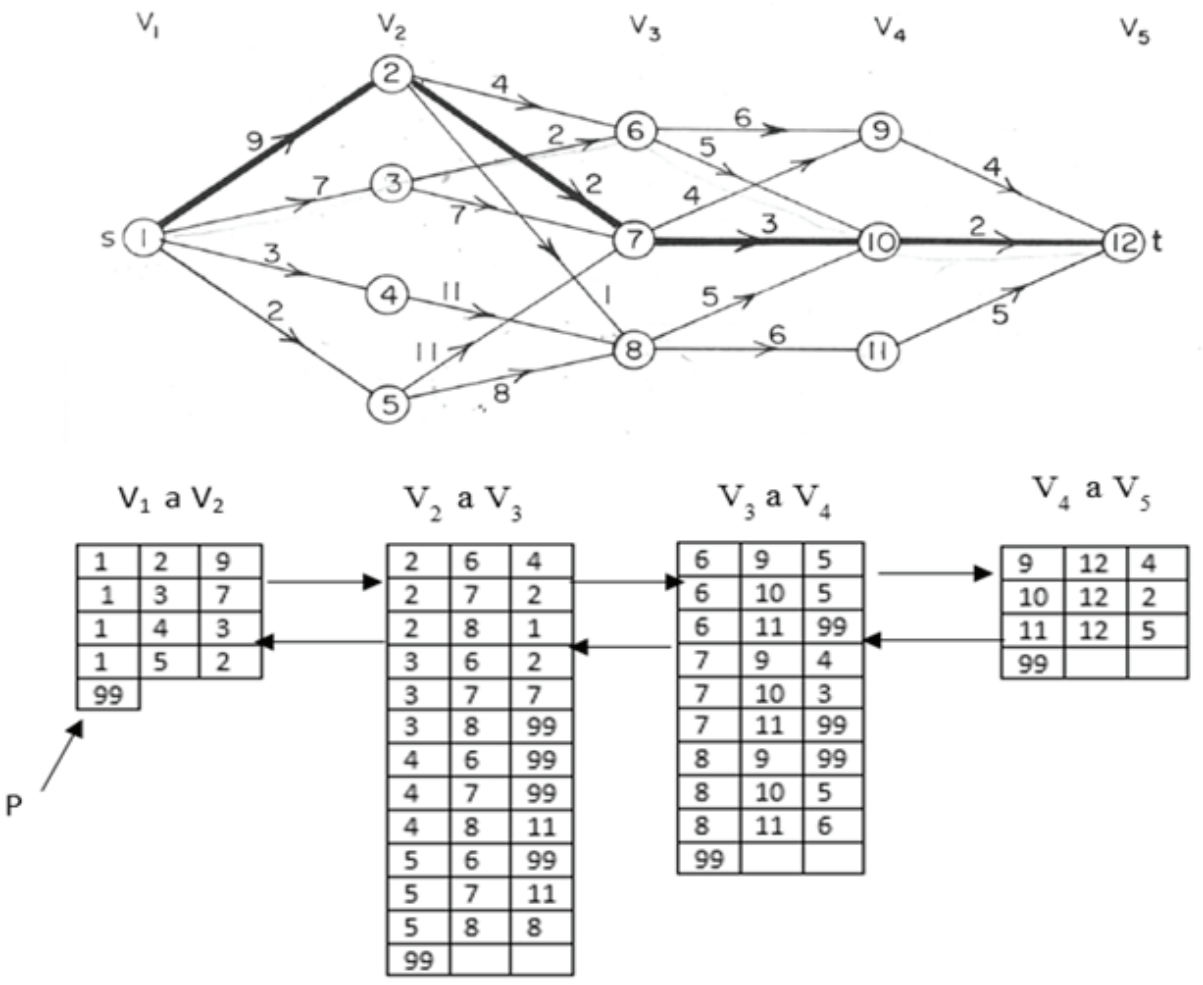
COSTO (1,1) = min {9+COSTO (2,2), 7 + COSTO (2,3), 3+COSTO(2,4), 2+COSTO(2,5)}=16.

Por lo anterior, el mínimo costo de *s a t* es 16.

UNA FORMA ITERATIVA

Una forma sencilla de programar el algoritmo es usar una tabla por etapa que indique las conexiones existentes de V_i *a* V_{i+1}. Para poder conectar cada una de las tablas se utiliza una lista doblemente enlazada. Las tablas se muestran en la figura 4.2:

Figura 4.2. Método multietapas utilizando listas doblemente enlazadas



Para realizar la programación secuencial de este problema se utilizaron listas doblemente enlazadas y tablas que se generan en forma dinámica.

En este caso, las tablas se llenan de izquierda a derecha y el óptimo de cada nodo se calcula de derecha a izquierda. El algoritmo 4.1 ilustra la estrategia iterativa utilizando listas doblemente enlazadas.

Algoritmo 4.1. Problema multietapas utilizando listas doblemente enlazadas

```
#include <stdio.h>
#include <stdlib.h>

struct NodoEtapa {
int ** nodo;//obtiene el valor del nodo de cada NodoEtapa a los nodos de
la NodoEtapa siguiente
NodoEtapa * izq; NodoEtapa * der;
};

void lectura(void);
void * crear (void, int, int);
void llenar (void*, int*, int);
void calcular (void*, int);
```

```
main(){ lectura(); }

void lectura (){
   int numeroEtapas, *ne, i, nodos = 2; //ne es el número de nodos por
etapa
   void * p = NULL;
   printf ("Da el número de etapas =>"); scanf ("%d", &numeroEtapas);
   ne = (int*) malloc(sizeof(int) * numeroEtapas);
   for (i = 1; i < numeroEtapas-1; i++){
    printf ("Da los nodos de la etapa %d=>", i+1); scanf ("%d", &ne[i]);
    nodos = nodos + ne[i];
   }
   printf ("No. De nodos =>%d\n", nodos);
   printf ("Nodos por etapa:\n");
   ne [0] = ne[numeroEtapas-1] = 1;//Inicializando la etapa incial y la final con 1
   ne[numeroEtapas]=0;
   for (I = 0; I < numeroEtapas; i++)
    printf ("%d\t", ne[i]);
   putchar('\n');
   for (I = 0; I < numeroEtapas; i++)
    p = crear (p, ne, i);              ////Crear los nodos de cada etapa
   llenar (p, ne, numeroEtapas);
   calcular (p, nodos);
   getchar (); getchar ();
}

void * crear (void * p, int * m, int nodo) {
    printf ("Etapa %d nodos %d\n", nodo+1, m[nodo]);
    NodoEtapa q, aux;
    int ** pm, f, j;
    q = (NodoEtapa *) malloc (sizeof (NodoEtapa));
    q->der = q->izq = NULL;
    if (p == NULL)
     p = q;
    else {
```

```
      aux = (NodoEtapa*) p;
      while(aux->der! = NULL) {
      aux = aux->der;
           }
      aux->der = q;
      q->izq = aux;
```

```
     }
     f = m[nodo] * m[nodo+1] + 1;
     pm = (int **) malloc (sizeof (int *) * f);
     for (j = 0; j < f; j++)
      pm[j] = (int*) malloc(sizeof(int) * 3);
     pm[f-1][0] = 999; //fin de tabla
     q->nodo = pm;
     return p;
}

void llenar (void *p, int a[], int x){
      NodoEtapa *q;
      q = (NodoEtapa*) p;
      int ** s;
      int i, j, k, t, w, z;
      s = (int **) malloc (sizeof (int *) * x);
      //En s se guarda el límite inferior y límite superior de cada etapa. Se
calcula //con el vector de número de nodos por etapa.
    for (j = 0; j < x; j++) {s[j] = (int*) malloc(sizeof(int) * 2);}
```

```
        z++;
        }
        }
        w++;
        q = q->der;
       }
}

void calcular (void *s, int x) {NodoEtapa *q;
      int * costo, min = 999, I, **a, b, k, band=0;
      costo = (int*) malloc (sizeof(int) * (x+1));
      q = (NodoEtapa*) s;
      costo[x] = 0;
      while (q->der! = NULL) {q = q->der;}
      printf ("\nCOSTO MINIMO DEL NODO K A LA META \n");
      while (band == 0) {
       a = q->nodo;
       i = 0;
       while (a[i][0]! = 999) {
       b = a[i][2] + costo [ a[i][1]];
       if (b < min)
       min = b;
       if (a[i][0]! =a [i+1][0]) {
       costo[a[i][0]] = min;
       printf ("Costo al nodo %d=%d\n", a[i][0], min);
       min = 999;
       }
       i++;
       }
       if (q->izq == NULL) {band = 1;} else {q = q->izq; }
      }
      printf ("El costo en nodo uno es=>%d\n", costo [1]);
}
```

UNA FORMA RECURSIVA

Como es un problema de programación dinámica, una de sus características es el tipo de solución, la cual se puede encontrar en forma recursiva. En el algoritmo 4.2 se demuestra el programa que resuelve el problema según dicha técnica.

Algoritmo 4.2. Programación del problema multietapas utilizando recursividad

```
#include <stdio.h>
#include <stdlib.h>
int MSG(int **, int, int, int*);

main(){
      int f,i,j,ini,fin,dist;
      int **pm, *c;
      printf("Da el número de nodos=>"); scanf("%d",&f);
      c=(int*)malloc(sizeof(int)*(f-1));
      pm=(int **)malloc(sizeof(int *)*(f-1));
      for (j=0;j<f;j++) pm[j]=(int*)malloc(sizeof(int)*f);

      for (i=0;i<f-1;i++)
      for (j=i+1;j<f;j++){
          printf("a[%d][%d]=",i+1,j+1); scanf("%d",&pm[i][j]); }
      printf("LA MATRIZ ES:\n");
      for (i=0;i<f-1;i++){
         for (j=0;j<f;j++)
         if (j<=i) printf("%d\t",0);
         else printf("%d\t",pm[i][j]);
         putchar('\n');
      }
      dist=MSG(pm,0,f,c);
      printf("El vector de costos es:\n");
      for (i=0;i<f-1;i++) printf("COSTO(%d)=%d\n",i+1,c[i]);
      printf("El costo entre nodo 1 y nodo %d es %d\n",f,dist);
      getchar(); getchar();
}

int MSG(int ** costo, int ini, int tam, int * c){
      if (ini==tam-1)
      return 0;
      int min=9999, trayecto;
      for (int i=ini+1; i<tam;i++)
      if (costo[ini][i]>0){
      trayecto=costo[ini][i]+MSG(costo, i,tam,c);
      if (min>trayecto)
      min=trayecto;
                  }
      c[ini]=min;
      return min;
}
```

En la figura 4.3 se observa la ejecución del programa utilizando lo enseñado en la figura 4.1. Véase que a los nodos que no eran conexos se les dio el valor de cero.

Figura 4.3. Ejecución del programa utilizando lo enseñado en la figura 4.1

EL PROBLEMA DEL AGENTE VIAJERO (THE TRAVELING SALES PERSON PROBLEM, TSP)

Un problema que se resuelve por medio del agente viajero es el siguiente: suponga que se tiene la ruta de una camioneta postal que recoge el correo localizado en *n* sitios diferentes. Una gráfica de n+1 vértices pueden ser usada para representar tal situación. El vértice origen representa la oficina postal donde la camioneta inicia su recorrido y donde debe retornar. Toda arista *<i, j>* tiene asignado un costo igual a la distancia desde el sitio i al sitio *j*. La ruta tomada por la camioneta postal es una gira (visitar todos los sitios en forma no repetida) y lo que se espera es minimizar el trayecto. Un comentario interesante es que el costo de la arista <i, j> es diferente al costo de la arista < j, i >.

Hablando matemáticamente, sea G = (V, E) una gráfica dirigida con costo de cada arista c_{ij}, c_{ij} se define tal que $c_{ij} > 0$ para todo trayecto existente de i a j. Y $c_{ij} = \infty$ si <i, j> E. Sea |V| = n y asuma que n > 1. Una gira de G es un ciclo dirigido que incluye todos los vértices en V. El costo de la gira es la suma de los costos de las aristas en la gira. El problema del agente viajero es encontrar una gira que minimice los costos.

En la siguiente discusión se comentará sobre un recorrido que inicia en el vértice 1 y termina en el mismo vértice, siendo el recorrido el del mínimo costo. Toda gira consiste en una arista <1, k> para algún k ϵ V – {1} y un trayecto desde el vértice k al vértice 1. El trayecto desde el vértice k al vértice 1 va a través de cada vértice en V – {1, k}. De aquí que el principio de optimización se mantiene. Sea g (i, S) la longitud del trayecto más corto iniciando en el vértice i, yendo a través de todos los vértices en S y terminando en el vértice 1; g (1, V – {1}) es la longitud de una gira óptima de un agente viajero. Desde el principio de optimalidad se deduce lo siguiente:

$$g(1, V-\{1\}) = \min_{2 \le k \le n} \{c_{1,k} + g(k, V-\{1, k\})\} \quad \ldots\ldots\ldots 1$$

Generalizando, se obtiene (para $i \notin S$):

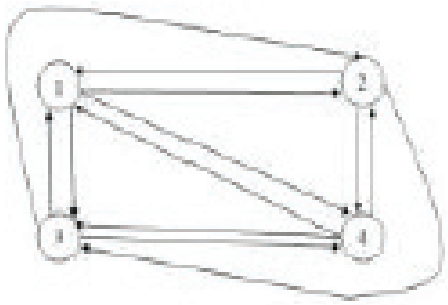
$$g(i, S) = \min_{j \in S} \{c_{i,j} + g(j, S-\{j\})\} \quad \ldots\ldots\ldots\ldots\ldots\ldots 2$$

Se puede resolver g (1, V- {1}) si conocemos g (k, V- {1, k}) para todas las opciones de k. Los valores de g pueden ser obtenidos usando (2). Claramente, g (i, ϕ) = $C_{i,1}$, 1 i n. De aquí podemos usar (2) para obtener g (i, S) para todo S de tamaño 1, entonces se puede obtener g (i, S) para S con |S| = 2, etc. Cuando |S| < n – 1, los valores de i y S para el cual se necesita g (i, S) son tales que $i \neq 1; 1 \notin S$ e $i \notin S$.

Por ejemplo, considere la gráfica de la figura 4.4, donde el tamaño de las aristas se observa en la matriz de costos c:

Figura 4.4. Gráfica dirigida cuya longitud de cada arista se localiza en la matriz C

(Horowitz y Sahni, 1978)



$$C = \begin{vmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{vmatrix}$$

Utilizando la ecuación (2) se obtienen los siguientes valores:

$$g(2, \phi) = c_{2,1} = 5; \qquad g(3, \phi) = c_{3,1} = 6; \qquad g(4, \phi) = c_{4,1} = 8;$$

Utilizando a (2) se obtiene:

$$g(2, \{3\}) = c_{2,3} + g(3, \phi) = 15; \qquad g(2, \{4\}) = c_{2,4} + g(4, \phi) = 18;$$

$$g(3, \{2\}) = 18; \qquad g(3, \{4\}) = 20;$$
$$g(4, \{2\}) = 13; \qquad g(4, \{3\}) = 15;$$

Ahora, calculamos $g(i, S)$ con $|S| = 2$, $i \neq 1$, $1 \notin S$ e $i \notin S$.

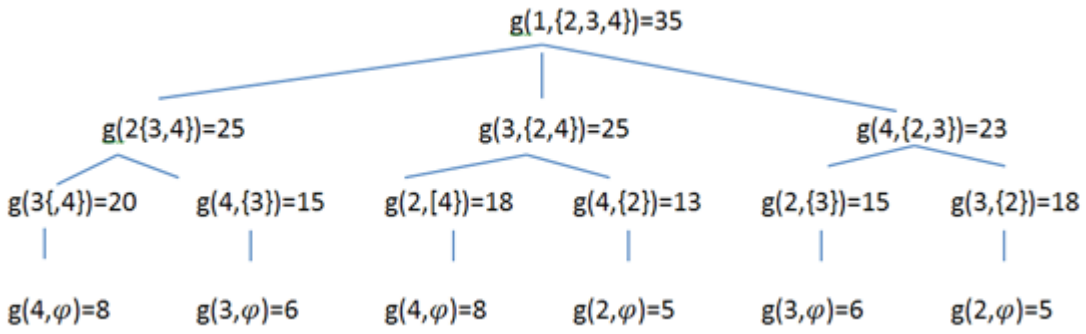
$$g(2, \{3,4\}) = \min \{c_{2,3} + g(3, \{4\}), c_{2,4} + g(4, \{3\})\} = 25$$
$$g(3, \{2,4\}) = \min \{c_{3,2} + g(2, \{4\}), c_{3,4} + g(4, \{2\})\} = 25$$
$$g(4, \{2,3\}) = \min \{c_{4,2} + g(2, \{3\}), c_{4,3} + g(3, \{2\})\} = 23$$

Finalmente, de (1) obtenemos:

$$g(1, \{2,3,4\}) = \min \{c_{1,2} + g(2, \{3, 4\}), c_{1,3} + g(3, \{2,4\}), c_{1,4} + g(4, \{2,3\})\}$$
$$= \min \{35, 40, 43\} = 35.$$

El árbol recursivo se observa en la figura 4.5.

Figura 4.5. Árbol recursivo del agente viajero



El algoritmo 4.3 programado en lenguaje C es:

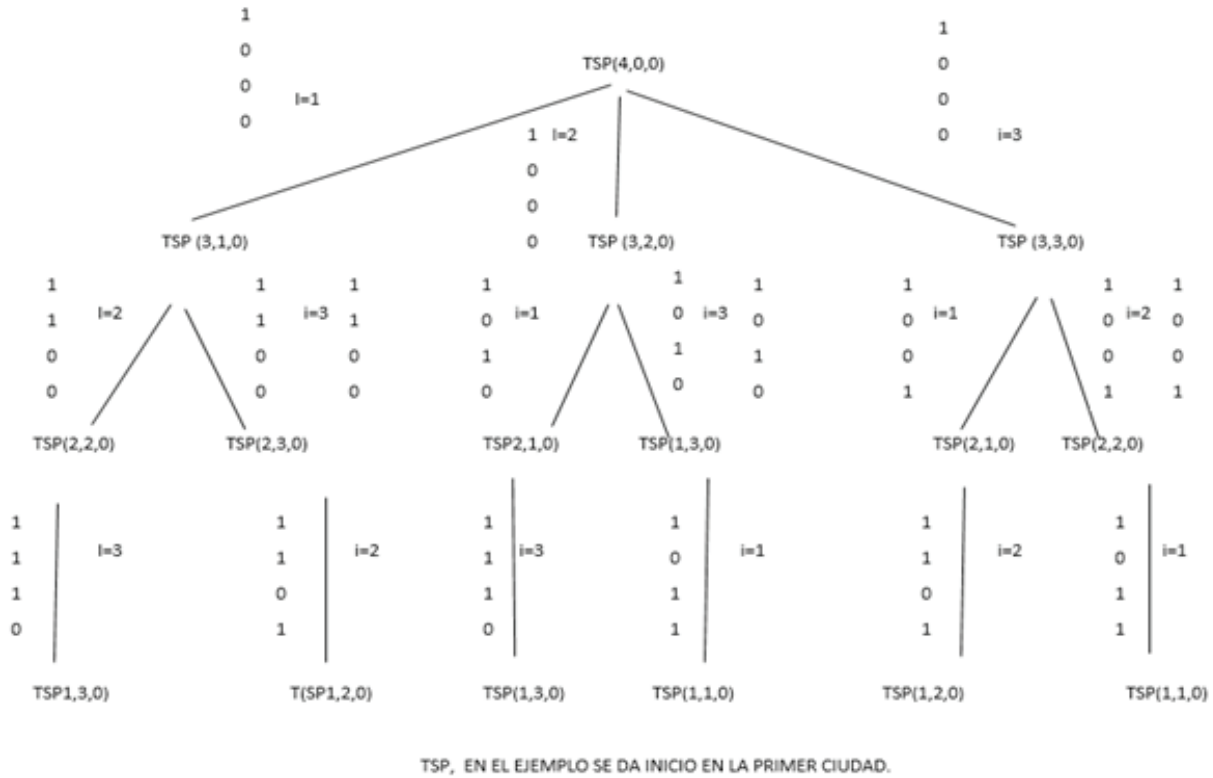
Algoritmo 4.5. TSP

```
#include<stdio.h>
#include<stdlib.h>

void generar(int **,int);
int tsp(int **,int *,int,int,int,int);

main(){
 int **cost,*m,d,o,v,i,j,r;
 printf(" *************** TSP ***************\n");
 printf("\nIntroduce el numero de vertices: ");
 scanf("%d",&v);
 cost=(int **)malloc(sizeof(int *)*v);
 for(i=0;i<v;i++)
 cost[i]=(int *)malloc(sizeof(int)*v);
 m=(int *)malloc(sizeof(int)*v);
 for(i=0;i<v;i++)
 m[i]=0;
 for(i=0;i<v;i++)
 for(j=0;j<v;j++){
```

```
int tsp(int **cost,int *m,int d,int o,int v,int r){
 if(d==1)
 return cost[o][r];
 int dist,dmin=999;
 m[o]=1;
 for(int i=0;i<v;i++)
 if(m[i]==0){
 dist=cost[o][i]+tsp(cost,m,d-1,i,v,r);
 m[i]=0;
 if(dist<dmin){
 dmin=dist;
 }
 }
 return dmin;
 }
```

El programa incluye a un vector *m* que permite indicar la secuencia de ciudades a las que se va a visitar. En la figura 4.6 se aprecia el árbol de recursividad que incluye al referido vector:

Figura 4.6. Uso del vector m



En la función *tsp( )* de la figura 4.6 se colocan el tercero, cuarto y sexto parámetro para mayor simplicidad del árbol.

Para la figura 4.4, una gira óptima tiene una longitud de 35. Una gira de esta longitud puede ser construida si se retiene de cada *g (i, S)* el valor de j que minimiza el lado derecho de (2). Sea *J (i, S)* este valor, entonces *J (1, {2, 3, 4}) = 2.* De esta forma, la gira inicia de 1 a 2. El siguiente punto a visitar se obtiene de *g (2, {3,4}), J (2, {3, 4} = 4*, por lo que la siguiente arista es *<2. 4>*. Lo que falta de la gira es *g (4,{3}), J (4,{3}) = 3*. El recorrido óptimo es 1, 2, 4, 3, 1.

CÁLCULO DE LA COMPLEJIDAD

En el problema del agente viajero, la solución más directa es la de intentar todas las premutaciones de las n ciudades y encontrar el recorrido más barato, dando una complejidad de (n!). Desafortunadamente, con 8 ciudades ya se tienen problemas (ver tabla 4.1). Un problema de permutaciones es mucho más difícil de resolver que un problema donde solo se escojan subconjuntos, ya que existen *n*! permutaciones y 2^n diferentes subconjuntos de n objetos. Con programación dinámica se han podido resolver desde 49 ciudades en 1954 al récord actual de 85 900 ciudades en 2006, el último tomando 136 años de CPU (De los Cobos Silva, Goddard, Gutiérrez Andrade y Martínez Licona, 2010).

Tabla 4.1. O(n!) > O(2^n)

n	2^n	n!
1	2	1
2	4	2
4	16	24
8	256	40320

Utilizando la estrategia de programación dinámica, la complejidad del algoritmo es esta:

- En la primer iteración se calculan los g(j,): n-1, siendo esta la primer iteración.
- Se calculan los *g(j,S)* tales que *1 ≤ S ≤ n-2:*

(n-1) $\left|\begin{matrix} n-2 \\ k \end{matrix}\right|$ k sumas en total.

Por lo tanto,

$$\Theta(n\text{-}1 + \sum_{k=0}^{n-2}(n-1)\ k\left|\begin{matrix} n-2 \\ k \end{matrix}\right|) = \Theta(n^2 2^n).$$

Un algoritmo que procede a encontrar un recorrido óptimo haciendo uso de (1) y (2) requerirá θ ($n^2 2^n$) veces para el cálculo de *g (i, S) con* |S| = *k requiere k – 1* comparaciones para resolver (2). Esto es mejor que la enumeración de todos los n diferentes recorridos para encontrar el mejor recorrido (conocido como fuerza bruta). El inconveniente más grave de esta solución con programación dinámica es el espacio requerido. El espacio necesario es ($n2^n$). Esto es demasiado grande incluso para valores modestos de n.

Programación lineal (linear programming)

INTRODUCCIÓN

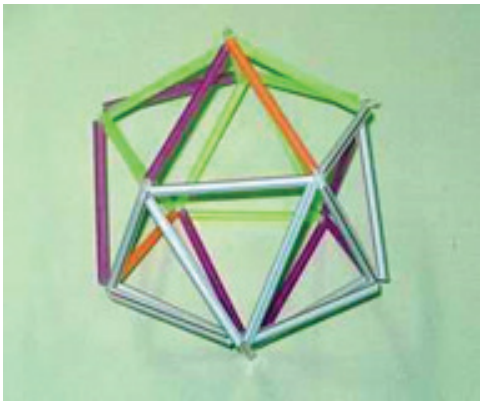
La programación lineal es una pequeña parte de todo un cuerpo matemático que se ha venido consolidando en el pasado siglo XX con el nombre *optimización*. En general, se trata de un conjunto de técnicas matemáticas que intentan obtener el mayor provecho posible de sistemas económicos, sociales, tecnológicos cuyo funcionamiento se puede describir matemáticamente de modo adecuado. Además de la programación lineal, otras disciplinas de este campo son la teoría de juegos, la programación no lineal, la cibernética y la teoría de control. Para los problemas complejos que aborda la optimización, ha sido de vital importancia la interacción, cada vez más fácil y fluida, entre el pensamiento matemático y el ordenador, **y se piensa que los avances futuros en el tratamiento de la complejidad de los sistemas que hay que abordar en la actualidad dependerán más de los progresos matemáticos que de las mejoras tecnológicas de nuestros ordenadores.**

El problema básico de la programación lineal es el de la maximización de una cierta expresión lineal, que se llama *función objetivo*, cuyas variables están sometidas a una serie de restricciones que vienen expresadas por inecuaciones lineales. En la práctica, tanto el número de variables como el de restricciones lineales puede ser de cientos de miles, lo cual hace imprescindible encontrar algoritmos eficaces para la solución del problema e implementables con el ordenador.

Los fundadores de la técnica son George Dantzig (1914-2005), quien publicó el algoritmo Simplex en 1947; John von Neumann, que desarrolló la teoría de la dualidad en el mismo año, y Leonid Kantoróvich, un matemático ruso que utilizó técnicas similares en la economía antes de Dantzig y ganó el Premio Nobel de Economía en 1975.

La idea sobre la que se asienta el método Simplex puede entenderse fácilmente con la siguiente comparación: supóngase que sobre la mesa está el esqueleto, formado por las aristas, de un poliedro complicado convexo como el de la figura 5.1.

Figura 5.1. Esqueleto, formado por las aristas, de un poliedro complicado convexo



Ahora, imagínese que una hormiga que deambula por la mesa intenta subir escalando por las aristas del poliedro al vértice situado en la altura máxima. ¿Qué camino tomará para subir cuanto antes? Comienza a subir por una arista hasta llegar al vértice final de ella. Desde ese vértice tiene varias otras aristas que podría recorrer para seguir escalando. ¿Cuál debe escoger? Parece razonable elegir la que suba más rápidamente, es decir, la de mayor pendiente hasta llegar al próximo vértice. Y desde ese vértice escogerá de nuevo la arista de mayor pendiente hasta llegar a un vértice desde el que no pueda ascender más. Habrá llegado a la cima.

El método Simplex es un algoritmo que indica, paso a paso, un procedimiento para resolver el problema que se propone en la programación lineal. Si bien es cierto que se pueden construir ejemplos en los que el método Simplex resulta ser un algoritmo muy lento (es fácil imaginar un poliedro en el que las aristas de mayor pendiente, por las que la hormiga del ejemplo escoge subir, sean muy cortas y muchas), en la práctica sucede, normalmente, que el método Simplex funciona muy eficientemente y conduce de forma rápida a la solución.

Pero en 1984, Narendra Karmarkar, un matemático de origen indio establecido en Estados Unidos, logró un algoritmo que supera por mucho, en eficiencia, al algoritmo Simplex para el tratamiento de problemas con un gran número de variables y de restricciones. Puestos nuevamente como ejemplo el caso de la hormiga y el poliedro, alcanzar la cima del poliedro aplicando esta modificación supondría que la hormiga podría ascender por el interior del poliedro abandonando las aristas por las que, según el método Simplex, debía subir. Con ello, escogiendo las rutas de ascenso adecuadamente, se podría colocar más rápidamente en la cima. Al principio, el método de Karmarkar fue acogido con cierto escepticismo. Unos años antes, el matemático soviético L. G. Khachian había desarrollado un método, basado en otro llamado *elipsoide*, que si bien teóricamente parecía superior al Simplex, resultó ser menos eficiente desde el punto de vista práctico.

Pero el algoritmo de Karmarkar ha demostrado una eficacia bastante mayor, sobre todo cuando se trabaja con sistemas de un número de variables y de inecuaciones verdaderamente grande. Un problema reciente de programación lineal con 800 000 variables fue resuelto con el algoritmo de Karmarkar tras 10 horas de trabajo del ordenador. Se cree que el problema hubiera necesitado semanas enteras de trabajo de ordenador mediante el método Simplex. Sin embargo, en el tratamiento de sistemas moderadamente grandes, el método Simplex parece, todavía, preferible (Docentes Educación Navarra, 29 de diciembre de 2018).

MÉTODO SIMPLEX

Las metodologías empleadas dentro de la investigación de operaciones están fundamentadas en álgebra lineal, donde los primeros problemas de resolución de sistemas lineales de inecuaciones se remontan a Joseph Fourier (en el año 1826), quien más adelante desarrolló el método de eliminación de Fourier-Motzkin.

La programación lineal se planteó como un problema matemático desarrollado durante la Segunda Guerra Mundial para planificar los gastos y los retornos, a fin de reducir los costos al ejército y aumentar las pérdidas del enemigo. Esta se mantuvo en secreto hasta 1947. En la posguerra, la industria observó las bondades del método para reducir los gastos de sus operaciones diarias, por lo cual originó que se usara en su planificación diaria.

El ejemplo original de Dantzig de la búsqueda de la mejor asignación de 70 personas a 70 puestos de trabajo es un ejemplo de la utilidad de la programación lineal. La potencia de computación necesaria para examinar todas las permutaciones (o fuerza bruta) a fin de seleccionar la mejor asignación es inmensa (¡factorial de 70 o 70!); el número de posibles configuraciones excede al número de partículas en el universo. Sin embargo, toma solo un momento encontrar la solución óptima mediante el planteamiento del problema como programación lineal y la aplicación del algoritmo Simplex. **La teoría de la programación lineal reduce drásticamente el número de posibles soluciones óptimas que deben ser revisadas.**

La programación lineal es el campo de la optimización matemática dedicado a maximizar o minimizar (optimizar) una función lineal, denominada *función objetivo*, de tal modo que las variables de dicha función estén sujetas a una serie de restricciones expresadas mediante un sistema de inecuaciones también lineales. Los métodos más recurridos para resolver problemas de programación lineal son algoritmos de pivote, en particular los algoritmos simplex.

Esta función objetivo representa cada uno de los elementos (recursos) que se desean maximizar o minimizar, la cual puede ser expresada de la siguiente manera:

$$Z = \sum_{i=1}^{n} c_i \, x_i$$

La función objetivo (Z) puede quedar expresada de acuerdo con lo que se desee realizar.

$$Min\ Z = c_1\, x_1 + c_2\, x_2 + \cdots + c_n\, x_n$$
$$Max\ Z = c_1\, x_1 + c_2\, x_2 + \cdots + c_n x_n$$

donde

C_1 son contantes conocidas (costo por unidad de la variable x_j).

X_1 representan cada uno de los elementos (recursos) que se desean satisfacer.

RESTRICCIONES

Constituyen las limitaciones de los recursos. Estas pueden ser representadas como igualdades y desigualdades dentro del problema de programación lineal. Las restricciones están dadas de la siguiente forma:

$$\sum_{j=1}^{n} a_{ij}\, x_j \leq b_j$$

donde

a_{ij}, b_j son constantes conocidas.

x_n representan cada uno de los elementos que se desean satisfacer.

$\leq$ es la restricción produciendo un subespacio específico.

Las restricciones también pueden ser representadas en forma matricial de la siguiente manera:

$$\left[\begin{array}{ccc|c} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{1m} & \cdots & a_{mn} & b_m \end{array}\right]$$

donde

a_{mn} son contantes conocidas que representan la necesidad a satisfacer.

b_m representa el límite del recurso que se tiene para satisfacer la necesidad.

Dentro de las restricciones se deben cumplir dos elementos básicos:

Estructurales

Reflejan factores como la limitación de recursos y otras condiciones que impone la situación del problema.

No negatividad

Garantizan que ninguna variable sea negativa.

RESTRICCIONES DE ACUERDO CON EL PROBLEMA DE PROGRAMACIÓN

Las restricciones se deben establecer de acuerdo con lo que se busca satisfacer en el problema que se tiene en ese momento. Si buscamos maximizar los recursos dentro de un problema de programación lineal, el conjunto de restricciones debe ser de signo menor o igual; en caso de ser mayor o igual se debe multiplicar por -1 para cambiarla por la restricción adecuada.

Si buscamos minimizar los recursos, los signos de las restricciones deben ser mayor o igual en todos los casos; si una no está así, se debe multiplicar por -1 para establecerla de la manera adecuada.

De acuerdo con lo anterior, tenemos que una restricción representa una frontera, que es un concepto geométrico, donde nosotros podemos obtener la ecuación de la frontera de restricción de cualquier restricción al sustituir los signos ≤, ≥ por el signo igual.

En consecuencia, la forma de la ecuación de frontera de restricción es esta:

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i.$$

Las únicas que no van a obedecer lo anterior son las restricciones de no negatividad:

$$x_j \geq 0.$$

La solución óptima de cualquier problema de programación lineal se encuentra sobre la frontera de la región factible, siendo una propiedad general.

REGIONES

La región factible es un espacio convexo donde se satisfacen todas y cada una de las restricciones a las cuales está sujeto nuestro problema de programación lineal.

Las restricciones en los problemas de programación lineal, debido a que son desigualdades lineales, geométricamente representan hiperespacios, que son conjuntos convexos.

Debido a que la intersección de conjuntos convexos es convexa, la región factible resulta ser un conjunto convexo y a su vez tiene la forma de un poliedro, la cual es una figura geométrica de caras planas.

Si se tiene una región factible, que es de forma de poliedro y convexa, se puede garantizar lo siguiente:

- Se tiene el óptimo y este ocurre en al menos un punto extremo.

Dentro de la región factible tenemos las siguientes regiones:

- Región factible acotada.
- Región factible acotada con múltiples soluciones.
- Región factible no acotada.

REGIÓN FACTIBLE ACOTADA

Existe este tipo de región cuando solo se tiene una única solución que satisface el problema de programación lineal. Por ejemplo, en una urbanización se van a construir casas de dos tipos: A y B. La empresa constructora dispone para ello de un máximo de 1800 millones de pesos, y el coste de cada tipo de casa es de 30 millones y 20 millones, respectivamente. El Ayuntamiento exige que el número total de casas no sea superior a 80.

Sabiendo que el beneficio obtenido por la venta de una casa de tipo A es de 4 millones y por una de tipo B de 3 millones, ¿cuántas casas deben construirse de cada tipo para obtener el máximo beneficio?

Variables:

x = casas tipo A.

y = casas tipo B.

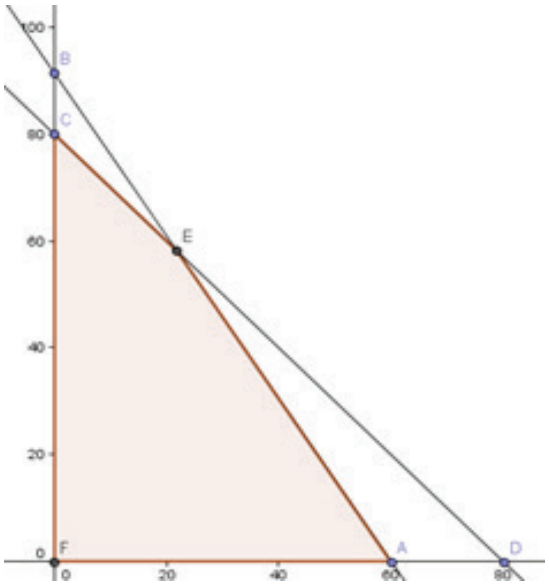
Función objetivo:

Maximizar $Z = f(x, y) = 4x + 3y$

Sujeto a:

- El coste total $30_x + 20_y \leq 1800$.
- El Ayuntamiento impone $x + y$ 80.
- No negatividad: $x \geq 0$, $y \geq 0$.

Figura 5.2. Región factible acotada



En la figura 5.2, los vértices extremos que producen un espacio factible son V_A, V_C y V_E y V_F. Si se hallan los valores de la función objetivo en cada uno de los vértices,

$$V_F(0,0) = 0$$
$$V_A(60,0) = 240$$
$$V_E(21.7,58.23) = 261.4$$
$$V_C(0,80) = 240$$

La solución es única y corresponde al vértice para el que la función objetivo toma el valor máximo.

En este caso es el Por tanto se deben construir 21 casas de tipo A y 58 de tipo B con un coste de 261.4 millones de pesos. Se observará que el óptimo tiene valores fraccionarios, es decir, se tienen que construir 21.7 casas del tipo A, aunque construir 0.7 casas suena ilógico. Por ello, construir 21 y 58 casas no garantiza plenamente que sea el óptimo, de esta forma se creó la programación entera.

REGIÓN FACTIBLE ACOTADA CON MÚLTIPLES SOLUCIONES

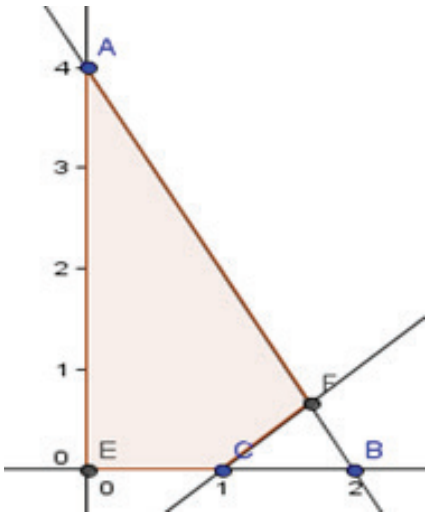
Esta situación surge cuando se tiene más de una solución dentro de la región factible que satisface el conjunto de restricciones a las cuales está sujeto el problema de programación lineal, donde al menos dos deben de ser soluciones adyacentes. Por ejemplo:

Maximizar la función Z = 4x + 2y

Sujeta a:

$$2x + y \leq 4$$
$$x - y \leq 1$$
$$x \geq 0$$
$$y \geq 0$$

Figura 5.3. Región factible con múltiples soluciones



Observando la figura 5.3, los valores de la función objetivo en cada uno de los vértices son:

$$V_E(0,0) = 0$$
$$V_C(1,0) = 4(1) + 2(0) = 4$$
$$V_F(5/3, 2/3) = 4\left(\frac{5}{3}\right) + 2\left(\frac{2}{3}\right) = 8$$
$$V_A(0,4) = 4(0) + 2(4) = 8$$

La función objetivo alcanza el valor máximo en los vértices A y F, por tanto, todos los puntos del segmento AF son óptimos (tiene una infinidad de soluciones).

REGIÓN FACTIBLE NO ACOTADA

Este tipo de región se tiene cuando no existe un límite para la función objetivo. Por ejemplo:

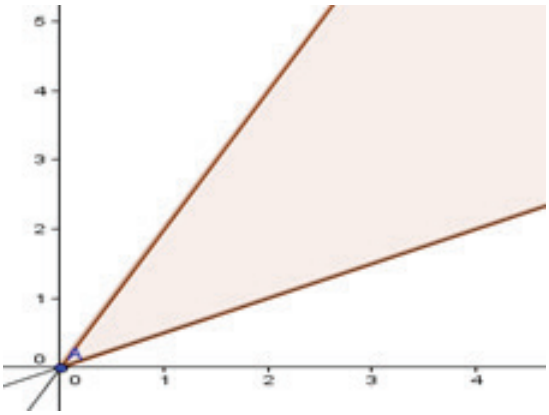
Maximizar la función Z = x + y

Sujeta a:

$2x - y \geq 0$

$x/2 + 7y \geq 0$

Figura 5.4. Región factible no acotada



Observando la figura 5.4, la función crece indefinidamente para valores crecientes de x y y. En este caso no existe un valor extremo para la función objetivo, por lo que puede decirse que el problema carece de solución (el valor de $Z \rightarrow \infty$).

REGIÓN NO FACTIBLE

Cuando se presenta esta región lo que nos indica es que no existe solución para el problema de programación lineal que se esté trabajando, lo cual suele suceder cuando la región es vacía.

Por ejemplo:

Maximizar la función $Z = 3x + 8y$

Sujeta a:

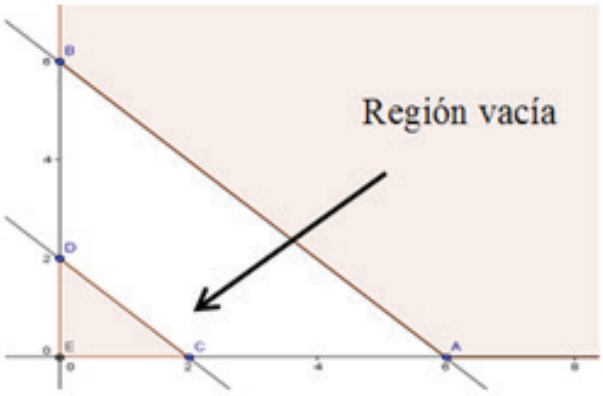
$X + y \geq 6$

$x + y \leq 2$

$x \geq 0$

$y \geq 0$

Figura 5.5. Región no factible



Observando la figura 5.5, el conjunto de soluciones del sistema de desigualdades no determina ninguna región factible, ya que se tienen dos regiones, lo cual solo satisface a alguna restricción sin poder formar una región para el conjunto de restricciones. Este tipo de problemas carece de solución.

MÉTODO DE GAUSS

Como ya se comentó anteriormente, el algoritmo Simplex se sustenta en el método de Gauss, en donde cada cambio de base se localiza la intersección de una recta, un plano o un hiperplano con otra recta, plano o hiperplano.

Para poder entender mejor el conjunto de intersecciones de Ax = b, es necesario hablar acerca de las variables básicas. Una variable básica genera el espacio. La forma más sencilla de ver las variables básicas es la matriz *identidad*, que siempre formará una base.

Cambio de base

Para poder entender mejor el conjunto de soluciones de Ax = b, es necesario hablar acerca de las variables básicas (VB) y las no básicas (VNB). En un sistema de n ecuaciones con m incógnitas se puede añadir un conjunto de n variables en las cuales forman una base ortonormal. En este caso, si se tiene Ax = b, se pueden añadir las variables de la siguiente forma: Ax + Iy = b, donde el vector y forma una base ortonormal.

Por ejemplo, se tiene el siguiente sistema de ecuaciones:

$$\begin{aligned} x + 3w + 5z &= 13 \\ 3x + 8w - z &= 12 \\ 12x - 34w + 2z &= -6 \end{aligned}$$

Se puede incluir al sistema las siguientes variables:

$$\begin{aligned} x + 3w + 5z + y_1 + 0 + 0 &= 13 \\ 3x + 8w - z + 0 + y_2 + 0 &= 12 \\ 12x - 34w + 2z + 0 + 0 + y_3 &= -6 \end{aligned}$$

Esto produce la siguiente matriz:

$$\begin{array}{ccccccc} 1 & 3 & 5 & 1 & 0 & 0 & 13 \\ 3 & 8 & -1 & 0 & 1 & 0 & 12 \\ 12 & 34 & 2 & 0 & 0 & 1 & -6 \end{array}$$

Otro ejemplo puede ser este:

$$\begin{aligned} x + 3w + 5z &= 13 \\ 3x + 8w - z &= 12 \\ 12x - 34w + 2z &= -6 \\ 10x + 5w - 4z &= 4 \end{aligned}$$

Véase que en este sistema de tres variables, para formar la base, solo se requieren tres ecuaciones, por lo que sobra una. Se pueden incluir al sistema las siguientes variables:

$$\begin{aligned} x + 3w + 5z + y_1 + 0 + 0 + 0 &= 13 \\ 3x + 8w - z + 0 + y_2 + 0 + 0 &= 12 \\ 12x - 34w + 2z + 0 + 0 + y_3 + 0 &= -6 \\ 10x + 5w - 4z + 0 + 0 + 0 + y_4 &= 4 \end{aligned}$$

donde y_1, y_2, y_3, y_4 forman una base produciendo la siguiente matriz:

$$\begin{array}{ccccccc} 1 & 3 & 5 & 1 & 0 & 0 & 0 & 13 \\ 3 & 8 & -1 & 0 & 1 & 0 & 0 & 12 \\ 12 & 34 & 2 & 0 & 0 & 1 & 0 & -6 \\ 10 & 5 & -4 & 0 & 0 & 0 & 1 & 4 \end{array}$$

La última matriz puede representarse de la siguiente forma:

	x	w	z	y_1	y_2	y_3	y_4	
y_1	1	3	5	1	0	0	0	13
y_2	3	8	-1	0	1	0	0	12
y_3	12	34	2	0	0	1	0	-6
y_4	10	5	-4	0	0	0	1	4

Por lo tanto, para esta base en particular: $y_1 = 13$, $y_2 = 12$, $y_3 = -6$, $y_4 = 4$. $x = 0$, $w = 0$, $z = 0$ no forman la base (ya que es un sistema de 3 variables y 4 ecuaciones).

Ejemplo sobre cambio de base

Ahora se tiene un ejemplo en el cual se observa cómo se mueven los puntos de intersección con cada cambio de base. Obsérvese el siguiente sistema de ecuaciones:

$$2x_1 + x_2 = 1$$
$$x_1 - 2x_2 = 2$$

Incluyendo una base (y_1, y_2), se tiene:

	x_1	x_2	B	y_1	y_2
y_1	2	1	1	1	0
y_2	1	-2	2	0	1

La matriz se puede expresar como A b I, donde

$x = \binom{0}{0} y = \binom{1}{2}$, siendo el origen (ver figura 5.6).

Ahora, si

$$R1 \rightarrow \frac{1}{2} R1, R2 \rightarrow R$$

se tiene el siguiente sistema equivalente, donde entra a la base la variable x_1 y sale de la base y_1, lo cual produce la siguiente matriz:

	y_1	x_2	B	y_1	y_2
x_1	1	½	½	½	0
y_2	0	5/2	-3/2	1/2	-1

En la figura 5.6, el punto A es x = y = $\binom{1/2}{0}, y = \binom{0}{-3/2}$

Si de la matriz original se realiza la siguiente operación: $R2 \rightarrow R2 - \frac{1}{2}R1,$

,se tiene el siguiente sistema equivalente, donde entra a la base la variable x_2 y sale de la base y_1, lo que produce el siguiente tablero:

	y_2	x_2	B	y_1	y_2
y_1	0	-5/2	3/2	-1/2	1
x_1	1	-2	2	0	1

En la figura 5.6, el punto B es x = , y = $\binom{2}{0}, y = \binom{3/2}{0}$

Si de la matriz original se realiza la siguiente operación: $R2 \rightarrow R1 + \frac{1}{2}R2,$

entonces se tiene el siguiente sistema equivalente, donde entra a la base la variable x_2 y sale de la base y_1 produciendo:

	x_1	y_1	B	y_1	y_2
x_2	2	1	1	1	0
y_2	5/2	0	2	1	1/2

En la figura 5.6, el punto C es x= y = $\binom{0}{1}, y = \binom{0}{2}$

Ahora, si de la matriz original se realizan las siguientes operaciones:

$$R2 \rightarrow -\frac{1}{2}R2,$$
$$R1 \rightarrow R2 - R1,$$

entonces, se tiene el siguiente sistema equivalente:

	x_1	y_2	B	y_1	y_2
y_1	-3/2	0	-2	-1	-1/2
x_2	-1/2	1	-1	0	-1/2

En la figura 5.6, el punto D es x = , y = $\binom{0}{-1}, y = \binom{-2}{0}.$

Tomando la primera matriz transformada y realizando las siguientes operaciones:

$$R2 \rightarrow \frac{2}{5}R2, R1 \rightarrow R1 - \frac{1}{2}R2,$$

entonces, se tiene el siguiente sistema equivalente:

	y_1	y_2	B	x_1	x_2
x_1	1	0	8/10	2/5	1/5
x_2	0	1	-3/5	1/5	-2/5

En este caso, son variables básicas x_1 y x_2. Y son variables no básicas tanto y_1 como y_2.

En la figura 5.6, el punto E es $x = \begin{pmatrix} 8/10 \\ -3/5 \end{pmatrix}, y = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Este punto es la intersección de las dos rectas o la solución del sistema de ecuaciones. La última matriz se puede representar de este modo:

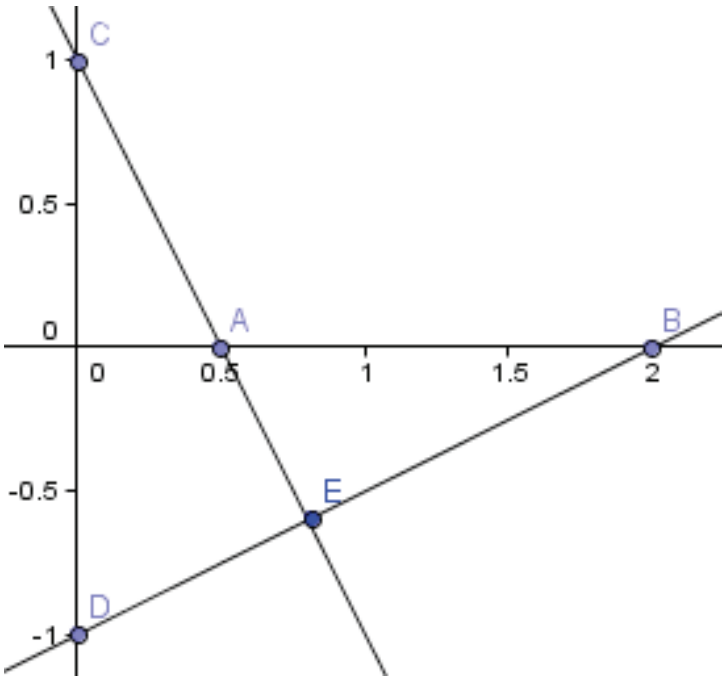
$$\mathbf{I X A^{-1}}$$

Se puede observar que la matriz original es $A = \begin{pmatrix} 2 & 1 \\ 1 & -2 \end{pmatrix}$

y la matriz inversa es $A^{-1} = \begin{pmatrix} 2/5 & 1/5 \\ 1/5 & -2/5 \end{pmatrix}$.

Por lo tanto, $AA^{-1} = I$.

Figura 5.6. Gráfica que muestra el desplazamiento de los puntos extremos



De acuerdo con lo anterior, se puede apreciar, en general, que cuando se establece un sistema de m-ecuaciones y n-incógnitas con , teniendo n-variables y forman la base n-ecuaciones, las ecuaciones restantes son no básicas.

Si el sistema que se tiene es de m-ecuaciones con n-incógnitas, se establece que el número de soluciones básicas está dado por el número de combinaciones de n-variables tomadas de m en m, es decir:

$$N = \frac{n!}{m!\,(n-m)!}$$

Del ejemplo anterior, encontramos los puntos de la a a la e, teniendo las siguientes bases:

$$\binom{y_1}{y_2}\binom{y_1}{x_1}\binom{x_1}{y_2}\binom{x_2}{y_2}\binom{y_1}{x_2}\binom{x_1}{x_2},$$

siendo 4 incógnitas (*x1, x2, y1, y2*) y 2 ecuaciones, por lo que:

$$N = \frac{4!}{2!\,(4-2)!} = \frac{4!}{2! * 2!} = 6$$

Con las soluciones básicas y los conceptos del conjunto convexo se pueden resumir las siguientes tres características:

- Las restricciones factibles forman un conjunto convexo cuyos extremos son soluciones factibles.
- Si las restricciones definen una solución factible, existe cuando menos una solución básica factible.
- Si la función objetivo tiene un máximo finito, entonces al menos una de las soluciones óptimas es una solución básica factible.

Por esto, se puede concluir lo siguiente:

- La solución existe y es única.
- Existe un número infinito de soluciones y están acotadas.
- Existe un número infinito de soluciones que no están acotadas.
- No existe solución.

Para los tres primeros puntos, se dice que el sistema es consistente, mientras que en el último caso se dice que el sistema es inconsistente.

En ocasiones, trabajar todas las posibles combinaciones es complicado; por ejemplo, teniendo un sistema de 4 ecuaciones con 8 incógnitas, da el siguiente valor:

$$N = \frac{8!}{4!\,(8-4)!} = 70\ soluciones\ básicas.$$

Por esto, el algoritmo más conocido para localizar el punto óptimo es el método Simplex.

ALGORITMO DEL MÉTODO SIMPLEX

El método Simplex es un procedimiento algebraico cuyos conceptos fundamentales son geométricos. Se basa en el hecho de que el óptimo de un problema de programación lineal siempre deberá ocurrir en un punto extremo de acuerdo con la linealidad y convexidad de las restricciones, por lo cual solo basta con analizar los puntos extremos para conocer el punto que produce el óptimo.

El método se inicia en un punto extremo factible y se mueve a otro punto extremo adyacente que incrementa la función objetivo. Cuando ya no es posible movernos a otro extremo adyacente mejorando la función objetivo, se dice que ya se ha alcanzado el óptimo.

FORMA ESTÁNDAR DEL PROBLEMA DE PROGRAMACIÓN LINEAL

$$Maximizar\ Z = CX$$

Sujeta a:

$$Ax \leq b$$
$$x \geq 0$$

donde

$x = x_1, x_2, \ldots, x_n$, *siendo las variables del problema.* $c = c_1, c_2, \ldots, c_3$, *siendo las constantes conocidas del problema.* $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & a_{23} & \ddots & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \ldots & a_{mn} & b_n \end{bmatrix}$ *es la matriz ampliada de las restricciones.*

Si el problema no reúne estas condiciones, se debe manipular de modo que se pueda expresar mediante alguna de las siguientes transformaciones:

a) Si se tiene una restricción del tipo:

$$a_{i1}x_1 + \cdots + a_{n1}x_n \geq b_i,$$

para transformarla a la forma ≤ se debe multiplicar por -1, quedando de la siguiente forma:

- $a_{i1}x_1 - \cdots - a_{n1}x_n \leq -b_i.$

b) Si se tiene una restricción de del tipo:

$$a_{i1}x_1 + \cdots + a_{n1}x_n = b_i,$$

,

se puede sustituir por las siguientes dos restricciones:

$$a_{i1}x_1 + \cdots + a_{n1}x_n \geq b_i,$$

$$a_{i1}x_1 + \cdots + a_{n1}x_n \geq b_i,$$

c) Si se tiene una variable que pueda tomar valores positivos y negativos, es decir, que sea irrestricta, se puede redefinir de la siguiente forma:

$$x_i = x_i' - x_i''$$
$$donde$$
$$x'_i,\ x_i'' \geq 0$$

y sustituir x_1¨ - x_1" en cada aparición de

MÉTODO DE LAS DOS FASES

Como su nombre lo indica, el método resuelve la programación lineal en dos fases:

- Fase uno: Trata de determinarse una solución básica factible de inicio.
- Fase dos: Si se cumple la fase uno, esta fase se ocupa de resolver el problema original.

EL TABLEAU DE TUCKER PARA EL MÉTODO SIMPLEX

El problema de programación lineal escrito en forma estándar es de siguiente:

$$\begin{aligned} &\text{Maximizar} \quad Z=cx \\ &\text{Sujeta a:} \\ &\qquad Ax \leq b \\ &\qquad x \geq 0 \end{aligned}$$

Se puede reescribir de la siguiente forma:

$$\begin{aligned} &\text{Maximizar} \quad Z=cx \\ &\text{Sujeta a:} \\ &\qquad Y = A(-x) + b \\ &\qquad x \geq 0 \end{aligned}$$

Las restricciones son ahora un conjunto de relaciones lineales, con la restricción adicional de que tanto las variables independientes (X) como las dependientes (Y) son no-negativas.

Este problema se puede resumir en una tabla, que en la terminología del método Simplex se llama *tableau de Tucker,* de la siguiente forma:

		Variables independientes				
		$-X_1$	$-X_2$	...	$-X_n$	1
Variables dependientes	$Y_1 =$	a_{11}	a_{12}	...	a_{1m}	b_1
	$Y_2 =$	a_{21}	a_{22}	...	a_{2m}	b_2
	⋮	⋮	⋮	⋱	⋮	⋮
	$Y_m =$	a_{m1}	a_{m2}	...	a_{nm}	b_m
	$Z =$	$-c_1$	$-c_2$	...	$-c_n$	0

El *tableau* contiene relaciones lineales que representan a las restricciones y a la función objetivo. La hilera i de la tabla que se leerá como $y_i = a_{i1}(-x_1) + \cdots + a_{in}(-x_n) + b_i$ que es igual a multiplicar la hilera *i* de la matriz A por el vector (-*X*) y sumarle b_i, es decir, $y_i = a_{ij}(-x_j) + b_i$.

El *tableau* representará el punto X en el espacio Rn, que se obtiene al hacer cero las variables independientes. En el *tableau* inicial, generalmente, este punto es $x_1 = 0, x_2 = 0, \ldots, x_n = 0$ y el valor de la función objetivo que se obtiene en ese punto es cero, que es el elemento que aparece en la última celda de la última columna del *tableau*. Por ejemplo:

Sujeta a:

$$Max = 2x_1 - x_2$$

$$\begin{array}{rcrcl} -x_1 & + & 2x_2 & \leq & 2 \\ x_1 & - & x_2 & \leq & 1 \\ x_1 & + & x_2 & \leq & 2 \end{array}$$
$$x_1, x_2 \geq 0$$

Por lo tanto, si se reescribe de acuerdo a Y= A(-x) + b el siguiente sistema de ecuaciones lineales:

$$\begin{aligned} y_1 &= x_1 - 2x_2 + 2 \geq 0 \\ y_2 &= -x_1 + x_2 + 1 \geq 0 \\ y_3 &= -x_1 - x_2 + 2 \geq 0 \\ Z &= 2x_1 - x_2 \geq 0, \end{aligned}$$

se tendría el siguiente tableau:

	$-x_1$	$-x_2$	1
y_1	-1	2	2
y_2	1	-1	1
y_3	1	1	2
z	-2	1	0

ALGORITMO DE LAS DOS FASES BAJO TUCKER

Fase I

Se emplea cuando el origen no es factible. El método consiste en introducir una variable adicional *p* en la parte no básica tratando de hacer que la restricción que viola

$$y_i = \sum a_{ij}\left(-x_j\right) + b_i \leq 0$$

se transforme a

$$y_i = \sum a_{ij}\left(-x_j\right) + b_i + \rho \geq 0,$$

siendo el espacio que se trabaja en $\mathbb{R}^{n+1}$.

Se adiciona la función objetivo siendo el coeficiente de *x1 = 0* y el coeficiente de *p* = 1. La variable básica para elegir será aquella donde ocurra la mayor infactibilidad, por lo cual el elemento pivote será aquel con la columna *p* y cuya hilera sea más negativa en b_i, por lo que *p* pasará a ser básica.

Se maximiza *Z´* con el criterio de la segunda fase. Si a *p* se localiza como no básica al encontrar el óptimo de *z*', se ha logrado colocar la función objetivo en un punto extremo del espacio convexo.

Se elimina la columna del *p tableu* y a Z', siendo Z en un punto factible, de aquí se aplicará la fase II hasta llegar al punto deseado.

Si al optimizar *Z´* no se localiza a *p* como no básica, el sistema original no forma un espacio convexo, por lo que no existe solución.

Fase II

A. Para mejorar el valor de la función objetivo se recomienda tomar como columna el coeficiente más negativo de la última hilera (columna s).

B. La hilera pivote será:

$$\propto = min\left\{ {}^{b_i}/_{a_{is}} \middle| a_{is} > 0 \right\}$$

Localizando la hilera pivote, se realizan las siguientes operaciones de acuerdo con el pivoteo de Jordan:

- El pivote se transforma a $a'_{rs} = 1/a_{rs}$.
- Elementos sobre la hilera pivote (hilera r).

$$d_{rj} = \frac{a_{rj}}{a_{rs}}, s \neq j$$

- Elementos sobre la columna pivote (columna s):

$$d_{is} = -\frac{a_{is}}{a_{rs}}, i \neq r$$

- Resto de los elementos:

$$d_{ij} = a_{ij} - \frac{a_{is}a_{rj}}{a_{rs}}, j \neq s\ ;\ i \neq r$$

Para el ejemplo anterior se tiene:

	$-x_1$	$-x_2$	1
y_1	-1	2	2
y_2	1	-1	1
y_3	1	1	2
z	-2	1	0

Ahora se procede a elegir el pivote, iniciando por la última hilera y seleccionando el número más negativo, y después dividiendo cada coeficiente por el valor de la última columna seleccionando el mínimo valor obtenido siempre y cuando este sea mayor que cero; entonces se tendría lo siguiente:

	$-x_1$	$-x_2$	1
y_1	-1	2	2
y_2	1	-1	1
y_3	1	1	2
Z	-2	1	0

Valor mínimo obtenido

$\alpha = \frac{2}{-1} = -2.\ A_{i,s} < 0$

$\alpha = \frac{1}{1} = 1$

$\alpha = \frac{2}{1} = 2$

El pivote $a'_{rs} = 1$,

Ahora se procede a hacer cada una de las operaciones indicadas de acuerdo con el pivoteo de Jordan y haciendo el cambio de base correspondiente:

	$-y_2$	$-x_2$	1
y_1	$-\left(-\frac{1}{1}\right)$	$2-\frac{(-1)(-1)}{1}$	$2-\frac{(-1)(1)}{1}$
x_1	1	$-1/1$	$1/1$
y_3	$-\frac{1}{1}$	$1-\frac{(1)(-1)}{1}$	$2-\frac{(1)(1)}{1}$
z	$-\frac{-2}{1}$	$1-\frac{(-2)(-1)}{1}$	$0-\frac{(-2)(1)}{1}$

De acuerdo con las operaciones realizadas, se obtiene el siguiente *tableau*:

	$-y_2$	$-x_2$	1
y_1	1	1	3
x_1	1	-1	1
y_3	-1	2	1
z	2	-1	2

Ahora se vuelve a seleccionar el nuevo pivote del *tableau* obtenido, recordando que *a* ≥ 0.

	$-y_2$	$-x_2$	1	Valor mínimo obtenido
y_1	1	1	3	$\alpha = \frac{3}{1} = 3$
x_1	1	-1	1	$\alpha = \frac{1}{-1} = -1$
y_3	-1	2	1	$\alpha = \frac{1}{2} = 0.5$
z	2	-1	2	

Al realizar cada una de las operaciones anteriormente mencionadas, se consigue el siguiente *tableu*:

	$-y_2$	$-y_3$	1
y_1	3/2	-1/2	5/2
x_1	½	½	3/2
x_2	-1/2	½	½
z	3/2	½	5/2

Por lo tanto, el óptimo se encuentra en X_1 = 3/2, X_2= ½, siendo el resultado de la función objetivo Z = 5/2.

Otro ejemplo: $Max\ Z = x_1 + x_2$

Sujeta a:

$$x_1 + 2x_2 \geq 2$$
$$3x_1 + x_2 \geq 3$$
$$2x_1 + 3x_2 \leq 6$$

Como se puede ver, el problema no está escrito en forma cónica de acuerdo con el *tableau* de Tucker, ya que las restricciones son $A_x \geq b$, por lo cual las ecuaciones se multiplican por -1 para tener las ecuaciones de la siguiente forma $A_x \leq -b$, quedando así:

- $-x_1 - 2x_2 \leq -2$
  $-3x_1 - x_2 \leq -3$

Ahora se reescriben las restricciones de acuerdo con Y= A(-x) + b, quedando de la siguiente forma:

$$y_1 = x_1 + 2x_2 - 2 \geq 0$$
$$y_2 = 3x_1 + x_2 - 3 \geq 0$$
$$y_1 = -2x_1 - 3x_2 + 6 \geq 0$$

Realizado lo anterior, se prosigue a colocar el sistema de ecuaciones en el *tableau* de Tucker quedando de la siguiente forma:

	$-x_1$	$-x_2$	1
y_1	-1	-2	-2
y_2	-3	-1	-3
y_3	2	3	6
z	-1	-1	0

Como se aprecia en el último *tableau*, se tiene en b valores negativos, por lo cual no se puede maximizar directamente con la fase II; de esta forma se procede a realizar la fase I, agregando a *p y Z´* y dentro del *tableau*, colocando el valor de -1 sobre la columna de *p*, siempre y cuando *b* sea un valor negativo y ceros cuando son positivos, en Z' el valor de 1 sobre la columna *p* y en las demás columnas colocamos ceros, quedando de la siguiente forma:

	$-x_1$	$-x_2$	ρ	1
y_1	-1	-2	-1	-2
y_2	-3	-1	-1	-3
y_3	2	3	0	6
Z	-1	-1	0	0
z'	0	0	1	0

A continuación, el primer pivote se selecciona sobre la columna de *p*, el cual debe tener en su hilera en *b* el valor más negativo, siendo el pivote como se muestra a continuación:

	$-x_1$	$-x_2$	ρ	1
y_1	-1	-2	-1	-2
y_2	-3	-1	**-1**	-3
y_3	2	3	0	6
z	-1	-1	0	0
z'	0	0	1	0

Realizando las operaciones según el pivoteo de Jordan, se consigue el siguiente *tableau*:

	$-x_1$	$-x_2$	$-y_2$	1
y_1	2	-1	1	1
ρ	3	1	-1	3
y_3	2	3	0	6
z	-1	-1	0	0
z'	-3	-1	-1	-3

Ahora se procede a realizar el segundo pivote seleccionando la columna en Z' más negativa y buscando a *a* mínima, como se muestra a continuación:

	$-x_1$	$-x_2$	$-y_2$	1	Valor mínimo
y_1	2	-1	-1	1	$\alpha = \frac{1}{2} = 0.5$
ρ	3	1	-1	3	$\alpha = \frac{3}{3} = 1$
y_3	2	3	0	6	$\alpha = \frac{6}{2} = 3$
z	-1	-1	0	0	
z'	-3	-1	1	-3	

Luego se realizan las operaciones indicadas y se obtiene el siguiente *tableau*:

	$-y_1$	$-x_2$	$-y_2$	1
x_1	½	-1/2	-1/2	½
ρ	-3/2	5/2	½	3/2
y_3	-1	4	1	5
z	½	-3/2	-1/2	½
z'	3/2	-5/2	-1/2	0

Repitiendo de nuevo el método Simplex, se tiene el siguiente *tableau*:

	$-y_1$	$-\rho$	$-y_2$	1
x_1	1/5	1/5	-4/10	8/10
x_2	-3/5	2/5	1/5	3/5
y_3	1.4	-1.6	0.2	2.6
z	-0.4	0.6	-1/5	1.4
z'	0	1	0	3

En este momento termina la fase, ya que en *Z'* se tienen valores positivos, aunque aún se debe verificar si se puede pasar a la segunda fase; para eso, se tienen los siguientes dos casos:

- Si a se localiza como no básica al encontrar el óptimo de Z', se elimina la columna del *p tableau* y a Z', siendo que a partir de este nuevo *tableau* se aplicará la fase II hasta llegar al punto deseado.
- Si al optimizar Z´ no se localiza a *p* como no básica, el sistema original no existe solución.

De acuerdo con lo anterior, sí se puede proceder a la fase II, por lo que el *tableau* resultante para esta es el que se muestra a continuación:

	$-y_1$	$-y_2$	1
x_1	1/5	-4/10	8/10
x_2	-3/5	1/5	3/5
y_3	1.4	0.2	2.6
z	-0.4	-1/5	1.4

Ahora se realizan las operaciones de manera normal, iniciando por la elección del primer pivote en la fase II.

	$-y_1$	$-y_2$	1	Valor mínimo
x_1	1/5	-4/10	8/10	No se realiza por ser negativo
x_2	-3/5	1/5	3/5	$\alpha = \frac{\frac{3}{5}}{\frac{1}{5}} = \frac{3}{5} = 0.6$
y_3	1.4	0.2	2.6	$\alpha = \frac{2.6}{0.2} = 8$
Z	-0.4	-1/5	1.4	

A partir de aquí, en adelante solo se muestran los *tableaus* resultantes:

	$-y_1$	$-y_3$	1
x_1	-0.14	-0.43	0.43
x_2	0.43	0.29	1.71
y_2	0.71	0.14	1.86
z	0.29	-0.14	2.14

	$-y_1$	$-x_2$	1
x_1	0.5	1.5	3
y_3	1.5	3.5	6
y_2	0.5	-0.5	1
Z	0.5	0.5	3

Por lo tanto, $X_1 = 3$ y $Z = 3$, siendo este el punto óptimo del sistema de ecuaciones (Bueno de Arjona, 1987). La programación del algoritmo de Tucker se desarrolló en el lenguaje de programación Java y se localiza en el anexo A.

ANÁLISIS DE COMPLEJIDAD

Teniendo a

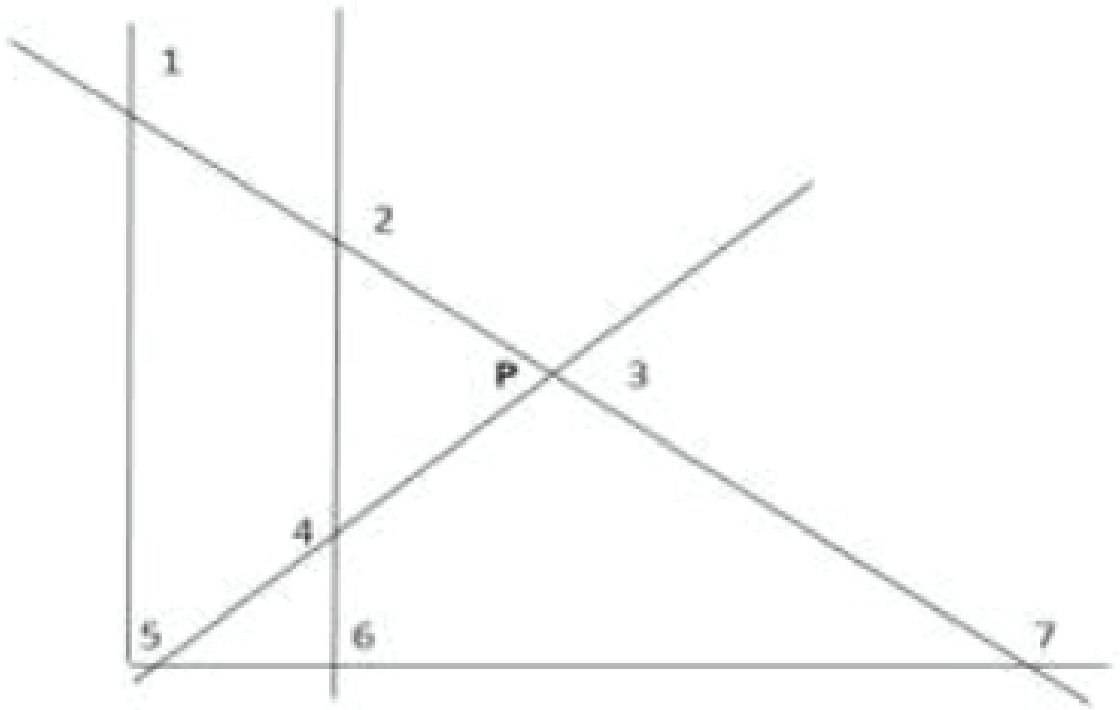
$$\text{Max } c^T x$$

$$\text{Tal que } Ax \leq 0$$

$$X \geq 0$$

con *n* incógnitas y m inecuaciones, el algoritmo Simplex es iterativo; en este va recorriendo cada uno de los vértices donde se logran satisfacer las *m* inecuaciones. Iniciemos computando el tiempo que toma realizar una iteración suponiendo que el vértice actual es *u*. Por definición, es un punto donde se satisfacen las *m* inecuaciones, por lo que *u* tiene por mucho *n* vecinos, como se muestra en la figura 5.7, donde se tienen 3 ecuaciones y 2 incógnitas, por lo que el punto P contiene 6 puntos vecinos.

Figura 5.6. Análisis de complejidad



Una **forma ingenua para realizar una iteración** sería revisar cada vecino potencial para observar si es realmente un vértice del poliedro de restricciones y determinar su costo. Calcular el costo es sencillo: simplemente se realiza un producto punto, pero revisar si es un vértice que cumpla con las restricciones representa resolver un sistema de n ecuaciones y n incógnitas por el método de Gauss, teniendo una complejidad de $\mathcal{O}(n^3)$ (con la suposición de que se tienen n inecuaciones), lo que da un tiempo aproximado de ejecución del $\mathcal{O}(m.n^4)$ por iteración.

Afortunadamente, existe un mejor camino, y el factor $m.n^4$ se puede mejorar a $m.n$, lo que hace del Simplex un algoritmo práctico. El algoritmo Simplex, estando en un punto factible (la fase II), permite mejorar el valor de la función objetivo de manera acelerada, ya que no existe una búsqueda a ciegas del nuevo punto factible. Para seleccionar el mejor vecino, la función objetivo es de la forma "max c_u + c .y", donde *cu* es el valor de la función objetivo en el punto *u*. Esto, de forma directa, identifica un sentido prometedor hacia donde moverse: en este caso se busca cualquier punto tal que $\hat{c} > 0$ (si no existe alguno, entonces el punto u es óptimo y el algoritmo se detiene).

Por ende, se puede observar que el tiempo por iteración del Simplex es $\mathcal{O}(nm)$. Pero ¿cuántas iteraciones se pueden realizar? Por supuesto, no pueden ser más de $\binom{n+m}{n}$, cota superior indicada por el número de vértices. Esta cota superior es exponencial en *n*.

De hecho, existen ejemplos de programación lineal en los cuales el método Simplex requiere un número exponencial de iteraciones. Por ello, el método Simplex es un algoritmo de grado exponencial. Sin embargo, esto no sucede en la práctica, lo cual en realidad hace que el algoritmo sea tan valorado y ampliamente usado.

Por supuesto que Simplex no es un algoritmo de tiempo polinomial. Ciertamente, pocos problemas causan que el algoritmo vaya de una esquina factible a una esquina factible mejor hasta encontrar el óptimo en un tiempo exponencial. Por ello, por un largo tiempo **la programación lineal se ha considerado una paradoja: ¡un problema que se puede resolver en tiempo polinomial en práctica, pero no en teoría!**

Entonces, en 1979, un joven matemático soviético llamado Leonid Khachiyan presentó el algoritmo del elipsoide (el cual es totalmente diferente al Simplex). Este nuevo algoritmo es extremadamente simple en su concepción (pero sofisticado en su prueba) y puede resolver cualquier problema de programación lineal en tiempo polinomial en lugar de moverse de una esquina a otra del poliedro. El algoritmo de Khachiyan confina a un elipsoide cada vez más pequeño hasta localizar el punto óptimo.

Cuando este algoritmo fue anunciado, se convirtió en un Sputnik matemático. Incluso Estados Unidos se preocupó bastante de la posible superioridad científica de la Unión Soviética, ya que la demostración se publicó en la parte más tensa de la guerra fría. El algoritmo del elipsoide fue un avance teórico importante, aunque en la práctica no ha podido competir con el método Simplex. En este caso, **la paradoja de programación lineal es la siguiente: un problema con dos algoritmos (uno que es eficiente en teoría y uno que es eficiente en la práctica).**

Unos años después Narendra Karmarkar, un estudiante graduado de UC Berkeley, desarrolló un algoritmo completamente diferente para ejecutar en un tiempo polinomial. El algoritmo de Karmarkar

se conoce como *método del punto interior,* porque justamente realiza eso: no va al óptimo recorriendo cada esquina factible de un poliedro (como lo realiza el método Simplex); en lugar de eso, realiza un camino inteligente en el interior del poliedro.

Pero tal vez el gran avance en los algoritmos de programación lineal no fue la penetración teórica de Khachiyan o el aprovechamiento novedoso de Karmarkar, sino la fiera competencia entre los algoritmos Simplex y punto interior, lo cual provocó el desarrollo de un código sumamente eficiente para programación lineal (Dasgupta, Papadimitriou y Vazirani, 2008).

Por último, vale comentar que algunos problemas del tipo NP se pueden reducir a programación lineal o programación entera. Como ejemplo de ello están el problema SAT y el problema del agente viajero.

Retorno sobre la misma ruta (backtracking)

INTRODUCCIÓN

En la búsqueda de los principios fundamentales del diseño de algoritmos, backtracking representa una de las técnicas más generales. Varios problemas que tratan la búsqueda de un conjunto de soluciones o intentan hallar una solución óptima satisfaciendo algunas restricciones pueden ser resueltos empleando *backtracking*.

Para utilizar este método, el problema debe de estar expresado en n tuplas $(x_1, x_2, x_3, \ldots, x_n)$, donde x_i se escoge desde un conjunto finito S_i. A veces el problema que se quiere resolver es maximizar o minimizar una función $P(x_1, x_2, x_3, \ldots, x_n)$. **A veces se buscan todos los vectores que satisfacen a P.** Por ejemplo, ordenar los enteros localizados en $A(1: n)$ es un problema cuya solución se expresa por *n* tuplas, donde x_i es el índice de A, donde se localiza el i-ésimo elemento más pequeño. La función *P* es la desigualdad $A(x_i) \leq A(x_{i+1})$ para $1 \leq i < n$. La ordenación de números no es en sí un ejemplo de *backtracking*, solo es una muestra de un problema cuya solución se puede formular por medio de n tuplas. En esta sección se estudiarán algunos algoritmos cuya solución se realiza mejor con *backtracking*.

Suponga que m_i es el tamaño del conjunto S_i. Entonces existen $m = m_1, m_2, \ldots, m_n$ tuplas cuyos posibles candidatos pueden satisfacer la función *P.* el enfoque de la "fuerza bruta" formaría todas las n tuplas y evaluaría a cada una con P, salvando a los que producen el óptimo. La virtud de *backtracking* es la habilidad para producir la misma respuesta con un menor número de pasos. Su idea básica es construir un vector y evaluarlo con la función $P(x_1, x_2, x_3, \ldots, x_n)$ para probar si el vector recién formado tiene una oportunidad de ser el óptimo. La gran ventaja de este método es la siguiente: si se observa que el vector parcial $(x_1, x_2, x_3, \ldots, x_n)$ no tiene la posibilidad de conducir hacia una posible solución óptima, entonces $m_{1+1}, \ldots, m_n$ puede ser ignorado completamente.

Varios de los problemas que se resuelven utilizando *backtracking* requieren que todas las soluciones satisfagan a un complejo conjunto de restricciones. Estas restricciones pueden ser divididas en dos categorías: explícitas e implícitas. Las primeras son reglas cuyas restricciones de cada x_i toman valores para un conjunto determinado. Un ejemplo de restricciones explícitas es:

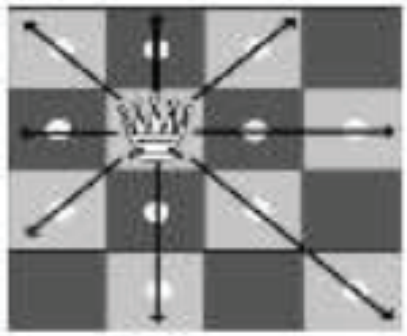
$x_i > 0$	o	S_i = {todos los números reales son no negativos}
$x_i = 0$ o 1	o	$S_i = \{0, 1\}$
$l_i \leq x_i \leq u_i$	o	$S_i = \{a: l_i \leq a \leq u_i\}$

Las restricciones explícitas pueden o no depender de una particular instancia I del problema a ser resuelto. Todas las tuplas que satisfacen a restricciones explícitas definen un posible espacio de solución para I. Las restricciones implícitas determinan cuál de las tuplas en un espacio de solución de I en realidad satisfacen la función del criterio. Así, las restricciones implícitas describen el camino en el cual las x_i deben de relacionarse una a otra.

EL PROBLEMA DE LAS OCHO REINAS (8-QUEENS)

Un problema combinatorio clásico es colocar las ocho reinas en un tablero de ajedrez de tal forma que no se puedan atacar entre ellas, es decir, que no existan dos reinas en la misma hilera, columna o diagonal, enumerando las hileras y columnas del tablero del 1 al 8, del mismo modo que se haría con las reinas. Ya que cada reina debe estar sobre una hilera diferente, se puede asumir que la reina i se colocará en la hilera i. Toda solución puede ser representada como 8 tuplas (x_1,…, x_8) donde x_i es la columna en la cual la reina i será colocada. La restricción explicita usando esta formulación será S = {1, 2, 3,…, 8}, $1 \leq i \leq n$, por lo que el espacio de soluciones consta de 8^8 tuplas de 8. Una de las restricciones implícitas del problema es que dos x no deben de ser las mismas (esto es, toda reina debe de estar en diferente columna) y tampoco en la misma diagonal (figura 6.1). La primera restricción indica que todas las soluciones son permutaciones de las ocho tuplas (1,.., 8). Esto reduce el espacio de la solución de 8^8 tuplas a 8! tuplas.

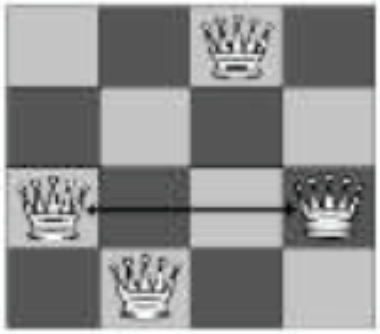
Figura 6.1. Posiciones que puede atacar la reina



PLANTEAMIENTO DEL PROBLEMA

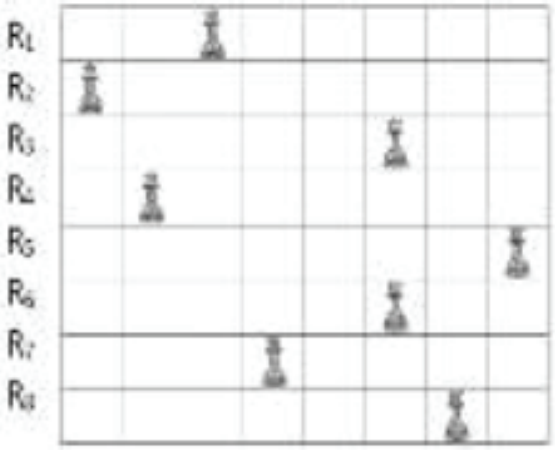
Como cada reina puede amenazar a todas las reinas que estén en la misma fila (figura 6.2), cada una ha de situarse en una fila diferente. Se pueden representar las ocho reinas mediante un vector [1-8], teniendo en cuenta que cada índice del vector representa una fila y el valor una columna. Así, cada reina estaría en la posición *(i, v[i])* para i = 1-8.

Figura 6.2. Ejemplo de dos reinas amenazadas en el tablero de 4 por 4



El vector (3,1,6,2,8,6,4,7) significa que la reina 1 está en la columna 3, fila 1; la reina 2 en la columna 1, fila 2; la reina 3 en la columna 6, fila 3; la reina 4 en la columna 2, fila 4, etc. (figura 6.3). Como se puede apreciar, esta solución es incorrecta, ya que estarían la reina 3 y la 6 en la misma columna. Por tanto, el vector correspondería a una permutación de los ocho primeros números enteros.

Figura. 6.3. Un ejemplo de ocho reinas



El problema de las filas y columnas está cubierto, pero qué ocurre con las diagonales. Para las posiciones sobre una misma diagonal descendente se cumple que tienen el mismo valor *fila – columna*, mientras que para las posiciones en la misma diagonal ascendente se cumple que tienen el mismo valor *fila + columna*. Así, si se tienen dos reinas colocadas en posiciones *(i, j) y (k, l)*, entonces están en la misma diagonal si y solo si cumple:

$$i - j = k - l \text{ o } i + j = k + l$$

$$j - l = i - k \text{ o } j - l = k - i$$

Teniendo todas las consideraciones en cuenta, se puede aplicar el esquema *backtracking* para implementar las ocho reinas de una manera realmente eficiente. Para ello, se reformula el problema como uno de búsqueda en un árbol. Entonces se dice que en un vector $V_{1...k}$ de enteros entre 1 y 8 es k-prometedor, para $0 \leq k \leq 8$ si ninguna de las k reinas colocadas en las posiciones $(1, V_1), (2, V_2), \ldots, (8, V_8)$ amenaza a ninguna de las otras. Las soluciones a nuestro problema se corresponden con aquellos vectores que son 8-prometedores.

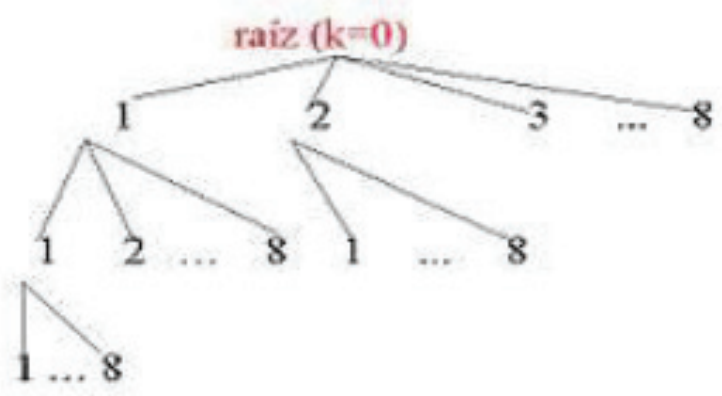
Establecimiento del algoritmo

Sea N el conjunto de vectores de k-prometedores, $0 \leq k \leq 8$, sea $G = (N, A)$ el grafo dirigido tal que (U, V) ϵ A, si y solo si existe un entero k, con $0 \leq k \leq 8$ tal que

- U es k-prometedor.
- V es (k + 1)-prometedor.
- $U_i = V_i$ para todo i ϵ {1,..., k}.

Este grafo es un árbol. Su raíz es el vector vacío correspondiente a $k = 0$. Sus hojas son o bien soluciones $(k = 8)$ o posiciones sin salida $(k < 8)$. Las soluciones del problema de las ocho reinas se pueden obtener explorando este árbol. Sin embargo, no se genera explícitamente el árbol para explorarlo después. Los nodos se van generando y abandonando en el transcurso de la exploración mediante un recorrido en profundidad (figura 6.4).

Figura 6.4. Esquema reducido del árbol de soluciones



Hay que decidir si un vector es k-prometedor; sabiendo que es una extensión de un vector (*k* – 1)-prometedor, únicamente se requiere comprobar la última reina que se deba añadir. Este se puede acelerar si se asocia a cada nodo prometedor el conjunto de columnas, el de diagonales positivas (a 45 grados) y el de diagonales negativas (a 135 grados) controlados por las reinas que ya están puestas.

Descripción del algoritmo

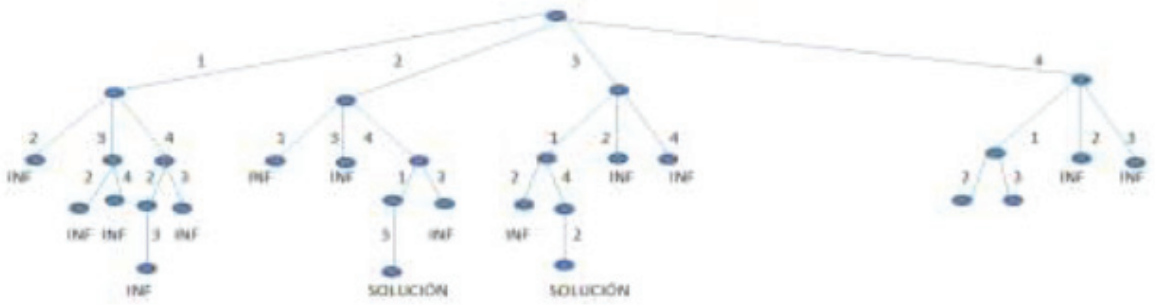
A continuación, se muestra el algoritmo (Horowitz y Sahni, 1978) que arroja la solución de nuestro problema, en el cual $S_{1...8}$ es un vector global. Para imprimir todas las soluciones, la llamada inicial es *reinas (0,0,0,0)* (algoritmo 6.1).

Algoritmo 6.1. Algoritmo para la solución del problema

```
Procedimiento reinas (k, col, diag45, diag135)
//sol1 . . . k es el k prometedor
//col = {soli | 1 ≤ i ≤ k}
//diag45 = {soli - i + 1| 1 ≤ i ≤ k}
// diag135 = {soli + i - 1| 1 ≤ i ≤ k}
si k = 8 entonces //un vector 8-prometedor es una solución
 Escribir sol
Si no //explorar las extensiones (k+1) prometedoras de sol
 Para j←1 hasta 8 hacer
 Si j∉ col y j – k ∉ diag45 y j + k ∉ diag135 entonces
 Sol k + 1 ← j//sol1, . . ., k+1 es (k + 1) prometedor
Reinas (k+1, col U {j}, diag45 U {j – k}, diag135 U {j + k})
```

El algoritmo comprueba primero si *k = 8*. Si esto es cierto, se tiene un vector 8-prometedor, lo cual indica que cumple todas las restricciones originando una solución. Si k es distinto de 8, el algoritmo explora las extensiones *(k + 1)*-prometedoras; para ello realiza un bucle, el cual va de 1 a 8 debido al número de reinas. En este bucle se comprueba si entran en jaque las reinas colocadas en el tablero; si no entran en jaque, se realiza una recurrencia en la cual se incrementa k (buscamos *(k + 1)*-prometedor) y se añaden la nueva fila, columna y diagonales al conjunto de restricciones. Al realizar la recurrencia se ha añadido al vector *sol* una nueva reina, la cual no entra en jaque con ninguna de las anteriores. Además, se ha incrementado el conjunto de restricciones añadiendo una nueva fila, columna y diagonales (una positiva y otra negativa) prohibidas. Un ejemplo del árbol en profundidad se puede ver fácilmente en la figura 6.5 con un ejemplo de las 4 reinas:

Figura 6.5. Un árbol de decisión para cuatro reinas



El algoritmo 6.2 muestra el programa de las *n* reinas:

Algoritmo 6.2. El problema de la n reinas

```
#include<stdio.h>
#include<stdlib.h>
void marcar(int **,int,int,int);
void vaciar(int **,int);
void solucion(int **,int);
void dam(int **,int **,int,int,int,int);
void regresar(int **,int **,int,int,int);
int cont=0;

main(){ system("color 2f");
   int **matriz,**tablero,reinas,fila=0,columna=0;
   printf("Introduce el numero de reinas: "); scanf("%d",&reinas);
   matriz=(int **)malloc(sizeof(int *)*reinas);
   tablero=(int **)malloc(sizeof(int *)*reinas);
   for(int i=0;i<reinas;i++){
    matriz[i]=(int *)malloc(sizeof(int)*reinas);
    tablero[i]=(int *)malloc(sizeof(int)*reinas); }
   vaciar(matriz,reinas);
   vaciar(tablero,reinas);
   for(int i=0;i<reinas;i++) dam(matriz,tablero,i,0,1,reinas);
   if(cont==0)  printf("No  hay  soluciones  para  el  problema  con  %d
reinas\n",reinas);
   system("PAUSE");
}

void  dam(int  **matriz,int  **tablero,int  fila,int  columna,int  reinas,int
R){
   matriz[fila][columna]=1;
   tablero[fila][columna]=1;
   marcar(matriz,R,fila,columna);
   if(reinas==R) solucion(tablero,reinas);
   else{
    for(int j=0;j<R;j++)
    if(matriz[j][columna+1]==0)
    dam(matriz,tablero,j,columna+1,reinas+1,R);
```

```
void solucion(int **vect,int reinas){
      printf("Solucion %d \n",++cont);
      for(int i=0;i<reinas;i++){
         for(int j=0;j<reinas;j++){ printf("%d ",vect[i][j]); }
         printf("\n");
       } printf("\n\n");
}

void vaciar(int **vect,int reinas){
      for(int i=0;i<reinas;i++)
      for(int j=0;j<reinas;j++) vect[i][j]=0;
}

void marcar(int **matriz,int reinas,int falfil,int calfil){
      for(int fila=0;fila<reinas;fila++)
      for(int columna=0;columna<reinas;columna++)
      if((fila+columna==falfil+calfil)||(fila-columna==falfil-calfil))
      matriz[fila][columna]=1;
      for(int fila=0;fila<reinas;fila++){
      matriz[falfil][fila]=1;
      matriz[fila][calfil]=1; }
 }
```

Ciclo hamiltoniano (hamiltonian cycles)

INTRODUCCIÓN

El problema de los puentes de Königsberg, también llamado más específicamente *problema de los siete puentes de Königsberg*, es un célebre problema matemático, resuelto por Leonhard Euler en 1736 y cuya solución dio origen a la teoría de grafos. Su nombre se debe a Königsberg, capital de Prusia Oriental hasta 1945, cuando fue tomada por los soviéticos, los cuales la renombraron como Kaliningrado.

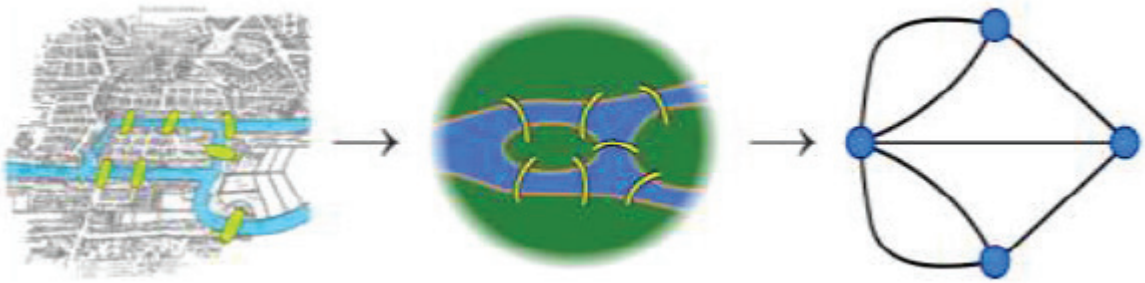
Esta ciudad es atravesada por el río Pregel (en ruso Pregolya), el cual se bifurca para rodear con sus brazos a la isla Kneiphof, dividiendo el terreno en cuatro regiones distintas, las que entonces estaban unidas mediante siete puentes llamados puente del herrero, *puente conector, puente verde, puente del mercado, puente de madera, puente alto y puente de la miel*. El problema fue formulado en el siglo XVIII y consistía en encontrar un recorrido para cruzar a pie toda la ciudad pasando solo una vez por cada uno de los puentes y regresando al mismo punto de inicio. El problema, formulado originalmente de manera informal, consistía en responder la siguiente pregunta:

> Dado el mapa de Königsberg, con el río Pregel dividiendo el plano en cuatro regiones distintas, que están unidas a través de los siete puentes, ¿es posible dar un paseo comenzando desde cualquiera de estas regiones, pasando por todos los puentes, recorriendo solo una vez cada uno, y regresando al mismo punto de partida?

La respuesta es negativa, es decir, no existe una ruta con estas características. El problema puede resolverse aplicando un método de fuerza bruta, lo que implica probar todos los posibles recorridos existentes. Sin embargo, Euler en 1736, en su publicación *Solutio problematis ad geometriam situs pertinentis*, demostró una solución generalizada del problema, que puede aplicarse a cualquier territorio en que ciertos accesos estén restringidos a ciertas conexiones, tales como los puentes de Königsberg.

Para dicha demostración, Euler recurrió a una abstracción del mapa, enfocándose exclusivamente en las regiones terrestres y en las conexiones entre ellas. Cada puente lo representó mediante una línea que unía a dos puntos, cada uno de los cuales representaba una región diferente. De este modo el problema se reduce a decidir si existe o no un camino que comience por uno de los puntos azules (figura 6.6), transitando por todas las líneas una única vez y regresando al mismo punto de partida.

Figura 6.6. Los puentes de Königsberg representados en un grafo

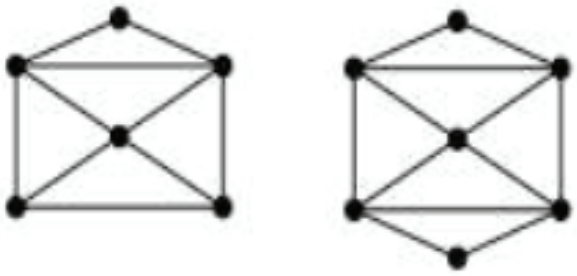


Euler determinó, en el contexto del problema, que los puntos intermedios de un recorrido posible necesariamente han de estar conectados a un número par de líneas. En efecto, si se llega a un punto desde alguna línea, entonces el único modo de salir de ese punto es por una línea diferente.

Esto significa que tanto el punto inicial como el final serían los únicos que podrían estar conectados con un número impar de líneas. Sin embargo, el requisito adicional del problema dice que el punto inicial debe ser igual al final, por lo que no podría existir ningún punto conectado con un número impar de líneas.

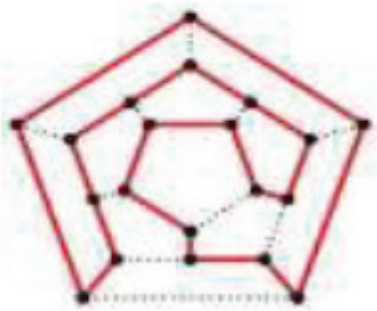
En particular, como en la figura 6.6, los cuatro puntos poseen un número impar de líneas incidentes (tres de ellos inciden en tres líneas y el restante incide en cinco), por lo que se concluye que es imposible definir un camino con las características buscadas que son los siete puentes de Königsberg. Por eso, un circuito euleriano en un grafo G es un circuito que recorre cada arista una y solo una vez (figura 6.7):

Figura 6.7. Circuito euleriano en un grafo G



En 1859 William Rowan Hamilton inventó un juego que consistía en encontrar un recorrido de todos los vértices de un dodecaedro sin repetir vértices y volviendo al original (figura 6.8).

Figura 6.8. Ejemplo de un ciclo hamiltoniano



Un ejemplo práctico se observa en la figura 6.9:

Figura 6.9. Ejemplo práctico de un ciclo hamiltoniano

Los caminos y ciclos hamiltonianos fueron nombrados después que su inventor lanzara un juguete que involucraba encontrar un ciclo hamiltoniano en las aristas de un grafo de un dodecaedro. Hamilton resolvió este problema usando cuaterniones (extensión de los números reales, similar a la de los números complejos), aunque esta solución no se generaliza a todos los grafos.

En el campo matemático de la teoría de grafos, un camino hamiltoniano en un grafo es un camino, una sucesión de aristas adyacentes, que visita todos los vértices del grafo una sola vez. Si, además, el último vértice visitado es adyacente al primero, el camino es un ciclo hamiltoniano, aunque con una observación: si un grafo no es conexo, no puede tener camino ni circuito Hamiltoniano.

El vector solución por *backtraking* (x_1, x_2, x_3,..., x_n) se define de tal forma que x_i representa el i-ésimo vértice visitado del ciclo propuesto. Ahora, todo lo que se tiene que hacer es determinar cómo calcular el conjunto posible de vértices para x_k si x_1,..., x_{k-1} han sido ya escogidos. Si k = 1, entonces X (1) puede ser cualquiera de los n vértices. Para evitar la impresión del mismo ciclo n veces se requiere que X (1) = 1. Si 1 < k < n, entonces X(k) puede ser cualquier vértice v, el cual es distinto de X (1), X (2),..., X(k-1) y v es conectado por una arista a X(k-1). X(n) puede ser solo un vértice restante y debe ser conectado a ambos X(n-1) y X (1). El pseudocódigo se muestra en el algoritmo 6.3:

Algoritmo 6.3. Pseudocódigo (Horowitz y Sahni, 1978)

```
Procedure NEXTVALUE(K)
//X(1),… X(k-1) es un trayecto de k-1 vertices. Si X(k)=0 entonces no se
//ha asignado un vértice a X(K). Después de la ejecución de X(k).
//es asignado al siguiente vértice numerado mayor el cual (i) aun no
aparece //en X(1), . . , X(k-1). De otra manera X(k)=0. Si k=n entonces en
adición //X(k) se conecta a X(1).
Global integer n, X(1:n), Boolean GRAPH(1:n, 1:n)
Integer k, j
Loop
X(k)← (X(k)+1) mod (n+1)//el siguiente vértice.
if X(k) =0 then return endif
if GRAPH (X(k-1), X(k)) then //existe una arista?
For j←1 to k-1 do //verificación de distinción
If (X(j)=X(k) then
Exit
Endif
Repeat
If j=k then //si es verdadero entonces el vértice es distinto.
If k<n or (k=n and GRAPH(X(n),1)) then return
Endif
Endif
Endif
Repeat
End NEXTVALUE
```

Usando el procedimiento *nextvalue*, se puede particularizar el esquema recursivo de *backtracking* para encontrar todos los ciclos hamiltonianos (algoritmo 6.4).

Algoritmo 6.4. Procedimiento *nextvalue* (Horowitz y Sahni, 1978)

```
Procedure HAMILTONIAN (K)
//Este procedimiento usa una formulación recursiva de backtracking para
//encontrar todos los ciclos Hamiltonianos de la gráfica. La gráfica se
//guarda como una matriz adyacente en GRAPH(1:n, 1:n). Todo ciclo inicia
en //el vértice 1.
global integer X(1:n)
local integer k, n
loop //genera valores para X(k)
call NEXTVALUE(K)
if X(k)=0 then return endif
if k=n then
print(X, '1')
else call HAMILTONIAN(k+1)
endif
repeat
end HAMILTONIAN
```

Este procedimiento primero inicializa la matriz adyacente GRAPH (1: n, 1:n), a continuación establece X(2:n)←0, X(1)←1 y ejecuta la llamada a HAMILTONIAN(2). El problema del agente viajero es un ciclo hamiltoniano con la diferencia de que cada arista tiene un costo diferente. El algoritmo 6.5, programado en C, es una combinación del programa del agente viajero por medio de programación dinámica y el código de las n reinas que resuelve el ciclo hamiltoniano:

Algoritmo 6.5. Ciclo hamiltoniano

```
#include <stdio.h>
#include<stdlib.h>

int **generar(int *);
void Hamilton(int **,int *,int **,int,int,int, int *);
void solucion(int **, int);

main(){
      int **cost,*m,d,o,v,i,j,r,**t, control, *ptrcontrol;
      printf(" *************** CICLO HAMILTONIANO ***************\n");
      cost=generar(&v);
      m=(int *)malloc(sizeof(int)*v);
      t=(int **)malloc(sizeof(int*)*v);
      for (i=0;i<v;i++) t[i]=(int*)malloc(sizeof(int)*2);
      for(i=0;i<v;i++) m[i]=0;
      for (i=0;i<v;i++)
      for (j=0;j<2;j++) t[i][j]=0;
      printf("El grafo es el siguiente\n");
      for(i=0;i<v;i++){
      for(j=0;j<v;j++){ printf("%6d",cost[i][j]); } printf("\n");
      }
      control=0;
      ptrcontrol=&control;
      Hamilton(cost,m,t,v,0,v,ptrcontrol);
      if (*ptrcontrol==0)
      printf("El grafo no tiene ciclo Hamiltoniano\n"); system("PAUSE");
}
```

```
int **generar(int *tam){
      int v,i,j,**cost;
      printf("\nIntroduce el numero de vertices: "); scanf("%d",&v);
      cost=(int **)malloc(sizeof(int *)*v);
      for(i=0;i<v;i++) cost[i]=(int *)malloc(sizeof(int)*v);
      for(i=0;i<v;i++)
      for(j=0;j<v;j++)cost[i][j]=0;
      printf("Colocar el valor uno si existe trayecto, o cero si no existe
trayecto:\n");
      for (i=0;i<v;i++)
      for (j=0;j<v;j++){
      printf("trayecto[%d][%d]=",i+1,j+1); scanf("%d",&cost[i][j]);
          }
      *tam=v;
      return cost;
}

void solucion(int ** t, int v){
      printf("Una solución es:\n");
      for (int i=0;i<v;i++)
      printf("de %d a %d\n",t[i][0],t[i][1]);
}

void Hamilton(int **cost,int *m,int **t,int d,int o,int r,int * control){
      if(d==1){
      if (cost[o][0]==1){
      t[r-1][0]=o+1;
      t[r-1][1]=0+1;
      solucion(t,r);
      *control=1;
          }
      }
      m[o]=1;
      for(int i=0;i<r;i++)
      if(m[i]==0){
      if (cost[o][i]==1){
      t[r-d][0]=o+1;
      t[r-d][1]=i+1;
      Hamilton(cost,m,t,d-1,i,r,control);
      m[i]=0;
                        }
   }
}
```

Ramificación y acotamiento (branch and bound)

INTRODUCCIÓN

El método de diseño de algoritmos ramificación y acotamiento (también llamado *ramificación* y *poda*) es una variante del *backtracking* mejorado sustancialmente. El término (del inglés, *branch and bound*) se aplica mayoritariamente para resolver cuestiones o problemas de optimización.

La técnica de ramificación y acotamiento se suele interpretar como un árbol de soluciones, donde cada rama lleva a una posible solución posterior. La característica de esta técnica con respecto a otras anteriores (y a la que debe su nombre) es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas para "podar" esa rama del árbol y no continuar malgastando recursos y procesos en casos que se alejan de la solución óptima.

DESCRIPCIÓN GENERAL

Nuestra meta será encontrar el valor mínimo (o máximo) de una función *f(x)* (un ejemplo puede ser el coste de manufacturación de un determinado producto), para lo que se fijan x rangos sobre un determinado conjunto S de posibles soluciones. Un procedimiento de ramificación y poda requiere dos herramientas:

- La primera es la de un procedimiento de expansión, que dado un conjunto fijo S de candidatos, devuelva dos o más conjuntos más pequeños S_1 , S_2 ,…, S_n cuya unión cubra a S. Nótese que el mínimo de *f(x)* sobre *S* es min{ V_1, V_2,…} donde cada v_i es el mínimo de f(x) sin S_i. Este paso es llamado ramificación; como su aplicación es recursiva, esta definirá una estructura de árbol cuyos nodos serán subconjuntos de *S*.

- La segunda es el procedimiento de poda. La idea clave del algoritmo de ramificación y acotamiento es la siguiente: si estamos buscando un mínimo, y para una rama de algún árbol (conjunto de candidatos), una hoja A del árbol tiene un punto factible con un valor mayor a otro nodo hijo B de otra rama, entonces A debe ser descartada con seguridad de la búsqueda. Algo equivalente sucede cuando se busca un máximo. Este paso es llamado poda, y usualmente es implementado manteniendo una variable global m que graba el mínimo nodo padre visto entre todas las subregiones examinadas hasta entonces. Cualquier nodo hoja cuyo valor es mayor que m puede ser descartado. La recursión se detiene cuando el

conjunto candidato *S* es reducido a un solo elemento, o también cuando el nodo padre para el conjunto S coincide con el nodo hijo. De cualquier forma, cualquier elemento de *S* va a ser el mínimo de una función sin S.

El pseudocódigo del algoritmo 7.1 de ramificación y poda es el siguiente:

Algoritmo 7.1. Pseudocódigo de ramificación y poda

```
Funcion RyP {
P = Hijos(x,k)
mientras ( no vacio(P) )
   x(k) = extraer(P)
   si esFactible(x,k) y G(x,k) < optimo
        si esSolucion(x)
        Almacenar(x)
        sino
        RyP(x,k+1)
}
```

donde

- *G(x,k)* es la función de estimación del algoritmo.
- P es la pila de posibles soluciones.
- es*Factible(x,k)* es la función que considera si la propuesta es válida.
- *esSolución* es la función que comprueba si se satisface el objetivo.
- *Óptimo* es el valor de la función a optimizar evaluado sobre la mejor solución encontrada hasta el momento.

Subdivisión efectiva

La eficiencia de este método depende fundamentalmente del procedimiento de expansión de nodos o de la estimación de los nodos padres e hijos. Es mejor elegir un método de expansión que provea que no se solapen los subconjuntos para ahorrarnos problemas de duplicación de ramas.

Idealmente, el procedimiento se detiene cuando todos los nodos del árbol de búsqueda están podados o resueltos. En ese punto, todas las subregiones no podadas tendrán un nodo padre e hijo iguales a una función global mínima. En la práctica, el procedimiento a menudo termina cuando finaliza un tiempo dado, hasta el punto en que el mínimo de nodos hijos y el máximo de nodos padres sobre todas las secciones no podadas definen un rango de valores que contienen el mínimo global. Alternativamente, sin superar un tiempo restringido, el algoritmo debe terminar cuando un criterio de error, tal que el max o min cae bajo un valor específico.

La eficiencia del método depende especialmente de la efectividad de los algoritmos de ramificación y poda usados. Una mala elección puede llevar a una ramificación repetida, hasta que las subregiones se conviertan en muy pequeñas. En ese caso, el método sería reducido a una exhaustiva enumeración del dominio, que es a menudo impracticablemente larga. No hay un algoritmo de poda universal que trabaje para todos los problemas, pero existe una pequeña esperanza de que alguna vez se encuentre alguno. Hasta entonces, se necesitará implementar cada uno por separado para cada aplicación informática, con el algoritmo de ramificación y poda especialmente diseñado para él.

Los métodos de ramificación y poda deben ser clasificados según los métodos de poda y las maneras de creación/clasificación de los árboles de búsqueda. La estrategia de diseño de ramificación y poda es muy similar al de vuelta atrás (*backtracking*), donde el estado del árbol es usado para resolver un problema. Las diferencias son que el método de ramificación y poda no nos limitan a ninguna forma particular de obtener un árbol transverso, y es usado solamente para problemas de optimización.

ESTRATEGIAS DE PODA

Nuestro objetivo principal será eliminar aquellos nodos que no lleven a soluciones buenas. Podemos utilizar dos estrategias básicas. Supongamos un problema de maximización donde se han recorrido varios nodos i = 1,…, *n*, estimando para cada uno la cota superior *CS(xi)* e inferior $CI(x_j)$. Vamos a trabajar sobre un problema sobre el cual se quiere maximizar el valor (si fuese un problema de minimización, entonces se aplicaría la estrategia equivalente).

Estrategia 1

Si a partir de un nodo x_i se puede obtener una solución válida, entonces se podrá podar dicho nodo si la cota superior CS (x_i) es menor o igual que la cota inferior CI (x_j) para algún nodo j generado en el árbol.

Por ejemplo, supongamos el problema de la mochila en el cual se busca una ganancia maximima, donde utilizamos un árbol binario. Entonces, si a partir de x_i se puede encontrar un beneficio máximo de CS (x_i) = 4 y a partir de x_j se tiene asegurado un beneficio mínimo de CI (x_j) = 5, esto nos llevará a la conclusión de que se puede podar el nodo x_i sin que perdamos ninguna posible solución óptima.

Estrategia 2

Si se obtiene una posible solución válida para el problema con un beneficio B_j, entonces se podrán podar aquellos nodos x_i cuya cota superior CS (x_i) sea menor o igual que el beneficio que se puede obtener B_j (este proceso sería similar para la cota inferior).

Estrategias de ramificación

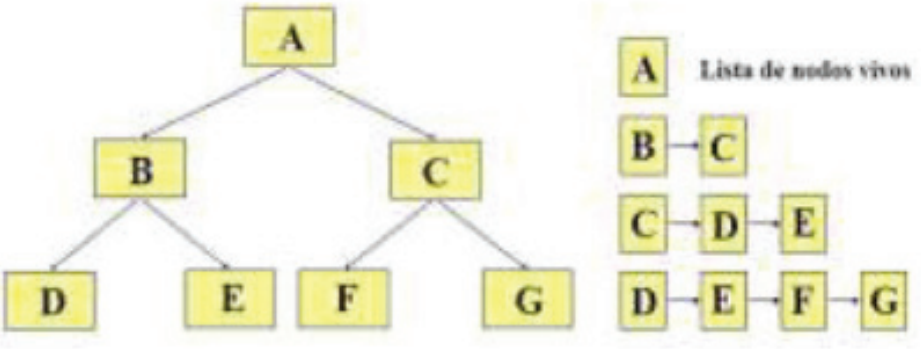
Como se comentó en la introducción de este apartado, la expansión del árbol con las distintas estrategias está condicionada por la búsqueda de la solución óptima. Debido a esto, todos los nodos de un nivel deben ser expandidos antes de alcanzar un nuevo nivel, lo cual es lógico, ya que para poder elegir la rama del árbol que va a ser explorada se deben conocer todas las ramas posibles.

Todos estos nodos que se van generando y que no han sido explorados se almacenan en lo que se denomina *lista de nodos vivos* (a partir de ahora LNV), nodos pendientes de expandir por el algoritmo. La LNV contiene todos los nodos que han sido generados, pero que no han sido explorados todavía. Según como estén almacenados los nodos en la lista, el recorrido del árbol será de uno u otro tipo, lo cual da lugar a las tres estrategias que se detallan a continuación.

Estrategia FIFO

En la estrategia FIFO (*first in first out*), la LNV será una cola, lo cual da lugar a un recorrido en anchura del árbol.

Figura 7.1. Estrategias de ramificación FIFO

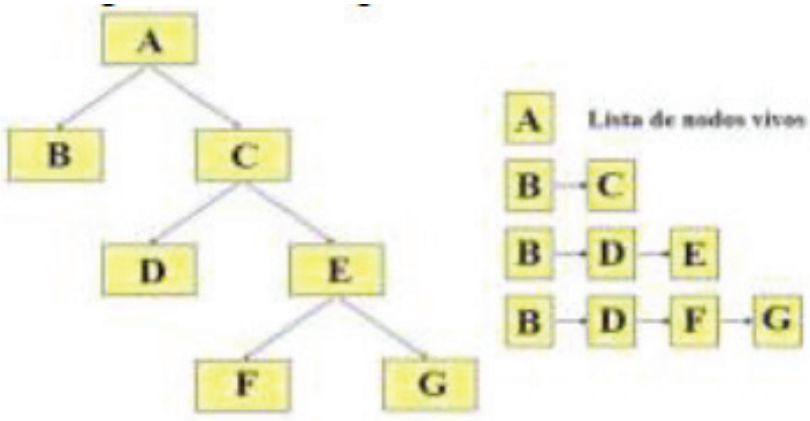


En la figura 7.1 se puede observar que se comienza introduciendo en la LNV el nodo A. Luego se extrae el nodo de la cola y se expande generando los nodos B y C, que son introducidos en la LNV. Seguidamente, se saca el primer nodo, que es el B, y se vuelve a expandir generando los nodos D y E, que se introducen en la LNV. Este proceso se repite mientras que quede algún elemento en la cola.

Estrategia LIFO

En la estrategia LIFO (last in first out), la LNV será una pila, lo que produce un recorrido en profundidad del árbol.

Figura 7.2. Estrategias de ramificación LIFO



En la figura 7.2 se muestra el orden de generación de los nodos con una estrategia LIFO. El proceso que se sigue en la LNV es similar al de la estrategia FIFO, pero en lugar de utilizar una cola se emplea una pila.

Estrategia de menor coste o LC

Al utilizar las estrategias FIFO y LIFO se realiza lo que se denomina una búsqueda "a ciegas", ya que expanden sin tener en cuenta los beneficios que se pueden conseguir desde cada nodo. Si la expansión se realizara en función de los beneficios que cada nodo reporta (con una "visión de futuro"), se podría conseguir en la mayoría de los casos una mejora sustancial.

Es así como nace la estrategia de menor coste o LC (*least cost*), la cual selecciona, para expandir entre todos los nodos de la LNV, a aquel que tenga mayor beneficio (o menor coste). Por tanto, ya no se habla de un avance "a ciegas".

Esto nos puede llevar a la situación de que varios nodos puedan ser expandidos al mismo tiempo. De darse el caso, es necesario disponer de un mecanismo que solucione este conflicto:

- *Estrategia LC-FIFO*: Elige de la LNV el nodo que tenga mayor beneficio; en caso de empate, se escoge el primero que se introdujo.
- *Estrategia LC-LIFO*: Elige de la LNV el nodo que tenga mayor beneficio; en caso de empate, se escoge el último que se introdujo.

Ramificación y poda "relajado"

Una variante del método de ramificación y poda más eficiente se puede obtener "relajando" el problema, es decir, eliminando algunas de las restricciones para hacerlo más permisivo.

Cualquier solución válida del problema original será solución válida para el problema "relajado", pero no tiene por qué ocurrir al contrario. Si conseguimos resolver esta versión del problema de forma óptima, y entonces si la solución obtenida es válida para el problema original, esto querrá decir que es óptima también para dicho problema.

La verdadera utilidad de este proceso reside en la utilización de un método eficiente que nos resuelva el problema relajado. Uno de los métodos más conocidos es el de ramificación y corte (*branch and cut*).

Ramificación y corte

Ramificación y corte es un método de optimización combinatoria para resolver problemas de enteros lineales, los cuales son problemas de programación lineal donde algunas o todas las incógnitas están restringidas a valores enteros. Se trata de un híbrido de ramificación y poda con métodos de

planos de corte.

Este método resuelve problemas lineales con restricciones enteras usando algoritmos regulares simplificados. Cuando se obtiene una solución óptima que tiene un valor no entero para una variable que ha de ser entera, el algoritmo de planos de corte se usa para encontrar una restricción lineal más adelante que sea satisfecha por todos los puntos factibles enteros. Si se encuentra esa desigualdad, se añade al programa lineal, de tal forma que resolverla nos llevará a una solución diferente que esperamos que sea "menos fraccional". Este proceso se repite hasta que se encuentre una solución entera (que podemos demostrar que es óptima) o hasta que no se hallen más planos de corte.

En este punto comienza la parte del algoritmo de ramificación y poda. Este problema se divide en dos versiones: una con restricción adicional en que la variable es más grande o igual que el siguiente entero mayor que el resultado intermedio, y uno donde la variable es menor o igual que el siguiente entero menor. De esta forma se introducen nuevas variables en las bases según el número de variables básicas que no son enteros en la solución intermedia, pero son enteros de acuerdo con las restricciones originales. Los nuevos programas lineales se resuelven empleando un método simplificado y después el proceso es repetido hasta que una solución satisfaga todas las restricciones enteras.

Durante el proceso de ramificación y poda, los planos de corte se pueden separar más adelante, y pueden ser cortes globales válidos para todas las soluciones enteras factibles, o cortes locales que son satisfechos por todas las soluciones llenando todas las ramas de la restricción del subárbol de ramificación y poda actual.

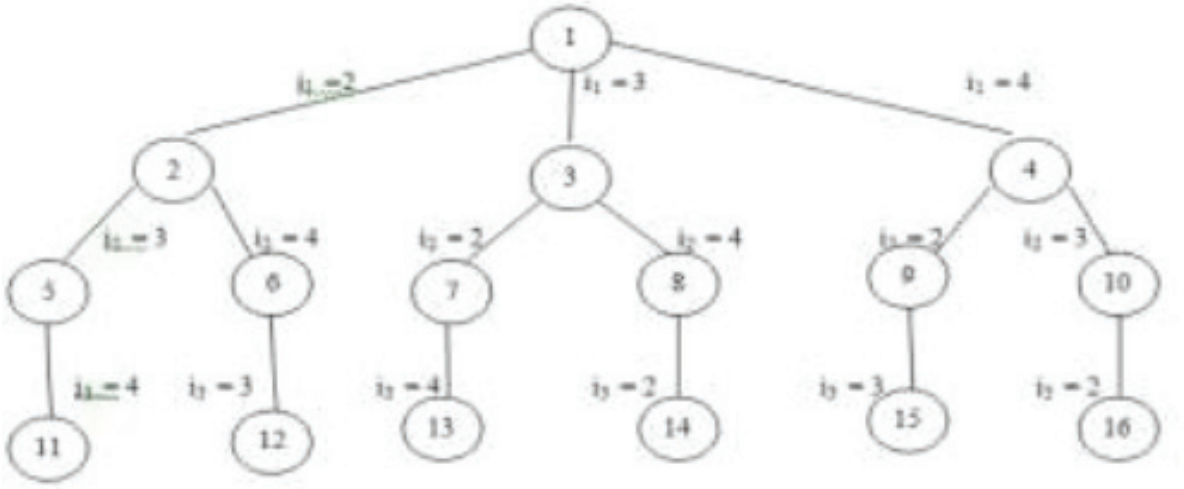
EL PROBLEMA DEL AGENTE VIAJERO (TRAVELING SALESPERSON TSP)

Para programación dinámica, el algoritmo del agente viajero tiene una complejidad de $O(n^2 2^n)$. Ahora, lo que se va a tratar de explicar es el algoritmo visto bajo la óptica de ramificación y acotamiento. El uso de una buena función de acotamiento permitirá que el algoritmo, bajo el paradigma de ramificación y acotamiento, en algunos casos, pueda ser resuelto en mucho menor tiempo que el requerido en programación dinámica.

Sea G = (V, E) una gráfica definida como una instancia del problema del agente viajero, y sea c_{ij} el costo de la arista $<i, j>$, $c_{ij} = \infty$ si $<i, j> \notin$ E y sea | V | = n. Sin pérdida de generalidad, se puede asumir que cualquier recorrido inicia y termina en el vértice 1. De esta forma la solución del espacio S está dado por S = {1, Π , 1| Π es la permutación de (2, 3,..., n)}. |S| = (n – 1)! El tamaño de S puede ser reducido restringiendo S, de modo que $(1, i_1, i_2, \ldots, \in_{\ldots}, 1)$ S si y solo si $< i_j, \ldots \in_{j+1}>$ E, $0 \le j \le n-1$, $i_0 = i_n = 1$. *S* puede estar organizado dentro de un árbol de estados.

La figura 7.3 muestra la organización del árbol para el caso de una gráfica completa con |V| = 4. Cada nodo hoja L es una solución y representa el recorrido definido por el trayecto desde la raíz hasta L. En la figura 7.3 el nodo 14 representa el recorrido $i_0 = 1$, $i_1 = 3$, $i_2 = 4$, $i_3 = 2$ y $i_4 = 1$.

Figura 7.3. Árbol de estados para un problema del agente viajero con n=4 y $i_0 = i_4 = 1$



En orden a usar ramificación y acotamiento para buscar el árbol de estados del agente viajero, se requiere definir una función de costo $\hat{c}(.)$ y otras dos funciones $\hat{c}(.)$ y u(.) de tal forma que $c(.) \leq c(.) \leq u(.)$ para todo nodo R. c(.) es el nodo solución si c(.) es el nodo de menor costo correspondiente al tour más corto en G. Una forma de escoger c(.) es la siguiente:

$$C(A) = \begin{cases} \text{La longitud definida por la trayectoria desde la raíz a A si A es una hoja} \\ \text{El mínimo costo de la hoja en el subárbol A.} \end{cases}$$

Una simple $\hat{c}(.)$ tal que $\hat{c}(A) \leq c(A)$ para todo A es obtenido definiendo $\hat{c}(A)$ a ser el tamaño de la trayectoria definida en el nodo A. Por ejemplo, la trayectoria definida en el árbol anterior es $i_0, i_1, i_2, = 1, 2, 4$. Este consiste en las aristas <1,2> y <2, 4>. Un $\hat{c}(.)$ mejor puede ser obtenido usando la matriz de costo reducida correspondiente a G. Una hilera (columna) se reduce si y solo si contiene al menos un cero y todos los demás valores son no negativos. Una matriz es reducida si y solo si toda hilera y columna es reducida. Como un ejemplo de la reducción del costo de una matriz de una gráfica dada G, consiste en la matriz de la figura 7.4:

Figura 7.4. Un ejemplo de la reducción de una matriz

La matriz corresponde a una gráfica con cinco vértices. Todo recorrido incluye exactamente una arista $<i, j>$ con $i = k$, $1 \leq k \leq 5$ y exactamente una arista $<i, j>$ con $j = k$, $1 \leq k \leq 5$, substrayendo una constante t de todos los elementos en una hilera o una columna de la matriz de costos se reduce el tamaño de cada recorrido exactamente t unidades. Un recorrido de costo mínimo se mantiene después de esta operación de sustracción. Si t se escoge para hacer mínima la entrada en la hilera i (columna *j*), restando i de todas las entradas en la fila *i* (columna *j*), presentará un cero en la fila *i* (columna j). Repitiendo este procedimiento tanto como sea necesario, la matriz de costos puede ser reducida. El monto total substraído de todas las columnas e hileras es el límite inferior y puede ser utilizado como el valor $\hat{c}$ de la raíz del árbol de espacio de estado. Substrayendo 10, 2, 2, 3, 4, de las hileras 1, 2, 3, 4, 5. Posteriormente, sustrayendo 1 y 3 en las columnas 1 y 3 respectivamente de la matriz del inciso b) de la figura 7.4, se tiene la matriz reducida del inciso c) de la misma figura. El monto total substraído es 25. Por lo tanto, todo recorrido del origen a origen tiene una longitud al menos de 25 unidades.

Con todos los nodos del agente viajero del árbol de estados se puede asociar una matriz de costos. Sea A la matriz de costos del nodo R. Sea S el hijo de R tal que la arista del árbol (*R, S*) corresponde a la arista $<i, j>$ en el recorrido. Si S no es una hoja entonces la matriz de costo para S puede ser obtenida de la siguiente forma:

1. Al escoger el trayecto $<i, j>$, cambiar todo valor en la hilera i y columna j de A por ∞. Esto previene el uso de algunas aristas salientes del vértice i o vértices entrantes de j.
2. Colocar A(*j,1*) en ∞. Esto previene el uso de aristas $<j, 1>$.
3. Reducir todas las hileras y columnas en la matriz resultante excepto para las

hileras y columnas que contienen solo ∞. Cada diferencia a cero se suma en la variable r. La matriz resultante será B. Ya realizadas todas las substracciones del paso anterior, entonces:

$\hat{c}(S) = \hat{c}(R) + A\langle i, j\rangle + r$ para nodos hoja $\hat{c}(.) = c(\,)$.

Siendo *S* el número de nodo actual.

Siendo *R* el número de nodo padre.

Los dos primeros pasos son válidos y no existirá un recorrido en el subárbol S que contenga las aristas del tipo *<j, k>* o *<k, j>* o *<j, 1>* (excepto para la arista *<i, j>*). En este momento r es el monto total substraído del paso 3, entonces $\hat{c}(S) = \hat{c}(R) + A\langle i, j\rangle + r$. Para los nodos hoja $\epsilon(.) = c()$ es fácil calcular, ya que cada rama hasta la hoja define un único recorrido. Para la función de la cota superior u, se requiere usar $u(R) = \infty$ para todo nodo R.

Siendo S el número de nodo actual.

Siendo R el número de nodo padre.

Para ver el árbol de transición se tomará el siguiente ejemplo:

$$\begin{vmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{vmatrix}$$

Tomando la notación de la transición:

$l_s = y$

donde

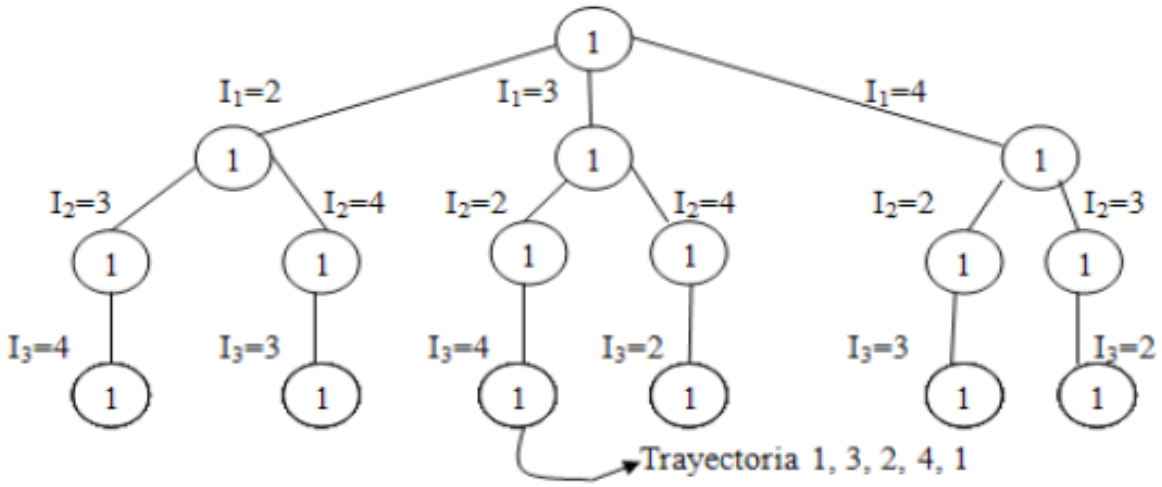
l indica en este caso transición.

s indica el nivel //nodo hijo.

y indica la columna.

De esta forma el árbol de trayectorias para el ejemplo quedaría como se muestra en la figura 7.5:

Figura 7.5. Posibles trayectos



Para localizar la cota inferior se debe realizar la reducción de la matriz. La suma de todas las diferencias será la cota inferior $\hat{c}(.)$. Retomando el primer ejemplo del agente viajero, se tiene:

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Restando por hileras:

A la h_1 se resta 10. Por lo tanto, r = 10

A la h_2 se resta 2. Por lo tanto, r = 12

A la h_3 se resta 2. Por lo tanto, r = 14

A la h_4 se resta 3. Por lo tanto, r = 17

A la h_5 se resta 4. Por lo tanto, r = 21.

La matriz resultante es:

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞

Restando por columna, se tiene:

A la c_1 se resta 1. Por lo tanto, r = 22

A la c_3 se resta 3. Por lo tanto, r = 25.

De esta forma, $\hat{c}(.) = 25$ y la cota superior $U = \infty$, por lo que la matriz resultante es:

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Para S = 2 (1,2)

Ya sabiendo el costo menor, se escoge la primera parte del trayecto, donde A<2,1> = ∞.

∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

En este caso, en toda hilera y en toda columna existe un cero, por lo que r = 0.

Para S = 3 (1,3)

∞	∞	∞	∞	∞
12	∞	∞	2	0
∞	3	∞	0	2
15	3	∞	∞	0
11	0	∞	12	0

En este caso, toda hilera tiene al menos un cero, pero la primera columna es diferente de cero, por lo que r = 11.

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

$\hat{c}(3) = \hat{c}(.) + A<1, 3> + r$

$\hat{c}(3) = 25 + 17 + 11 = 53$

Para S = 4 (1,4)

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

A<4, 1> = ∞.
A<1, 4> = 0.
En este caso, toda hilera y toda columna tiene al menos un cero.

$\hat{C}(4) = 25 + 0 + 0 = 25.$

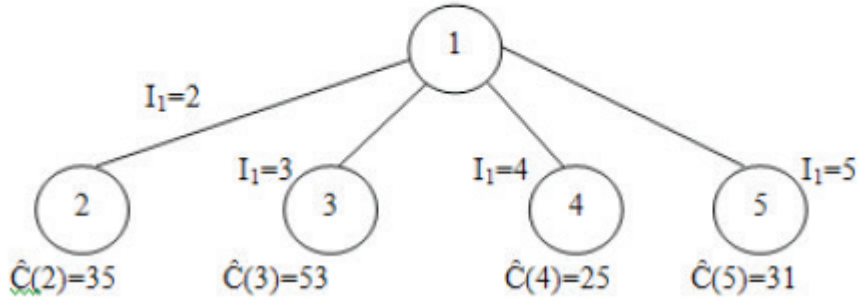
Para S = 5 (1,5)

∞	∞	∞	∞	∞
12	∞	11	2	∞
0	3	∞	0	∞
15	3	12	∞	∞
∞	0	0	12	∞

A<5, 1>=∞
En este caso, la hilera dos y la hiera cuatro tienen todos sus valores diferentes a cero, por lo que se crea otra matriz de la siguiente forma:

∞	∞	∞	∞	∞
10	∞	9	0	∞
0	3	∞	0	∞
12	0	9	∞	∞
∞	0	0	12	∞

Para la hilera 2, r=2. Para la hilera 4, r=3
r=2 + 3=5
$\hat{c}(4) = \hat{c}(.) + A<1, 5> + r$
$\hat{c}(4) = 25 + 1 + 5 = 31$



Se toma el nodo de menor costo, por lo que el nodo padre es S = 4.

Para S = 6 (4,2)

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

A<2, 1> = ∞
Toda hilera y columna no marcada tiene al menos un cero. Por lo que r=0.
$\hat{c}(6) = 25 + 3 = 28$

Para S = 7 (4,3)

∞	∞	∞	∞	∞
12	∞	∞	∞	0
∞	3	∞	∞	2
∞	∞	∞	∞	∞
11	0	∞	∞	∞

A<3,1> = ∞
En la columna 1 todos los números son diferentes de cero, por lo tanto r=11.
De esta forma, la nueva matriz es:

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	3	∞	∞	2
∞	∞	∞	∞	∞
0	0	∞	∞	∞

En la hilera 3 todos los números son diferentes de cero, por lo tanto r=11 + 2 = 13.
De esta forma, la nueva matriz es:

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	1	∞	∞	0
∞	∞	∞	∞	∞
0	0	∞	∞	∞

Checando el valor A<4,3> en S=4 se tiene el valor de 12. Por lo tanto:
$\hat{C}(7)=25+12+13=50$

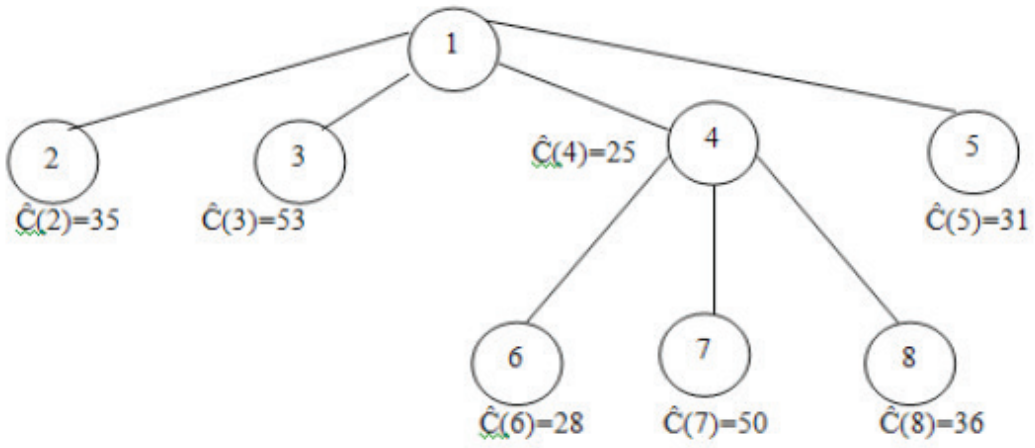
Para S = 8 (4,5)

∞	∞	∞	∞	∞
12	∞	11	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

A<5,1>=∞
En la hilera 2 todos los números son diferentes de cero, por lo tanto r= 11.
De esta forma, la nueva matriz es:

∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

Checando el valor A<4, 5> en S=4 se tiene el valor de 0. Por lo tanto:
$\hat{C}(8)=25+0+11=36$



Siendo S = 6 la ruta mínima se tiene:

Para S = 9 (2,3)

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	∞	∞	∞

A<3, 1>=∞
En las hileras 3 y 5 todos los números son diferentes de cero, por lo tanto:
r=2 + 11 = 13
Quedando la nueva matriz:

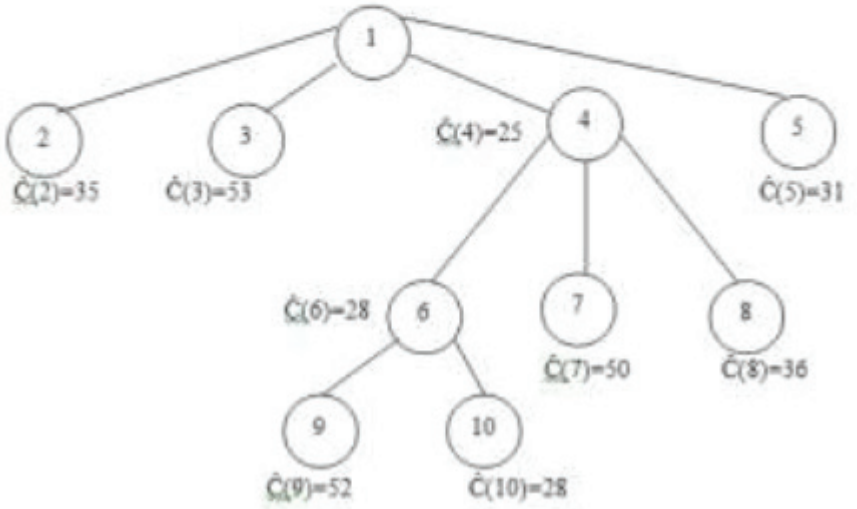
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	0
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

En S=6, A<2, 3> = 11
ĉ(9)=28 + 11 + 13 = 52

Para S = 10 (2,5)

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

A<5, 1> = ∞
r=0
En S=6, A<2, 5>=0.
ĉ(10)=28 + 0 + 0 = 28



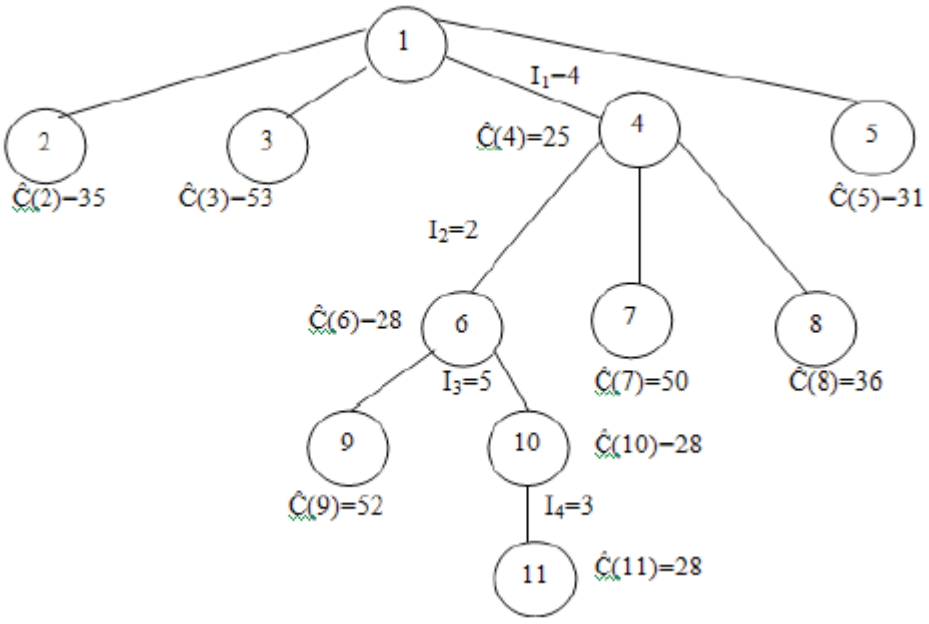
Se observa que el nodo hoja con el menor costo es *S* = 10.

Para S = 11 (5,3)

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

En este caso, es fundamental ir del nodo 3 al nodo 1. Por lo que A<3,1> no se modifica. Ahora bien, en S=6, A<5,3> = 0, de esta forma r=0.

Por lo tanto, $\hat{c}(11)=28 + 0 + 0 = 28$.



Sumando los trayectos a los nodos se tiene:

Del nodo 1 al nodo 4	10 unidades
Del nodo 4 al nodo 2	6 unidades
Del nodo 2 al nodo 5	2 unidades
Del nodo 5 al nodo 3	7 unidades
Del nodo 3 al nodo 1	3 unidades.

TOTAL 28 unidades.

La programación en el lenguaje C se muestra en el algoritmo 7.2:

Algoritmo 7.2. Programación en el lenguaje C

```
#include<stdio.h>
#include<stdlib.h>

struct Lista{ int **matriz,*marcas,costo,contador,ciudad,mc;
Lista *sig; };

void bloquear(Lista *p,int tam,int x,int y){//bloquea la hilera y fila, ademas
A<j,i>
    for(int i=0;i<tam;i++){
    p->matriz[x][i]=999;
    p->matriz[i][y]=999; }
    p->matriz[y][x]=999;
}

void restar_fila(Lista *q,int tam,int min,int i){ //Resta el minimo de cada
fila
      for(int k=0;k<tam;k++)
      if(q->matriz[i][k]!=999 && q->matriz[i][k]!=0)
      q->matriz[i][k]-=min;
 }

void restar_columna(Lista *q,int tam,int min,int i){ //Resta el minimo de
cada columna
      for(int k=0;k<tam;k++)
      if(q->matriz[k][i]!=999 && q->matriz[k][i]!=0)
      q->matriz[k][i]-=min;
 }

void costo(Lista *q,int tam){ //Realiza el costo con el apoyo de la funcion
restar_columna y restar_fila
       int min;
       for(int i=0;i<tam;i++){
        min=q->matriz[i][0];
        for(int j=1;j<tam;j++){
          if(min>q->matriz[i][j]) min=q->matriz[i][j];
       }
       if(min!=999) q->costo+=min;
       if(min!=0 && min!=999) restar_fila(q,tam,min,i);
       }

       for(int i=0;i<tam;i++){
          min=q->matriz[0][i];
          for(int j=1;j<tam;j++){
            if(min>q->matriz[j][i]) min=q->matriz[j][i];
        }
       if(min!=999) q->costo+=min;
       if(min!=0 && min!=999) restar_columna(q,tam,min,i);
       }
 }
```

```
void generar(Lista *q,int tam){ //Genera la matriz de costos minimos
int i,j,cont=0,destino;
for(i=0;i<tam;i++)
for(j=0;j<tam;j++) q->matriz[i][j]=999;
printf("Para dejar de intorducir una ciudad introduce 99");
while(cont<tam){
      printf("\n Ciudad %d \n ",cont+1);
      do{
      printf("De ciudad %d a: ",cont+1);scanf("%d",&destino);
      if(destino==99)
      printf("Termino ciudad %d",cont+1);
      else if(destino>tam || destino<=0)
      printf("Esa ciudad no existe\n");
      else if(cont==destino-1)
      printf("Te encuentras en esta ciudad\n");
      else{
      printf("Introduce el costo de %d a %d: ",cont+1,destino);
      scanf("%d",&q->matriz[cont][destino-1]);
 }
      }while(destino!=99);
cont++;
}
}
```

```
void *guardar(void *p,Lista *mmin,int tam,int num,int ciudad,int j){
//Genera la lista de matricez con costo, caminos por recorrer, marcas y ciudad en
la //que se encuentra
      Lista *aux, *q;
      int min;
      q=(Lista *)malloc(sizeof(Lista));
      if(p==NULL){
         q->matriz=(int **)malloc(sizeof(int *)*tam);
         q->marcas=(int *)malloc(sizeof(int)*tam);
         for(int i=0;i<tam;i++){
         q->matriz[i]=(int *)malloc(sizeof(int)*tam);
         q->marcas[i]=0;
       }
      q->contador=num;
      q->marcas[ciudad]=1;
      q->sig=NULL;
      q->ciudad=j;
      q->costo=0;
      q->mc=1;
      generar(q,tam);
      costo(q,tam);
      p=q;
      } else{
      q->matriz=(int **)malloc(sizeof(int *)*tam);
      q->marcas=(int *)malloc(sizeof(int)*tam);
      for(int i=0;i<tam;i++){
      q->matriz[i]=(int *)malloc(sizeof(int)*tam);
      q->marcas[i]=mmin->marcas[i];
      }
      for(int i=0;i<tam;i++)
      for(int j=0;j<tam;j++)
      q->matriz[i][j]=mmin->matriz[i][j];
      q->contador=num;
      q->marcas[ciudad]=1;
```

```
      q->sig=NULL;
      q->ciudad=j;
      q->costo=0;
      q->mc=0;
      min=q->matriz[ciudad][j];
      if(min!=999) q->costo=mmin->costo+min;
      else q->costo=mmin->costo;
      bloquear(q,tam,ciudad,j);
      costo(q,tam);
      aux=(Lista *)p;
      while(aux->sig!=NULL)
      aux=aux->sig;
      aux->sig=q;
       }
return p;
 }

int main(){
         int tam,num=1,ciudad=0,min;
         Lista *mmin,*aux;
         void *p=NULL;
         printf("Introduce el numero de ciudades: "); scanf("%d",&tam);
         p=guardar(p,mmin,tam,num,0,0);
         mmin=(Lista *)p;
         do{
          //Mientras el num no sea igual al numero de ciudades el arbol no ha
terminado, sigue //recorriendo los caminos que hacen falta y detectando los menores
          aux=(Lista *)p;
          num=num+1;
          for(int i=0;i<tam;i++)
          if(mmin->marcas[i]!=1)
          p=guardar(p,mmin,tam,num,ciudad,i);
          min=999;
          while(aux!=NULL){
          if(aux->costo<min && aux->mc!=1){
          min=aux->costo;
          mmin=aux;}
          aux=aux->sig;
 }
          ciudad=mmin->ciudad;
          mmin->marcas[ciudad]=1;
          mmin->mc=1;
          num=mmin->contador;
          }while(num!=tam);
          printf("\n\n  Recorrido  de  costo  minimo  \n\nCosto  minimo:
%d\n",mmin->costo); system("pause");
          }
```

SECCIÓN III

Tópicos sobre algoritmos

La corriente del conocimiento se dirige hacia una realidad no mecánica: el universo se empieza a parecer más a un gran pensamiento que a una gran máquina. La mente ya no parece ser un intruso accidental en el reino de la materia… más bien debemos saludarlo como el creador y gobernador del reino de la materia.

James Jeans

INTRODUCCIÓN

Se ha pensado que el lenguaje formal permite representar —sin paradojas— el conocimiento, y que es la panacea universal y el mejor camino —si no el único camino— hacia la verdad. Hasta cierto punto, esta opinión sobre los lenguajes formales dominó el pensamiento occidental por más de dos mil años. En un principio, los *Elementos* de Euclides sentaron las bases inconmovibles del pensamiento formal basado en un sistema axiomático y en la lógica, sustentos sobre los que se construyó la ciencia helénica y helínstica, todo el pensamiento medieval y renacentista, así como la filosofía de Kant, que consideró a la geometría euclidiana el paradigma de la razón. Más tarde, Newton, apoyado también en la geometría euclidiana, elaboró su formidable lenguaje formal, a partir del cual la ciencia contemporánea se instaló definitivamente en nuestra cultura y la transformó en la primera, donde para cada ser humano todo en este planeta es objeto de su decisión y, en consecuencia, de su responsabilidad.

Así, las ideas, los razonamientos, las deducciones y los conceptos avalados por un lenguaje formal adquirieron carta de ciudadanía en la ciencia y pasaron a formar parte de la razón y a configurar, hasta dominarlo totalmente, el pensamiento racional. Para 1930 cualquier otra forma de pensamiento —como el metafísico, el teológico o el que se esconde ambiguo, multifacético y escurridizo en la parábola, la alegoría, el mito o la fábula— dejó de ser pensamiento racional y quedó arrinconado entre las antiguallas como la magia, la hechicería, la brujería, es decir, el sinsentido.

Entonces, en 1930, ocurrió un desastre. Un joven matemático natural de Brünn —actual Brno, en República Checa— demostró dos teoremas que cambiaron para siempre las certezas matemáticas que había hasta el momento. Su nombre era Kurt Gödel (1906-1978). El primer teorema lo expuso abiertamente el 7 de septiembre. El artículo con el desarrollo de la demostración fue enviado a la

revista Monatshefte für *Mathematik and Physik* en noviembre y apareció en el volumen 38 (1931), una publicación cuya relevancia para la lógica es solo comparable con la *Metafísica* de Aristóteles. La exposición de la demostración fue tan clara que no generó ni la más mínima controversia. Este fue el embrión de las ciencias de la computación.

ANTECEDENTES DE ALGORÍTMICA

La idea de disponer de un algoritmo o receta para efectuar alguna tarea ha existido durante miles de años. También durante muchos años la gente creyó que si cualquier problema podía iniciarse de manera precisa, entonces con suficiente esfuerzo sería posible encontrar una solución con el tiempo (o tal vez una prueba de que no existe solución podría proporcionarse con el transcurso del tiempo). En otras palabras, se creía que no había problema que fuera tan intrínsecamente difícil que en principio nunca pudiera resolverse.

Se pensaba que las matemáticas eran un sistema *consistente* (es decir, que no era posible llegar a contradicciones a partir de los axiomas iniciales), *completo* (existe una prueba para toda proposición matemática correcta) y *decidible* (existe un método efectivo que decide para cada proposición posible, si es verdadera o falsa. A este se le conoció como el problema de la decisión o *Entscheidungsproblem*).

El proyecto de Hilbert

Uno de los principales promotores de esta creencia fue el famoso matemático David Hilbert (1862-1943), quien creía que en matemáticas todo se podía y debía probarse a partir de los axiomas básicos. El resultado de ello sería demostrar de manera concluyente los dos elementos básicos del sistema matemático. En primer lugar, las matemáticas debían ser capaces, al menos en teoría, de responder a cualquier interrogante concreto. En segundo lugar, las matemáticas deberían estar libres de incongruencias; es decir, una vez demostrada la veracidad de una premisa a través de un método, no sería posible mediante otro método concluir que esa misma premisa era falsa. Hilbert estaba convencido de que, asumiendo tan solo unos pocos axiomas, se podría responder a cualquier pregunta matemática concebible sin temor a una contradicción.

El 8 de agosto de 1900, Hilbert pronunció una conferencia histórica en el Congreso Internacional de Matemáticas en París. Allí planteó veintitrés problemas matemáticos sin resolver que él consideraba de una perentoria importancia. Hilbert pretendía sacudir a la comunidad para que lo ayudaran a realizar su sueño de crear un sistema matemático libre de toda duda e incoherencia. Una ambición que inscribió en su lápida:

Wir müssen wissen.

Wir werden wissen.

(Tenemos que saber.

Llegaremos a saber).

Al mismo tiempo, el lógico inglés Bertrand Russell, que también estaba contribuyendo al gran proyecto de Hilbert, había tropezado con una incoherencia. Russell evocó su propia reacción ante la temida posibilidad de que las matemáticas fueran intrínsecamente contradictorias. No había escapatoria a ello. El trabajo de Russell causó un perjuicio considerable al sueño de crear un sistema matemático libre de duda, incoherencia y paradoja. La paradoja de Russell se explica a menudo con el cuento del bibliotecario minucioso.

> *Un día, deambulando entre las estanterías, el bibliotecario descubre una colección de catálogos. Hay diferentes catálogos para novelas, obras de consulta, poesía y demás. Se da cuenta de que algunos de los catálogos se incluyen a sí mismos y otros, en cambio, no.*
>
> *Con el objeto de simplificar el sistema, el bibliotecario elabora dos catálogos más: en uno de ellos hace constar todos los catálogos que se incluyen a sí mismos y en el otro, más interesante aún, todos aquellos que no se catalogan a sí mismos. ¿Debe catalogarse a sí mismo? Si se incluye, por definición no debería estar incluido; en cambio, si no se incluye, debería incluirse por definición. El bibliotecario se encuentra en una situación imposible.*

La incongruencia que atormenta al bibliotecario causará problemas en la estructura lógica de las matemáticas. Las matemáticas no pueden tolerar inconsistencias, paradojas o contradicciones. El poderoso instrumento de la prueba por contradicción, por ejemplo, se fundamenta en una matemática libre de paradojas. La prueba por contradicción establece que si una afirmación conduce al absurdo, entonces tiene que ser falsa; sin embargo, según Russell, incluso los axiomas pueden llevar al absurdo. Así que la prueba por contradicción podría evidenciar que un axioma es falso y, no obstante, los axiomas son el fundamento de las matemáticas y se reconocen como ciertos.

El trabajo de Russell sacudió los cimientos de las matemáticas y arrastró el estudio de la lógica matemática a una situación de caos. Una manera de abordar el problema era crear un axioma adicional que prohibiera a cualquier grupo ser miembro de sí mismo. Eso reduciría la paradoja de Russell y convertiría en superflua la cuestión de si hay que introducir en el catálogo a aquellos otros que no se incluyen a sí mismos.

En 1910, Russell publicó el primero de los tres volúmenes de la obra *Principia Mathematica*, un intento de tratar el problema surgido de su propia paradoja. Cuando Hilbert se retiró en 1930, estaba seguro de que las matemáticas estaban bien encaminadas hacia su recuperación. Al parecer, su sueño de una lógica coherente y lo bastante sólida como para responder a cualquier pregunta se hallaba en vías de tornarse en realidad.

Gödel y sus dos teoremas de la incompletitud

Pero ocurrió que en 1931 un matemático desconocido de veinticinco años hizo público un artículo que destruiría para siempre las esperanzas de Hilbert. Kurt Gödel iba a forzar a los matemáticos a aceptar que las matemáticas nunca llegarían a alcanzar una lógica perfecta, y sus trabajos llevaban implícita la idea de que problemas como el último teorema de Fermat pudieran ser imposibles de resolver.

Gödel había demostrado que el intento de crear un sistema matemático completo y coherente era un imposible. Sus ideas se pueden recoger en dos proposiciones:

- *Primer teorema de indecidibilidad:* Si el conjunto de axiomas de una teoría es coherente, existen teoremas que no se pueden ni probar ni refutar.
- *Segundo teorema de indecibilidad:* No existe ningún proceso constructivo capaz de demostrar que una teoría axiomática es coherente.

El primer enunciado de Gödel dice básicamente que, con independencia de la serie de axiomas que se vaya a utilizar, habrá cuestiones que las matemáticas no puedan resolver; l**a completitud no podrá alcanzarse jamás.**

Peor aún, el segundo enunciado dice que los matemáticos jamás podrán estar seguros de que los axiomas elegidos no los conducirán a ninguna contradicción; **la coherencia no podrá demostrarse jamás.**

Gödel probó que el programa de Hilbert era una tarea imposible. A pesar de que el segundo enunciado de Gödel decía que era imposible demostrar que los axiomas fueran coherentes, eso no implicaba que fueran incoherentes. Muchos años después, el gran teórico de números André Weil dijo:

> *Dios existe porque las matemáticas son coherentes y el diablo existe porque no podemos demostrarlo.*

La demostración de los teoremas de Gödel es tremendamente complicada, pero se puede ilustrar con una analogía lógica que debemos a Epiménides, la cual se conoce como la paradoja cretense. Epiménides fue un cretense que afirmó:

Soy un mentiroso.

La paradoja surge cuando intentamos determinar si esta proposición es verdadera o falsa. Si la proposición es cierta, en principio afirmamos que Epiménides no es un mentiroso. Si la proposición es falsa, entonces Epiménides no es un mentiroso, pero hemos aceptado que emitió un enunciado falso y, por lo tanto, sí que es un mentiroso. Es decir, encontramos otra incoherencia: el enunciado no es ni verdadero ni falso.

Gödel dio una reinterpretación a la paradoja del mentiroso y le incorporó el concepto de demostración. El resultado fue un enunciado como el que sigue:

Este enunciado no tiene demostración.

Puesto que Gödel consiguió traducir tal proposición a una notación matemática, fue capaz de demostrar que hay enunciados matemáticos ciertos que jamás podrán probarse como tales; son los denominados *enunciados indecidibles*. Este fue el golpe mortal para el programa de Hilbert.

Esto mostró para Hilbert que el **Entscheidungsproblem** no es computable. Es decir, **no existe un algoritmo** del tipo que buscaba Hilbert. Un cínico podría decir que los matemáticos dieron un suspiro de alivio, porque si hubiera tal algoritmo, todos se quedarían sin trabajo en cuanto se encontrara. Sin embargo, a los matemáticos les sorprendió este notable descubrimiento.

Consecuencias laterales del teorema de Gödel

El teorema de Gödel no hizo obsoletos o inútiles a los lenguajes formales; por el contrario, hizo de ellos un mejor instrumento para el pensamiento, puesto que los depuró de un error catastrófico: el suponerlos perfectos; esta suposición, de forma mal disimulada, equivale a suponer perfecta la razón humana.

El teorema de Gödel de ninguna manera invalida al pensamiento científico ni a la racionalidad en general; lo que demuele es la pretensión de absoluta superioridad que se les concede sobre cualquier otra forma de pensamiento por considerar que son infalibles, o que si cometen errores estos tarde o temprano metódicamente serán corregidos. Hoy sabemos gracias a Gödel que el error y la incertidumbre son inherentes, consustanciales a nuestra capacidad de razonar y de percibir, que

estructuralmente se encuentran incorporadas estas características a nuestra condición humana.

Sabemos que la verdad y la certeza son categorías religiosas y escatológicas a las que no tendremos acceso en el transcurso de nuestras vidas; Kurt Gödel demostró que así operan nuestros sistemas formales de conocimiento. Werner Heisenberg, con su principio de incertidumbre, demostró que nuestra capacidad de percibir los fenómenos de la naturaleza tiene el mismo defecto.

Si se entiende que la ciencia es el camino por el cual el ser humano adquiere la verdad y la certeza, y conoce al mundo objetivamente, tal como él es, entonces el teorema de Gödel demolió a la ciencia y la colocó a la altura de cualquier superchería. Si se entiende a la ciencia como todo proceso o actividad relacionada con el ser humano, no incluye —gracias a su naturaleza estructural y metódica— garantía alguna de certeza, veracidad, racionalidad y objetividad, y que necesariamente se encuentra tan expuesta al error, la incertidumbre, la irracionalidad y la subjetividad como cualquier otra forma de pensamiento. En consecuencia, la ciencia no queda invalidada; al contrario, se evidencia como el mejor camino conocido para eliminar errores —sin que por ello se pretenda eliminar a todos los errores—, como la única disciplina que reconoce por principio que necesita cometer errores, subjetividades y actos irracionales y que, también necesariamente, habrá de aprender a convivir con la paradoja. Esta es la única forma de conocer nuestra mente y nuestra realidad que hoy por hoy tiene el ser humano.

Otra consecuencia importante que sigue del teorema de Gödel ha sido el haber marcado una frontera entre la inteligencia artificial y la inteligencia humana, a saber, la capacidad de pensar y razonar con paradojas. Todos los sistemas de inteligencia artificial que hoy se conocen operan exclusivamente con proposiciones; pero en cuanto se cuela inadvertidamente en la lógica del sistema una paradoja disfrazada de preposición, el sistema se paraliza o se queda trabajando en forma circular (son sistemas semidecidibles). La mente humana, en cambio, ante la paradoja, crece y desarrolla su ingenio, creando formas superiores de pensamiento. La mecánica cuántica, la mecánica relativista, así como las grandes teologías asociadas con religiones universales —no se diga el arte— son las mejores pruebas de la capacidad de crear que tiene la mente humana ante la paradoja (Fregoso, 1997).

El Entscheidungsproblem

El *Entscheidungsproblem* (en castellano, *problema de decisión*) fue el reto en lógica simbólica de encontrar un algoritmo general que decidiera si una fórmula del cálculo de primer orden es un teorema. En 1936, de manera independiente, Alonzo Church y Alan Turing demostraron que era imposible escribir tal algoritmo. En consecuencia, sería también imposible decidir con un algoritmo si ciertas frases concretas de la aritmética eran ciertas o falsas. Así se originó la teoría de la computabilidad, que estudia bajo qué condiciones un problema matemático es resoluble algorítmicamente.

El *Entscheidungsproblem* se remonta a Gottfried Leibniz, quien en el siglo XVII, luego de construir exitosamente una máquina mecánica de cálculo, soñaba con construir una máquina que pudiera manipular símbolos para determinar si una frase en matemáticas es un teorema. Para ello, lo prime-

ro que sería indispensable sería un lenguaje formal claro y preciso, por lo que mucho de su trabajo posterior se dirigió hacia ese objetivo. En 1928, David Hilbert y Wilhelm Ackermann propusieron la pregunta en su formulación anteriormente mencionada.

Una fórmula lógica de primer orden es llamada *universalmente válida o lógicamente válida* si se deduce de los axiomas del cálculo de primer orden. El teorema de completitud de Gödel establece que una fórmula lógica es universalmente válida en este sentido si y solo si es cierta en toda interpretación de la fórmula en un modelo.

Antes de poder responder a esta pregunta, hubo que definir formalmente la noción de *algoritmo*. Esto fue realizado por Alonzo Church en 1936 con el concepto de *calculabilidad efectiva*, basado en su cálculo lambda, y por Alan Turing, basándose en la máquina de Turing. Los dos enfoques son equivalentes, en el sentido de que con ellos se pueden resolver exactamente los mismos problemas.

La respuesta negativa al *Entscheidungsproblem* fue dada por Alonzo Church en 1936 e independientemente después (ese mismo año) por Alan Turing. Church demostró que no existe *algoritmo* (definido según las funciones recursivas) que decida para dos expresiones del cálculo lambda si son equivalentes o no. Church para esto se basó en el trabajo previo de Stephen Kleene. Por otra parte, Turing redujo este problema al problema de la parada para las máquinas de Turing. Generalmente se considera que la prueba de Turing ha tenido más influencia que la de Church. Ambos trabajos se vieron influidos por trabajos anteriores de Kurt Gödel sobre el teorema de incompletitud, especialmente por el método de asignar números a las fórmulas lógicas para poder reducir la lógica a la aritmética.

El argumento de Turing es como sigue: supóngase que se tiene un algoritmo general de decisión para la lógica de primer orden. Se puede traducir la pregunta sobre si una máquina de Turing termina con una fórmula de primer orden, que entonces podría ser sometida al algoritmo de decisión. Pero Turing ya había demostrado que no existe algoritmo general que pueda decidir si una máquina de Turing se para.

Es importante notar que si se restringe el problema a una teoría de primer orden específico con predicados, constantes y axiomas, es posible que exista un algoritmo de decisión para la teoría. Algunos ejemplos de teorías decidibles son la aritmética de Presburger y los sistemas estáticos de tipos de los lenguajes de programación.

Sin embargo, la teoría general de primer orden para los números naturales, conocida como la *aritmética de Peano*, no puede ser decidida con ese tipo de algoritmo. Esto se deduce del argumento de Turing resumido más arriba.

Además, el teorema de Gödel mostró que no existe algoritmo cuya entrada pueda ser cualquier proposición acerca de los enteros y cuya salida es o no verdadera. Siguiendo de cerca a Gödel, otros matemáticos —como Alonzo Church, Sephen Kleene, Emil Post y Alan Turing— encontraron más problemas que carecían de solución algorítmica. Tal vez la característica más notable de estos primeros resultados sobre problemas que no se pueden resolver por medio de computadoras es que se obtuvieron en la década de 1930: ¡antes de que se hubiera construido la primera computadora!

TESIS CHURCH–TURING

Existe un obstáculo importante al probar que no existe un algoritmo para una tarea específica. Primero es necesario saber con exactitud qué significa *algoritmo*. Cada uno de los matemáticos mencionados en la sección anterior había superado este obstáculo definiendo dicho vocablo de forma diferente:

- Gödel definió un algoritmo como una secuencia de reglas para formar funciones matemáticas complicadas a partir de funciones matemáticas más simples.
- Church utilizó un formalismo denominado *cálculo lambda*.
- Turing empleó una máquina hipotética conocida como *máquina de Turing*. Turing definió un algoritmo como cualquier conjunto de instrucciones para su máquina simple (Dasgupta, Papadimitriou y Vazirani, 2008).

Estas definiciones —en apariencia diferentes y creadas de manera independiente— resultan ser equivalentes. Conforme los investigadores se dieron cada vez más cuenta de esta equivalencia en la década de 1930, se creyó en forma amplia en las dos proposiciones siguientes:

1. Todas las definiciones razonables de *algoritmo* conocidas hasta el momento son equivalentes.
2. Cualquier definición razonable de *algoritmo* que se llegue a dar, a su vez será equivalente a las definiciones ya conocidas.

Estas creencias han llegado a denominarse *tesis de Church-Turing* en honor a dos de los primeros trabajadores que se dieron cuenta de la naturaleza fundamental del concepto que habían definido. Hasta el momento no ha existido evidencia en contra y se acepta ampliamente la tesis de Church-Turing.

En un planteamiento moderno, es posible definir *algoritmo* como cualquier cosa que pueda ejecutarse en una computadora. Dadas dos computadoras modernas, es posible escribir un programa para una de ellas que pueda comprender y ejecutarse en otra.

La equivalencia entre toda computadora moderna, así como entre la máquina de Turing con otros numerosos medios de definir *algoritmo*, es una evidencia más de la tesis de Church-Turing. Esta propiedad de los algoritmos se conoce como *universalidad*.

En términos informales, universalidad significa que cualquier computadora es equivalente a todas las otras en el sentido de que todas pueden efectuar las mismas tareas.

La conferencia de Gibbs

Aunque después de 1950 publicó muy poco, no por eso Gödel dejó de pensar y escribir. De hecho, al momento de su muerte había dejado un número impresionante de manuscritos inéditos, dedicados principalmente a la filosofía y a la teología, con investigaciones, entre otros temas, sobre la existencia de Dios, la transmigración de las almas o el análisis de los trabajos filosóficos de Gottfried Leibniz.

Entre estos papeles inéditos se destaca el texto *Conferencia de Gibbs* del 26 de diciembre de 1951. En los años siguientes, Gödel se dedicó a corregir y retocar el manuscrito con la intención de publicarlo, sin embargo, nunca logró darle una forma que fuera para él satisfactoria. Finalmente, fue publicado en 1994 como parte de un volumen titulado *Kurt Gödel*, *ensayos inéditos.*

¿Por qué es tan interesante la conferencia de Gibbs? Porque en ella Gödel analizó profundamente lo que para él eran las consecuencias filosóficas de sus teoremas de incompletitud. En concreto, Gödel sostuvo en esa conferencia que sus teoremas demostraban que el platonismo matemático era la postura correcta en la filosofía de las matemáticas.

¿Qué es el platonismo? La pregunta en realidad se divide en varias: ¿la matemática se crea o se descubre? ¿Es una creación humana, de la misma forma que lo es la música y la literatura? ¿O, por el contrario, los matemáticos descubren hechos que existen en una realidad externa preexistente a ellos? El platonismo sostiene que los objetos matemáticos tienen una existencia objetiva y que el trabajo de los matemáticos consiste en descubrir las características de esos objetos. El nombre proviene de Platón, quien afirmaba que nuestras percepciones son el reflejo deformado de una realidad superior que vive en el “mundo de las ideas”.

La postura opuesta recibe el nombre *formalismo* y recoge parte de las ideas del intuicionismo y del programa de Hilbert; este sostiene que la matemática es simplemente una creación humana, similar a la música. La matemática es esencialmente un juego lingüístico (un juego sintáctico) en el que hay ciertos puntos de partida, que son los axiomas, y ciertas reglas lógicas que permiten operar a partir de ellos. El trabajo del matemático constaría en descubrir hacia dónde nos llevan las reglas del juego

John D, Barrow, un filósofo de las matemáticas contemporáneo, ha escrito: “Los matemáticos son formalistas de lunes a viernes y platonistas los fines de semana”. Es decir, para el trabajo diario, la postura formalista es la más conveniente, porque en última instancia toda la verdad descansa en axiomas cuya elección no necesita ulteriores justificaciones (en el formalismo solo se requiere que los axiomas sean consistentes, no que reflejen una verdad externa). Sin embargo, los fines de semana, cuando están relajados, los matemáticos sienten en su fuero interno que trabajan con “objetos de verdad”, cuya existencia es independiente y real (signifique esto lo que signifique).

Las consecuencias de los teoremas de Gödel

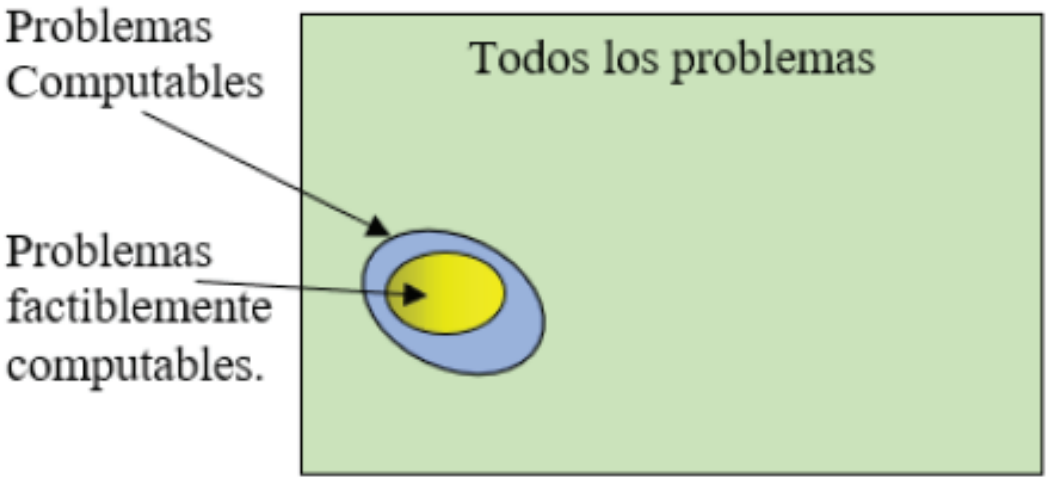
No existe un algoritmo que pueda verificar en todos los casos la verdad o falsedad de un enunciado aritmético. En otras palabras, **jamás se podrá programar a una computadora de modo que pueda demostrar todas las conjeturas de la aritmética** (se trata de una limitación esencial que los avances tecnológicos no podrán superar); de hecho, las computadoras jamás superarán a los matemáticos (aunque, como se verá más adelante, tampoco queda claro que los matemáticos sean siempre capaces de superar a las computadoras).

Computabilidad y complejidad

El estudio de la computabilidad lleva a comprender cuáles son los problemas que admiten solución algorítmica y cuáles no. De aquellos problemas para los que existen algoritmos, también resulta de interés saber cuántos recursos de cómputo se necesitan para su ejecución. Solo los algoritmos que utilizan una cantidad factible de recursos resultan útiles en la práctica. El campo de la ciencia de la computación denominado *teoría de la complejidad* es el que pregunta e intenta resolver cuestiones acerca del empleo de recursos de cómputo.

En la figura 8.1 se muestra una representación pictórica del universo de problemas. Aquellos que pueden computarse en forma algorítmica forman un subconjunto infinitesimalmente pequeño. Los que son *factiblemente computables*, tomando en cuenta sus necesidades de recursos, comprenden una diminuta porción del ya infinitesimalmente pequeño subconjunto. Sin embargo, la clase de problemas factibles computables es tan grande que la ciencia de la computación se ha vuelto una ciencia interesante, práctica y floreciente.

Figura 8.1. Problemas computables y no computables



Tesis de computabilidad secuencial

En la tabla 1.4 se describió la diferencia entre algoritmos que utilizaban cantidades polinomiales (n^c) y exponencial (c^n). Los algoritmos polinomiales tienden a ser factibles para tamaños razonables de datos de entrada. Los algoritmos exponenciales tienden a exceder los recursos disponibles aun tratándose de cantidades pequeñas de datos de entrada. Uno de los objetivos de la teoría de complejidad es mejorar esta clasificación de los algoritmos y, por ende, la comprensión de la diferencia entre problemas factibles y no factibles.

Si un algoritmo se construye tomando dos algoritmos factibles y colocándolos en forma secuencial uno después del otro, el algoritmo así construido debe ser factible. De manera semejante, si algún algoritmo factible se reemplaza por una llamada a un módulo que representa a un segundo algoritmo factible, el nuevo algoritmo combinado también debe ser factible. Esta propiedad de cerradura en realidad se cumple para algoritmos de tiempo polinomial.

Cualquier algoritmo que se ejecuta en tiempo polinomial en una computadora puede correr en tiempo polinomial en cualquier otra. De ahí que tenga sentido hablar de algoritmos de tiempo polinomial en forma independiente de cualquier computadora específica. Una teoría de algoritmos factibles basada en el tiempo polinomial es independiente de la máquina.

La creencia de que todas las computadoras secuenciales razonables que se llegan a crear tienen tiempos de ejecución relacionados polinomialmente recibe el nombre de *tesis de computación* secuencial. Esta puede compararse con la tesis de Curch-Turing. Es una versión más fuerte de esta, pues afirma no solo que todos los problemas computables son los mismos para todas las computadoras, sino también que todos los problemas computables factibles son los mismos para todas las computadoras.

Problemas NP

Esta sección contiene lo que tal vez fue el desarrollo más importante en investigación de algoritmos en la década de 1970, no solo en ciencias de la computación, sino también en ingeniería eléctrica, en investigación de operaciones y en otras áreas relacionadas.

Una idea importante es la distinción entre un grupo de problemas cuya solución se obtiene en tiempo polinomial y un segundo grupo de problemas cuya solución no se obtiene en tiempo polinomial.

Definición formal de NP

Un problema de decisión £ está en NP si y solo si existe un polinomio P y una máquina de Turing M_u, tales que para toda cadena posible de entrada X de longitud n, existe una cadena u cuya longitud es como máximo $p(n)$ y $M_u(x) = 1$, es decir, la máquina de Turing definida por u verifica afirmativamente el problema £. Esta cadena u se conoce como el certificado de £. En términos más profanos, u es el programa que verifica afirmativamente a £ (Areán Álvarez, 2014).

Estos problemas de decisión en los que la solución parece difícil, pero verificarla parece sencillo, forman su propia clase de complejidad, conocida como NP. Se ha dicho *parecen*, porque, como se ha comentado, no se sabe hoy en día si realmente son tan difíciles de resolver.

Como primera observación, se recalcará que en NP solo hay problemas de decisión. Así, en NP está la versión de decisión de TSP, pero no el problema de función más general de encontrar la ruta óptima. La segunda observación es que la búsqueda del certificado es un algoritmo. No el algoritmo que resuelve el problema, sino el que lo verifica. Finalmente, es importante hacer notar que el tiempo para dar por bueno el certificado, el tiempo de ejecución de este algoritmo, es polinómico con respecto a la cadena de entrada. Pero ¿cuál sería un posible certificado para TSP como problema de decisión? Claramente, el camino hamiltoniano.

Cook afirma que es trivial mostrar que P C NP y también afirma que hay dos formas de establecer si un problema pertenece a la clase NP:

- Si su **solución** implica el uso de una máquina de Turing no determinística en tiempo polinomial,
- o si su solución se puede **verificar** en una máquina de Turing determinística en tiempo polinomial.

Es posible demostrar que un problema pertenece a la clase P mediante dos formas:

- mostrando un algoritmo polinomial que lo resuelve,
- o usando una transformación polinomial a otro problema que ya se sabe que está en la clase P.

Una transformación polinomial de B en C es un algoritmo determinista que transforma instancias de b $\in$ B en instancias de c $\in$ C, tales que la respuesta a c es positiva si y solo si la respuesta a b lo es.

P puede ser un subconjunto propio de *NP o P* puede ser igual a *NP*. Es algo que no se sabe. De hecho, es el problema abierto más importante de la teoría de la complejidad computacional y uno de los más importantes de la matemática:

$$¿P = NP? \text{ o } ¿P \neq NP?$$

La inmensa mayoría de los expertos piensa que P ≠ NP, aunque no se ha podido demostrar. La importancia de la clase de problemas NP es que contiene muchos problemas de búsqueda y de optimización para los que se desea saber si existe cierta solución o si hay una mejor que las conocidas.

En esta clase se encuentra el problema del agente viajero, donde, como ya se mencionó, se quiere saber si existe una ruta óptima que pasa por todos los nodos requeridos sin repetirlos. Definición:

> *Un problema de decisión C es NP-Completo si es un problema NP y todo problema de NP se puede transformar polinomialmente en él. Como consecuencia de esta definición, si se tuviera un algoritmo polinomial para el problema C, se tendría una solución en la clase P para todos los problemas de NP.*

Esta definición fue propuesta por Stephen Cook en 1971. Cook demostró (teorema de Cook) que el problema de satisfacibilidad booleana es NP-completo. Desde entonces se ha demostrado que miles de otros problemas pertenecen a esta clase, casi siempre por reducción a partir de otros problemas para los que ya se había demostrado su pertenencia a NP-completo.

La teoría de NP-completo no provee algoritmos para resolver los problemas de este grupo en tiempo polinomial; tampoco dice que no exista algún algoritmo en tiempo polinomial. En lugar de eso, se explicará que todo aquel problema que no tiene en este momento un algoritmo en tiempo polinomial está computacionalmente relacionado. En realidad, se pueden establecer dos clases de problemas. Estos serán los problemas NP-difícil y los NP-completos. Un problema que es NP-completo tendrá la propiedad de que se puede resolver en tiempo polinomial si y solo si todos los demás NP-completo también se puede resolver en tiempo polinomial. Si un problema NP-difícil se puede resolver en tiempo polinomial, entonces todos los problemas NP-completos se pueden resolver en tiempo polinomial.

Cuando se prueba que un problema de optimización combinatoria en su versión *problema de decisión* (combinatorio binario) pertenece a la clase NP-completa, entonces la versión de optimización es NP-difícil.

Mientras que se definen varios problemas con la propiedad de ser clasificados como NP-difícil o NP-completos (problemas que no se resuelven en forma secuencial en tiempo polinomial), estos mismos problemas se pueden resolver en máquinas no determinísticas en tiempo polinomial.

Algoritmos no determinísticos

Hasta este momento, los algoritmos que se han explicado tienen la propiedad de que el resultado de toda operación es único y bien definido. Algoritmos con esta propiedad se conocen como algoritmos determinísticos, los cuales se pueden ejecutar sin problema en una computadora secuencial. En una computadora teórica, se puede remover esta restricción y permitir operaciones cuya salida no es única, pero limitada a un conjunto de restricciones. A la máquina que ejecute tales operaciones se le permitiría escoger alguno de los resultados sujeta a terminar bajo una condición especifica. Esto conduce al concepto de un algoritmo no determinístico. De esta forma, se definen una nueva función y dos nuevas declaraciones:

a. Choice(S)... en forma arbitraria se escoge un elemento del conjunto S.

b. Failure... señala una terminación sin éxito.

c. Success... señala una terminación con éxito.

X ⟵ *choice (1: n)* puede resultar en X la asignación de un entero en el rango *[1, n]*. No existe una regla especifica de cómo se escogió el número entero. La señal de éxito o no éxito se utiliza para definir un estado del algoritmo. Estas declaraciones son equivalentes a definir un stop y no son utilizadas para efectuar un retorno. Un algoritmo termina sin éxito si y solo si no existe un conjunto de elecciones que conduzcan al éxito.

Los tiempos de cómputo para *choice, success y failure* son tomados como O (1). Una máquina que sea capaz de ejecutar un algoritmo no determinístico se conoce como *máquina no determinística.* Mientras no exista una máquina no determinística (como la definida en este escrito), se tienen las suficientes razones intuitivas para concluir que ciertos problemas no se pueden resolver en algoritmos determinísticos en tiempo polinomial.

Por ejemplo: considere el problema de buscar para un elemento x en un conjunto dado de elementos *A(1: n), n ≥1*. Se requiere determinar el índice tal que *A(j) = x o j = 0* si x no se encuentra en A. Un algoritmo no determinístico es este:

```
j ⟵ choice (1: n)
if A(j) =x entonces
imprime(j)
success
endif
else
print ('0');
failure
```

En este ejemplo, una computadora no determinística imprimirá un cero si y solo si no existe algún j tal que A(j) = x. El algoritmo es no determinístico con complejidad O(1). Note que como A no está ordenado, cualquier algoritmo determinístico de búsqueda tiene una complejidad Ω(n).

Una interpretación de un algoritmo no determinístico puede ser permitida utilizando una computadora paralela sin límites. Se puede hacer en cada instante cada *choice*(S), el algoritmo realiza varias copias de él mismo. Una copia para cada *choice*(S), por lo que todas las copias son ejecutadas al mismo tiempo. La primera copia que termine con un sucessful obliga que terminen las demás co-

pias. Si una copia termina en failure, solo ella se detiene. Es importante indicar que una máquina no determinística no produce ninguna copia de algún algoritmo cada vez que un *choice*(S) sea ejecutado. Ya que la máquina es ficticia, no es necesario explicar cómo tal máquina determina si existe un success o un failure. Otra definición de P y NP es esta:

- P es el conjunto de todos los problemas resoluble por un algoritmo determinístico en tiempo polinomial.
- NP es el conjunto de todos los algoritmos de decisión resolubles por un algoritmo no determinístico en tiempo polinomial.

Ya que algoritmos determinísticos son un caso especial de algoritmos no determinísticos, se puede concluir que P $\subseteq$ NP.

Considerando este problema, Cook formuló la siguiente pregunta: ¿hay algún problema en NP único tal que si se lo mostró estar en P, entonces esto implicaría que P = NP? Cook respondió su propia pregunta con el siguiente teorema:

Teorema de Cook: satisfacibilidad es en P si y solo si P = NP.

Como paréntesis se puede decir que en teoría de la complejidad computacional, el problema de satisfacibilidad booleana (también llamado SAT) fue el primero identificado como perteneciente a la clase de complejidad NP-completo.

Stephen Cook demostró dicha pertenencia en 1971 utilizando una máquina de Turing no determinista (MTND) a través de la siguiente demostración:

> *Si se tiene un problema NP, entonces existe una MTND que lo resuelve en tiempo polinomial conocido p(n). Si se transforma esa máquina en un problema de satisfacibilidad (en un tiempo polinomial n) y se soluciona dicho problema, se habrá obtenido también la solución al problema NP original. En tal caso, se habrá demostrado que todo problema NP se puede transformar a un problema de satisfacibilidad y, por lo tanto, el SAT es NP-completo.*

El problema de satisfacibilidad booleana

- Un problema es satisfacible si existe al menos una asignación de valores a las variables del problema que lo hagan verdadero ($\top$).
- Un problema es insatisfacible si todas las posibles asignaciones de valores hacen el problema siempre falso ($\bot$).

Veamos esto con un ejemplo:

- Se va a partir de la siguiente proposición en forma normal disyuntiva:

$$(x_1 \vee \bar{x}_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3)$$

- Se realiza la siguiente asignación:

$$x_1 = \bot, x_2 = \bot \text{ y } x_3 = \top$$

- Se sustituye en la expresión:

$$(\bot \vee \mathrm{T}) \wedge (\bot \vee \mathrm{T}) \wedge (\bot \vee \bot) \wedge (\mathrm{T} \vee \mathrm{T})$$

- Se evalúa la expresión $\bot$.
- Como no se ha encontrado una solución válida, se hace una nueva asignación:

$$x_1 = \top, x_2 = \top \text{ y } x_3 = \top$$

- Se evalúa la expresión $\top$.

Como se ha encontrado una asignación de valores (modelo) que hacen a la expresión verdadera, se ha demostrado que este problema en concreto es *satisfacible*.

Estas son solo dos de las ocho ($2^{n=3}$) posibles asignaciones. Se puede apreciar que el número de soluciones crece rápidamente al añadir nuevas variables, de ahí que su complejidad computacional sea elevada.

Un algoritmo creado para la resolución de problemas SAT es el siguiente:

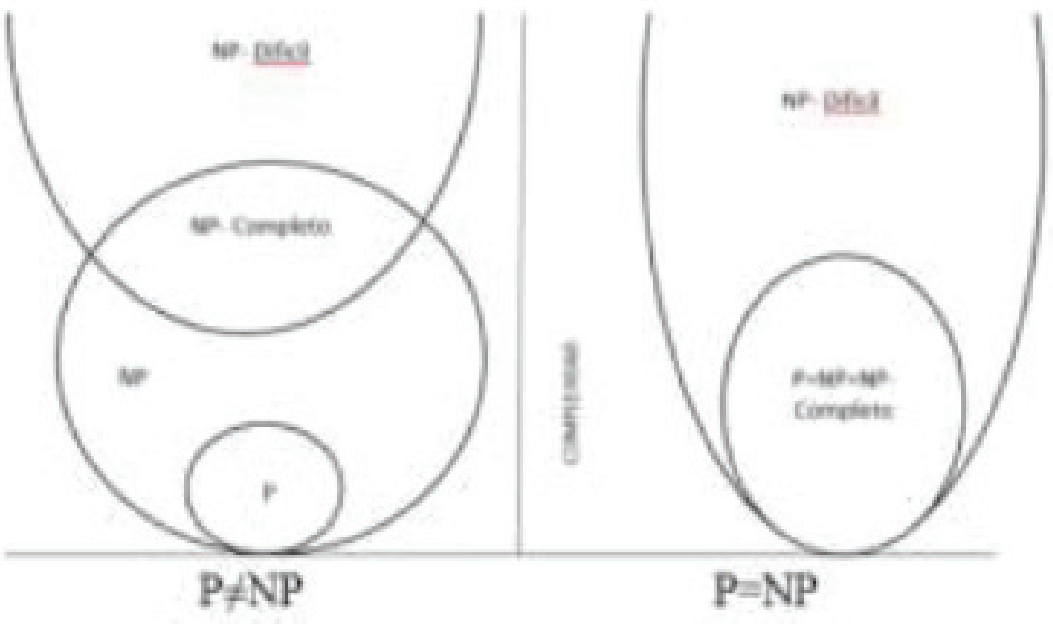
- Algoritmo DPLL: Utiliza una búsqueda hacia atrás sistemática (*backtracking*) para explorar las posibles asignaciones de valores a las variables que hagan al problema satisfacible.

En este momento se puede definir mejor los tipos de problemas NP-difíciles y NP-completo. Primero se definirá la noción de *reducibilidad*. Definición: Sean L_1 y L_2 dos problemas. L_1 reduce a L_2 (L_1 *a* L_2) si y solo si existe un camino para resolver L_1 con un algoritmo polinomial determinístico también se usará un algoritmo determinístico de tiempo polinomial que resuelva a L_2.

Esta definición implica que si existe un algoritmo determinístico en tiempo polinomial para L_2, entonces podemos resolver L_1 en tiempo polinomial. Este operador es transitivo, esto es, si L_1 ϵ L_2 y L_2 ϵ L_3 entonces L_1 *a* L_3.

Un problema $\mathcal{P}$ cualquiera (de función o de decisión) es NP-difícil si y solo si todo problema en NP es reducible a $\mathcal{P}$. Y si además de ser NP-difícil, $\mathcal{P}$ está en NP (es problema de decisión), decimos que es un problema NP-completo. El diagrama de Venn que se muestra en la figura 8.2 aclara la relación de estas clases.

Figura 8.2. Diagrama de Venn que muestra la relación P y NP



Un problema NP-difícil puede no ser NP-completo. Solo un problema de decisión puede ser NP-completo. Sin embargo, un problema de optimización puede ser NP-difícil. Además, si L_1 es un problema de decisión y L_2 es un problema de optimización, es bastante posible que L_1 *a* L_2. Se puede observar que el problema de decisión de la mochila se puede reducir al problema de optimización de la mochila. También se puede comentar que el problema de optimización se puede reducir a su correspondiente problema de decisión. Por lo tanto, problemas de optimización no pueden ser NP-completos, mientras que algunos problemas de decisión pueden ser del tipo NP-duro y no son NP-completos.

Ejemplo de un problema de decisión NP-difícil (NP-hard)

Considere el problema del paro para un algoritmo determinístico. El problema del paro (*the halting problem*) es determinar para un arbitrario algoritmo determinístico A y una entrada I si el algoritmo A con la entrada I termina (o entra en un ciclo infinito). Este problema es indecidible, por lo que no existe algoritmo (de ninguna complejidad) para resolver este problema. Es decir, el problema no es del tipo NP. Para poder mostrar satisfacibilidad *a halting problem*, simplemente se construye un algoritmo A cuya entrada es una formula proposicional X. Si X tiene n variables, entonces A realiza los 2^n posibles asignaciones y verifica si X es satisfacible. Si lo es, entonces A se detiene. Si X no es satisfacible, entonces A entra en un ciclo infinito. Si tenemos un algoritmo en tiempo polinomial para el *halting problem* entonces podemos resolver el problema de la satisfacibilidad en tiempo polinomial usando A y X como entrada del algoritmo para the *halting problem*. Por ello, el problema del paro es NP-difícil, pero no está en NP.

Definición. Dos problemas L_1 y L_2 son polinomialmente equivalentes si y solo si L_1 *a* L_2 y L_2 *a* L_1.

Para mostrar que un problema, L_2, es NP-difícil es adecuado mostrar que L_1 *a* L_2, donde L_1 es algún problema ya conocido como NP-difícil. Ya que *a* es un operador transitivo, se muestra que si satisfacibilidad *a* L_1 y L_1 *a* L_2, entonces satisfacibilidad *a* L_2. Para mostrar que un problema de decisión del tipo NP-difícil es NP-completo, solo se debe de mostrar un algoritmo de tiempo polinomial no determinístico.

Los problemas de la clase NP-difícil (y su subconjunto NP-completo) se encuentran en una gran variedad de disciplinas, como se muestra a continuación:

NP-HARD *graph problems*

- Clique Decision Problem (CDP).
- Node Cover Decision Problem.
- Chromatic Number Decision Problem (CN).
- Directed Hamiltonian Cycle (DHC).
- Traveling Salesperson Decision Problem(TSP).
- AND/OR Graph Decision Problem (AOG).

NP-HARD *scheduling problems*

- Scheduling Identical Processors.
- Flow Shop Scheluling.

NP-HARD *code generation problems*

- Code Generation With Common Subexpressions.
- Implementing Parallel Assignment Instructions (Garey y Johnson, 1975).

La literatura muestra una gran cantidad de problemas NP-difíciles.

Cómo simplificar los problemas NP

Una vez que se mostró el tiempo que se tarda en resolverse cualquier problema L del tipo NP-difícil, nos podríamos inclinar por desechar la posibilidad de que L se pueda resolver en un tiempo polinomial determinístico. En este punto, sin embargo, uno puede realizar la siguiente pregunta: ¿podría uno restringir el problema L a una subclase para poderse resolver en tiempo polinomial determinístico? Se puede observar que colocando las restricciones suficientes sobre un problema NP-difícil (o definiendo una subclase lo suficientemente representativa), se puede llegar a un problema que se pueda resolver en un tiempo polinomial, pero la solución tiene un relajamiento y no es el problema como tal.

Ya que es casi imposible que los problemas NP-difíciles se puedan resolver en tiempo polinomial, es importante determinar cuáles son las restricciones para relajar dentro de las cuales se pueda resolver el problema en tiempo polinomial.

Una posible máquina no determinística

La computación cuántica es un paradigma de computación distinto al de la computación clásica. Se basa en el uso de qubits en lugar de bits, lo que da lugar a nuevas puertas lógicas que hacen posibles nuevos algoritmos. Una misma tarea puede tener diferente complejidad en computación clásica y en computación cuántica, lo que ha dado lugar a una gran expectación, ya que algunos problemas intratables pasan a ser tratables. Mientras que un computador clásico equivale a una máquina de Turing, un computador cuántico equivale a una máquina de Turing cuántica.

En 1985, Deutsch presentó el diseño de la primera máquina cuántica basada en una máquina de Turing. Con este fin enunció una nueva variante de la tesis de Church-Turing, lo que dio lugar al denominado *principio de Church-Turing-Deutsch*.

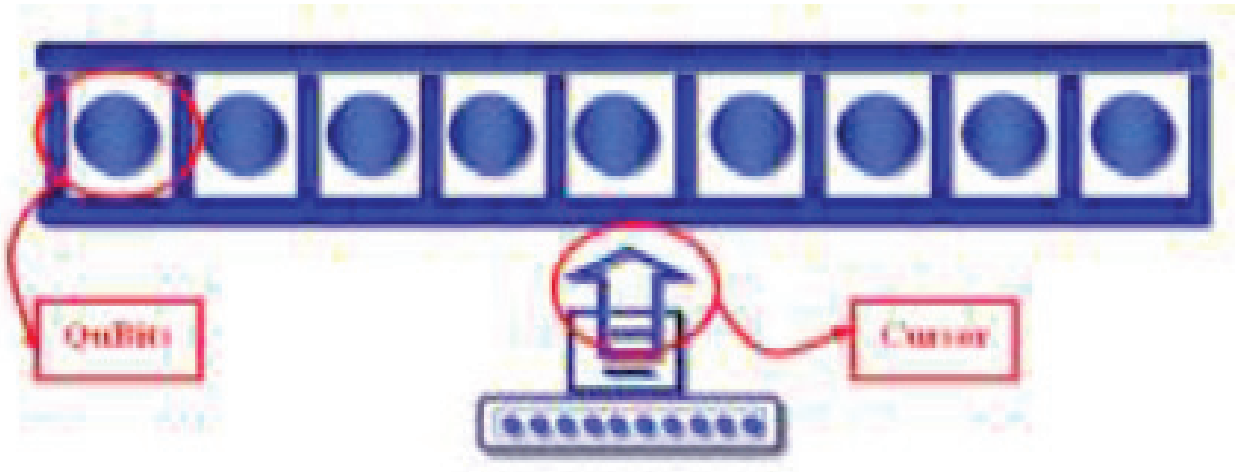
La estructura de una máquina de Turing cuántica es muy similar a la de una máquina de Turing clásica. Está compuesta por los tres elementos clásico.

- una cinta de memoria infinita en donde cada elemento es un qubit,
- un procesador finito y
- un cabezal.

El procesador contiene el juego de instrucciones que se aplica sobre el elemento de la cinta señalado por el cabezal. El resultado dependerá del qubit de la cinta y del estado del procesador. El procesador ejecuta una instrucción por unidad de tiempo.

La cinta de memoria es similar a la de una máquina de Turing tradicional. La única diferencia es que cada elemento de la cinta de la máquina cuántica es un qubit. El alfabeto de esta nueva máquina está formado por el espacio de valores del qubit. La posición del cabezal se representa con una variable entera (figura 8.3).

Figura 8.3. Una máquina de Turing cuántica



UN POCO DE ESPECULACIÓN

Por el lado de los problemas NP

Según ideas recientes de David Deutscth, es posible, en principio, construir una computadora cuántica para la que existen (clases de) problemas que no están en P, pero que podrían ser resueltos por dicho dispositivo en tiempo polinomial. No está claro todavía cómo podría construirse un dispositivo físico confiable que se comporte (confiablemente) como una computadora cuántica —además, la clase particular de problemas considerada hasta ahora es decididamente artificial—, pero subsiste la posibilidad teórica de que un dispositivo físico cuántico mejoraría una máquina de Turing.

¿Sería posible que un cerebro humano —que para nuestro estudio se está considerando como un "dispositivo físico" sorprendentemente sutil, delicado en su diseño, así como complicado— estuviera sacando provecho de la teoría cuántica? ¿Comprendemos el modo en el que podrían ser aprovechados los efectos cuánticos para la solución de problemas y la formación de juicios? ¿Es concebible que tengamos que ir aún más allá de la teoría cuántica de hoy para usar esas ventajas? ¿En verdad los dispositivos físicos pueden mejorar la teoría de la complejidad para máquinas de Turing? ¿Qué sucede con la teoría de la computabilidad para dispositivos físicos reales? Penrose deja una serie de interrogantes que permiten, en cierta forma, unir el procesamiento cerebral con el procesamiento de una computadora cuántica (Penrose, 1989).

Ahora, por el lado de Gödel: en esencia, ¿es la mente humana superior a un ordenador? ¿Nosotros "pensamos", mientras que el ordenador solamente "calcula"? O, por el contrario, no hay una diferencia esencial y algún día el avance tecnológico nos permitirá crear inteligencia artificial, androides, como los que muestra la ciencia ficción, cuyo pensamiento es indistinguible del humano.

La controversia en torno a este tema comenzó a mediados del siglo XX, con el desarrollo de los primeros ordenadores electrónicos, y desde entonces se han escrito decenas, quizá hasta centenares de artículos y libros con argumentos, refutaciones, debates y conjeturas sobre esta cuestión sin que haya hasta este momento respuesta que satisfaga a todos los involucrados.

Por todo lo dicho, es evidente que sería imposible en unas pocas líneas hacer ni siquiera un breve resumen de todos los argumentos a favor o en contra de una u otra postura. Solamente nos interesa mencionar aquí que los teoremas de incompletitud de Gödel han sido usados más de una vez en la discusión, sobre todo como argumento a favor de que la mente humana es esencialmente superior a un ordenador.

La realidad es que no podemos, por lo tanto, vanagloriarnos de superar a los ordenadores, porque jamás podremos tener la certeza de que nuestro razonamiento semántico es correcto (le falló a Fregué, por ejemplo, quien durante años estuvo convencido de la consistencia de sus axiomas, hasta que Bertran Russell descubrió que uno de ellos era autocontradictorio). Debemos aprender a vivir con la incertidumbre de que quizá en el futuro se descubra que todos (o casi todos) nuestros razonamientos son incorrectos.

¿Podría ocurrir tal descubrimiento? ¿Es verosímil esa posibilidad? La verdad es que sí, pues en realidad la discusión iniciada con el descubrimiento de la paradoja de Russell nunca llegó a ser terminada. Las tres propuestas que se hicieron a principio del siglo XX —intuicionismo, logicismo y formalismo (o el programa de Hilbert)— fallaron por diferentes razones y no han sido reemplazadas por otro programa de alcance equivalente. ¿Cuál es la naturaleza de los objetos matemáticos? ¿Existe un nivel intermedio entre el razonamiento puramente sintáctico y los razonamientos libremente semánticos que permitan superar la incompletitud de los teoremas de Gödel asegurando a la vez la consistencia? ¿Existe realmente una diferencia tajante entre "sintáctico" y "semántico", o los que llamamos conceptos semánticos no son más que conceptos sintácticos más sofisticados (en los que trabaja con grupos de símbolos en lugar de símbolos individuales)? (Piñeiro, 2017).

Asimismo, la prueba de Turing incide en uno de los problemas más difíciles de la filosofía: ¿qué es la conciencia? ¿Somos solo un conjunto de algoritmos ejecutados por nuestra mente, lo cual —a su vez— no sería otra cosa que un ordenador muy complejo? ¿O hay algo más en nuestra mente?

Hay muchos filósofos y científicos cognitivos que se inclinan por pensar que la mente es, en efecto, un ordenador cuyo funcionamiento tal vez logremos desentrañar algún día. Es lo que se llama *modelo computacional de la mente*. Otros filósofos se oponen vehementemente a este modelo y atribuyen a la conciencia un papel servidor.

Muy relacionado con esto está el problema del conocimiento: ¿cuándo podemos decir que conocemos algo? A primera vista, hay diversos tipos de conocimiento. La frontera entre lo que conocemos de forma inmediata y lo que conocemos después de un arduo proceso intelectual no está clara. Lo que es aún peor: decir "se deduce inmediatamente que" significa cosas distintas para un gran científico que escribió un libro de su área que para la mayoría de los lectores que son neófitos de tal área o resolver el último teorema de Fermat después de siete años de un arduo proceso intelectual (resuelto por Andrew Willes en 1995). Todavía hay muchas preguntas sin respuesta… afortunadamente.

SECCIÓN IV

Fundamentos matemáticos

TEORÍA DE CONJUNTOS

Un conjunto es una colección de objetos a los cuales se les da el nombre de *elementos*. El contenido de los conjuntos se denota enlistando sus elementos dentro de llaves {}. Por ejemplo:

- *{a, b, c}* este conjunto tiene los elementos *a, b y c*.
- {América, Europa, Asia, África, Oceanía} es el conjunto de continentes del mundo.

En algunos conjuntos es complicado enlistar todos sus elementos debido a que son demasiados, por lo que se utiliza el símbolo "…" que significa *sucesivamente*. Por ejemplo, *{1, 2, ...100}* denota el conjunto de numero enteros del y así sucesivamente hasta el .

Los conjuntos se pueden denotar de una forma "compacta", siguiendo una proposición *p(x)* sobre un universo de discurso; esto es, al conjunto se le conoce "el conjunto *{x \ p (x)}* de todas las 's tal que *p(x)*". Por ejemplo:

- *{x\ 1 ≤ x ≤ 100}* se lee como "el conjunto de todas las tales que son mayores o iguales a 1 y menores o iguales a 100".
- *{p \ q/ q ≠ 0}* se lee como "el conjunto de todos los tales que es diferente de 0".

A los conjuntos se les asignan nombres para identificarlos, normalmente letras mayúsculas; por ejemplo:

- .A= *{a, b, c}*
- .T = *{x / 1 ≤ x ≤ 100}*

A los conjuntos se les puede dar cualquier nombre, pero se recomienda que sea representativo de los elementos del conjunto.

Los conjuntos tienen solamente elementos distintos. Por ejemplo, el conjunto *{a, b, c}* es una representación redundante del conjunto *{a, b, c}*. También, los elementos no tienen un orden definido o establecido. Esto es, los conjuntos *{a, b, c}* y *{a, b, c}* representan la misma colección de elementos.

Para indicar que uno o varios elementos están o pertenecen a algún conjunto se emplea $\in$ el símbolo o la negación $\notin$, es decir, que no pertenece al conjunto. Por ejemplo:

- Sea A = *{a, b, c}*, entonces *a* $\in$ *A* se lee como "*a* pertenece al conjunto A". Por otra parte, d $\notin$ A significa que *d* no es elemento del conjunto A.

El conjunto que no tiene elementos se llama *conjunto vacío* y se denota como $\varnothing = \{\}$. Pueden existir conjuntos que son elementos de algún otro conjunto. Por ejemplo:

- El conjunto A = *{{a, b, c} , d}* contiene los elementos y . Aquí se puede ver que *{a, b, c}* y que también d $\in$ A.
- El conjunto B= *{{a, b} {c, d} a,f}* contiene cuatro elementos *{a, b}, {c,d} a y f* y se puede ver lo siguiente: $\{a, b\} \in B, \{c, d\} \in B, a \in B \text{ y } f \in B.$

Un conjunto A es un subconjunto de B si cada elemento de es también un elemento de . Declarar esto se escribe A c B, que se lee "A es subconjunto de B". Para decir lo contrario se escribe , que se lee " no es un subconjunto de ". Por ejemplo:

- Sean $A = \{a, b, c\} \text{ y } B = \{a, b, c, d, e, f\}$ se puede ver que todos los elementos de están en el conjunto B; por lo tanto A c B. Pero ,B $\not\subset$ A ya que $d, e, f \notin A.$
- Sea el conjunto $A = \{\{a, b, c\}, d\}$, se cumple que $\{\{a, b, c\}\} \subset A$ $\text{y también } \{d\} \subset A.$
- Para el conjunto B= *{{a, b} {c, d} a,f}*, se cumple que $\{\{a, b\}\} \subset B,\ \{\{c, d\}\} \subset B,$ $\{a\} \subset \text{y } \{f\} \subset B.$
- Para el conjunto $C = \{\emptyset\}$ que contiene un elemento que es un conjunto vacío. En este caso se puede ver que se cumple $\emptyset \in C \text{ y } \emptyset \subset C.$

De forma análoga a la desigualdad "$\leq$" está el símbolo "$\subseteq$", es decir A c B significa que el conjunto A es subconjunto o es igual al conjunto B.

Es muy importante mencionar que el conjunto vacío es subconjunto de cualquier conjunto. Sea un conjunto cualquiera, siempre se cumple que $\emptyset \subseteq A$. Pero no siempre el conjunto vacío es un elemento de cualquier conjunto, es decir, no siempre se cumple que $0 \in A$.

La cardinalidad de un conjunto es la cantidad de elementos de un conjunto que se denota con $|.|$. Por ejemplo:

- Sea A = *{a, b, c},* la cardinalidad es $|A| = 3.$
- $|\emptyset| = 0.$

Otra literatura lo marca con #. Por ejemplo:

- Sea A = {a, b, c, d], #A = 4.

El conjunto potencia es el número de subconjuntos que se pueden formar de un conjunto dado. Ejemplo:

- A = {1,2,3}, P(A) = {{},{1},{2},{3},{1,2},{1,3},{2,3},{1,2.3}},

por lo que la cardinalidad de un conjunto potencia se define como $\#P(A) = 2^{\#A}$. Ejemplo:

- A = {1,2,3}, P(A) = {{},{1},{2},{3},{1,2},{1,3},{2,3},{1,2.3}}, $P(A) = 2^{\#A} = 2^3 = 8$.

La unión de dos conjuntos da como resultado un conjunto cuyos elementos están en o en . Por ejemplo:

- $A = \{a, c\}$ y $B = \{b, d\}$ entonces $A \cup B = \{a, b, c, d\}.$
- $A = \{a, c, d\}$ y $B = \{a, b, d\}$ entonces $A \cup B = \{a, b, c, d\}.$

La intersección de dos conjuntos $A \cap B$ da como resultado un conjunto cuyos elementos están en A y en B. Por ejemplo:

- $A = \{a, c\}$ y $B = \{b, d\}$ entonces $A \cap B = \emptyset$. Se dice que la intersección da como resultado el conjunto vacío.
- $A = \{a, c, d\}$ y $B = \{a, b, d\}$, entonces $A \cap B = \{a, d\}$.

Sea un conjunto que represente un espacio de elementos y un conjunto A tal que $A \subset S$ y $A \cap B$ =A , se dice que A´ = S - A es el conjunto complemento de A.

Ejemplos de conjuntos y subconjuntos:

- $\mathbb{N} = \{1,2, \dots\}$ es el conjunto de números naturales.
- $\mathbb{Z} = \{\dots, -2 - 1,0,1,2, \dots\}$ es el conjunto de números enteros.
- $\mathbb{Q} = \left\{\frac{n}{d} | n, d \in \mathbb{Z}, d \neq 0\right\}$ es el conjunto de números racionales.
- $\mathbb{I} = \{e, \pi, \sqrt{2}, \dots\}$ es el conjunto de números irracionales. Son aquellos que expresar como un número racional.
- $\mathbb{R} = \mathbb{Q} \cup \mathbb{I}$ es el conjunto de números reales.
- $\mathbb{C} = \left\{a + bi | a, b \in \mathbb{R}, i = \sqrt{-1}\right\}$ es el conjunto de números complejos.

Se puede ver que entre estos conjuntos se cumple lo siguiente:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

El producto cartesiano se define como $A \times B = \{(a, b) | \forall a \in A, b \in B\}$; en otras palabras, es el conjunto de pares ordenados que se forman tales que para todo elemento *a* en A y *b* en B. Por ejemplo:

- Sean los conjuntos $A = \{1,2\}$ y $B = \{3,4\}$, entonces $A \times B = \{(1,3), (1,4), (2,3), (2,4)$.
  También $B \times A = \{(3,1), (3,2), (4,1), (4,2)\}$.

Ejercicios

1. Sean $S = \{s \in \mathbb{Z} | 0 \leq s \leq 10\}$, $A = \{0,1,2,3,4\}$, $B = \{3,4,5,6\}$ y $C = \{1,3,5\}$, realice las siguientes operaciones:

   a. $A \cup B \cup C$
   b. $A \cup (S - C)$
   c. $A \cap B \cup C$
   d. $A - B$

2. Si A es el conjunto de residentes en México, B es el conjunto de ciudadanos canadienses y C el conjunto de todas las mujeres del mundo, describa con palabras los siguientes conjuntos:

   a. $A \cap C$
   b. $A' \cup C$
   c. $A \cap B \cap C$
   d. $C' - A'$

3. Sea el universo S el conjunto del total de niños que estudian en una escuela, y sean los conjuntos A, B y C, y los conjuntos de niños que estudian matemáticas, física y español, respectivamente. ¿Con que operación de conjuntos se obtienen los siguientes conjuntos de niños?

   a. Que estudian matemáticas y física.

   b. Que no estudian ninguna materia.

   c. Que estudian español o que no estudian física.

   d. Que no estudian matemáticas y que estudian física y que no estudian español.

   e.

4. Determine si los siguientes enunciados son falsos o verdaderos:

   a. $\{a, b\} \subset \{a, b, c, \{a, b, c\}\}$
   b. $\{a, b\} \in \{a, b, c, \{a, b, c\}\}$
   c. $\{\emptyset\} \subset \{\emptyset\}$
   d. $\{\emptyset\} \in \{\emptyset\}$

5. Determine los siguientes conjuntos:

a. $\emptyset \cup \{\emptyset\}$
b. $\emptyset \cap \{\emptyset\}$
c. $\{\emptyset\} \cup \{a, \emptyset, \{\emptyset\}\}$
d. $\{\emptyset\} \cap \{a, \emptyset, \{\emptyset\}\}$
e. $\mathbb{Z} - \mathbb{N}$
f. $\mathbb{Q} - \mathbb{Z}$
g. $\mathbb{Q} - \mathbb{N}$

6. Sean $A = \{a, b, \{a, c\}, \emptyset\}$, determine los siguientes conjuntos:

a. $A - \{a\}$
b. $A - \emptyset$
c. $\{a\} - A$
d. $\emptyset - A$

7. Determine los conjuntos potencia de los conjuntos $\{a\}, \{\{a\}\}$ y $\{\emptyset, \{\emptyset\}\}$.

FUNCIONES

Una función *F* con *dominio X e imagen Z*, escrita *f: X* $\longrightarrow$ *Z* es una correspondencia que a cada x X le asocia o bien ninguno o bien un único punto z $\in$ Z, en el cual se escribe *f(x) = y*. En otras palabras, el dominio es el conjunto de valores para los cuales está definida la función, mientras que la imagen es el conjunto de valores que puede tomar la función.

Ejemplos:

- $f(x) = x^2, f: \mathbb{R} \to [0, \infty)$
- $f(x) = \cos x, f: \mathbb{R} \to [-1,1]$
- $f(x) = \frac{1}{x}, f: \mathbb{R} - \{0\} \to \mathbb{R}$
- $f(x) = \log x, f: (0, \infty) \to \mathbb{R}$

Una función *f* es *monótona si es creciente o decreciente*.

Es creciente si para todo $n_1 \leq n_2$ entonces $f(n_1) \leq f(n_2)$. Por otro lado, es decreciente si para todo $n_1 \leq n_2$ entonces $f(n_1) \geq f(n_2)$ por ejemplo:

- $f(x) = x^2$ es una función creciente.
- $f(x) = 1/x$ es una función decreciente.

Para cada $x \in \mathbb{R}$ se definen las siguientes funciones:

- La función *piso* de *x*, que se denota como $\lfloor x \rfloor$, que es el más grande de los enteros que no superan a *x*.
- La función *techo* de *x*, que se denota como $\lceil x \rceil$, que es el más chico de los enteros que no están por debajo de .

Por ejemplo,

- $\lfloor \pi \rfloor = 3,$
- $\lceil \pi \rceil = 4$
- $\lfloor -\pi \rfloor = -4,$
- $\lceil -\pi \rceil = -3.$

.

Ejercicios

Determine los dominios y las imágenes de las siguientes funciones (también indique cuáles funciones son crecientes o decrecientes):

- $f(x) = \sqrt{-x}$
- $f(x) = e^x$
- $f(x) = \sqrt{\log x}$
- $f(x) = \frac{1}{3^x}$
- $f(x) = \frac{1}{x-1}$
- $f(x) = \frac{2x-3}{x^2+1}$

Obtenga los pisos y techos de los siguientes números:

- e
- $\ln 7$
- $\sqrt{2}$
- $-\frac{1}{3}$
- $-e$
- $\frac{1}{2}$

SUCESIONES Y SERIES

Sucesiones

Una sucesión matemática es una lista ordenada de elementos c_1, c_2, ...c_n llamados *términos*. Por ejemplo, la sucesión de números -1, 1, 7, 17, 31 ..., que se puede abreviar con la expresión $c_n = 2n^2 - 1$, para $n \geq 0$.

Una sucesión aritmética es una lista de números cuya diferencia entre dos términos sucesivos cualesquiera son constantes. Por ejemplo, en la sucesión 3, 5, 7, 9, 11,... la diferencia común es 2; en la sucesión 5, 2, -1, -4,... la diferencia común es -3.

El primero término es c_1; el segundo término se obtiene con $c_2 = c_1 + d$. Para el tercer término queda $c_3 = c_2 + d$, pero como $c_2 = c_1 + d$, entonces $c_3 = c_1 + d + d = c_1 + 2d$. Para el cuarto término queda $c_4 = c_3 + d$, , pero como , entonces . Por inducción se puede establecer la expresión:

$$c_n = c_1 + (n-1)d$$

Una sucesión geométrica es aquella donde un nuevo término se obtiene multiplicando el término anterior por una constante (razón o factor). Por ejemplo, en la sucesión 5,15, 45 135,... la razón es 3; en la sucesión $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, .$ la razón es $\frac{1}{2}$.

El primer término es c_1; el segundo término se obtiene con $c_2 = c_1 r$. Para el tercer término queda $c_3 = c_2 r$, pero como $c_2 = c_1 r$, entonces $c_3 = c_1 r \times r = c_1 r^2$. Para el cuarto término queda , $c_4 = c_3 r$, pero como $c_3 = c_1 r^2$, entonces $c_4 = c_1 r^2 \times r = c_1 r^3$. Por inducción se establece la expresión:

$$c_n = c_1 r^{n-1}$$

Series

Dada una secuencia de números $c_1, c_2, \ldots, c_n$ donde $n \geq 0$, la suma finita se escribe así:

$$\sum_{k=1}^{n} c_k = c_1 + c_2 + \cdots + c_n$$

Si *n* = 0, el valor de la suma por definición es 0. El valor de las series finitas siempre está definido y se puede sumar en cualquier orden.

Por otra parte, la suma infinita de una sucesión infinita de números *c1, c2,* ... se escribe de este modo:

$$\sum_{k=1}^{\infty} c_k$$

En el caso de sumas infinitas, se analiza el límite de la suma para conocer si converge o no, de esta forma:

$$\lim_{n\to\infty} \sum_{k=1}^{n} c_k = \sum_{k=1}^{\infty} c_k$$

Si el límite no existe, se dice que la serie *diverge*; en caso contrario, se dice que *converge*. Los términos de una serie convergente no siempre pueden ser sumados en cualquier orden.

Las series tienen la propiedad de linealidad. Esto es, para cualquier número real *a* y para cualesquiera de las secuencias finitas $c_1, c_2, \ldots, c_n$ y $d_1, d_2, \ldots, d_n$

$$\sum_{k=1}^{n} (\alpha c_k + d_k) = \alpha \sum_{k=1}^{n} c_k + \sum_{k=1}^{n} d_k$$

La propiedad de linealidad también aplica para series infinitas convergentes.

Series aritméticas

La serie aritmética es la suma de los términos de una sucesión aritmética. De la definición, existe una diferencia constante entre los términos sucesivos que se denota como *d*. Entonces, sea la sucesión $c_1, c_2, \ldots c_n$, para determinar la serie se suman los elementos de los extremos. Si $c_n = c_1 + (n-1)d$., se pueden obtener las siguientes igualdades:

$$\begin{aligned}
c_1 + c_n &= c_1 + c_1 + (n-1)d \\
c_2 + c_{n-1} &= c_1 + d + c_1 + (n-2)d \\
c_3 + c_{n-2} &= c_1 + 2d + c_1 + (n-3)d \\
&\vdots \\
c_{\frac{n}{2}} + c_{\frac{n}{2}+1} &= c_1 + \left(\frac{n}{2} - 1\right)d + c_1 + \left(\frac{n}{2} + 1 - 1\right)d
\end{aligned}$$

Reduciendo términos, se obtiene:

$$\begin{aligned} c_1 + c_n &= 2c_1 + (n-1)d \\ c_2 + c_{n-1} &= 2c_1 + (n-1)d \\ c_3 + c_{n-2} &= 2c_1 + (n-1)d \\ &\vdots \\ c_{\frac{n}{2}} + c_{\frac{n}{2}+1} &= 2c_1 + (n-1)d \end{aligned}$$

Se puede ver que la suma de los extremos de los términos de la sucesión da el mismo resultado. Por otra parte, la cantidad de estas sumas es igual a la mitad de los términos de la sucesión, es decir, *n*/2. Entonces, se puede decir lo siguiente:

$$\sum_{k=1}^{n} c_k = (2c_1 + (n-1)d) \times \frac{n}{2}$$

Si sustituimos los valores $c_1 = 1$, $d = 1$ y en la suma, tenemos:

$$(2 + n - 1) \times \frac{n}{2} = \frac{n(n+1)}{2}$$

Además, si $c_1 = 1$, entonces la suma queda $\sum_{k=1}^{n} c_k = \sum_{k=1}^{n} k;$ finalmente la suma de esta sucesión aritmética es:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Por otra parte, las series de cuadrados y cubos son:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$$

Series geométricas

Para cualquier número real $c \neq 1$ la suma se dice ser una serie geométrica si tiene la forma:

$$\sum_{k=0}^{n} c^k = 1 + c + c^2 + \cdots + c^n$$

En otras palabras, es la suma de los términos de una sucesión geométrica. Para calcular la suma de la serie geométrica, se parte de la siguiente identidad algebraica:

$$a^n - b^n = (a - b) \sum_{k=1}^{n} a^{n-k} b^{k-1}$$

Pero si la suma empieza desde cero, es decir, desde *k = 0*, entonces:

$$a^{n+1} - b^{n+1} = (a - b) \sum_{k=0}^{n} a^{n-k} b^k$$

Desarrollando la suma, se tiene:

$$\sum_{k=0}^{n} a^{n-k} b^k = a^n + a^{n-1} b + \cdots + a b^{n-1} + b^n$$

En la serie geométrica, al sustituir se tiene:

$$\sum_{k=0}^{n} c^k = c_1 + c_1 r + c_1 r^2 + \cdots + c_1 r^{n-1}$$

Al factorizar c_1 se obtiene:

$$\sum_{k=0}^{n} c^k = c_1 (1 + r + r^2 + \cdots + r^n)$$

Se puede ver fácilmente que $c_1(1 + r + r^2 + \cdots + r^n) = c_1 \sum_{k=0}^{n} r^k$.

Si sustituimos $a = r$, $b = 1$ y $c_1 = 1$ en la identidad algebraica se tiene:

$$r^{n+1} - 1 = (r - 1) \sum_{k=0}^{n} r^{n-k}$$

Por otra parte, $\sum_{k=0}^{n} r^{n-k} = \sum_{k=0}^{n} r^k$; por lo tanto, tenemos:

$$r^{n+1} - 1 = (r - 1) \sum_{k=0}^{n} r^n$$

Despejando el término de la suma, finalmente llegamos a la siguiente expresión:

$$\sum_{k=0}^{n} r^n = \frac{r^{n+1} - 1}{r - 1}$$

Nótese que si $|r| < 1$, entonces $|r| < 1$, por lo tanto, la expresión queda así:

$$\sum_{k=0}^{\infty} r^n = \frac{-1}{r - 1}$$

Al factorizar el signo negativo del denominador, la suma finalmente es:

$$\sum_{k=0}^{\infty} r^n = \frac{1}{1 - r}$$

Esta última ecuación se cumple si $|r| < 1$, y si la suma es infinita.

Serie telescópica

Sea la sucesión $c_0, c_1, c_2, \ldots, c_n$, entonces $\sum_{i=1}^{n}(c_i - c_{i-1}) = c_n - c_0,$ pues los términos intermedios se anulan. Se dice que esa es una suma telescópica.

Sumas de mismas potencias

Sea $s_{mn} = \sum_{i=1}^{n} i^m$, a suma de las m-ésimas potencias de los primeros *n* + 1 números naturales (contados a partir de cero). De acuerdo con la fórmula del binomio de Newton, tenemos para cada *i* y donde $C_k^m = \frac{m!}{k!(m-k)!}$:

$$(i+1)^m = \sum_{k=0}^{m} C_k^m i^k$$

O sea,

$$(i+1)^m - i^m = \sum_{k=0}^{m-1} C_k^m i^k$$

Al sumar estos valores, desde i = 1 hasta i = n, tenemos una suma telescópica, por tanto:

$$(n+1)^m - 1^m = \sum_{k=0}^{m-1} C_k^m s_{kn}$$

Resulta el sistema de ecuaciones

$$\begin{aligned}
(n+1) - 1 &= s_{0n} \\
(n+1)^2 - 1 &= s_{0n} + 2s_{1n} \\
(n+1)^3 - 1 &= s_{0n} + 3s_{1n} + 3s_{2n} \\
(n+1)^4 - 1 &= s_{0n} + 4s_{1n} + 6s_2n + 4s_{3n} \\
(n+1)^5 - 1 &= s_{0n} + 5s_{1n} + 10s_{2n} + 10s_{3n} + 5s_{4n} \\
&\vdots \quad \vdots \quad \vdots
\end{aligned}$$

Al resolverlo para los primeros valores s_{mn} obtenemos:

$$\begin{aligned}
s_{0n} &= n \\
s_{1n} &= \frac{1}{2}n(n+1) \\
s_{2n} &= \frac{1}{6}n(n+1)(2n+1) \\
s_{3n} &= \frac{1}{4}n^2(n+1)^2 \\
s_{4n} &= \frac{1}{30}n(n+1)(2n+1)(3n^2+3n-1) \\
s_{5n} &= \frac{1}{12}n^2(n+1)^2(2n^2+2n-1) \\
\vdots &\quad \vdots \quad \vdots
\end{aligned}$$

De lo cual se ve que $\sum_{i=1}^{n} i^m = O(n^{m+1})$.

LOGARITMOS

El logaritmo de un número es el exponente al cual hay que elevar la base para obtener dicho número. Esto es, al despejar n de $b^n = x$ se tiene:

Esto se lee como "*n* es igual al logaritmo de base *b* de *x*". Por ejemplo:

- Sea $10^2 = 100$, entonces $\log_{10} 100 = 2$.
- Sea $5^3 = 125$, entonces $\log_5 125 = 3$.
- Sea $e^{-1} = 0.3678$, entonces $\log_e 0.3678 = \ln 0.3678 = -1$.

PROPIEDADES

Sea $n = \log x$ se debe cumplir que $x, n \in \mathbb{R}$ y $x > 0$. Los logaritmos tienen las siguientes propiedades, sin importar la base:

- $\log(x \cdot y) = \log x + \log y$
- $\log\left(\frac{x}{y}\right) = \log x - \log y$
- $\log x^n = n \log x$
- $\log \sqrt[n]{x} = \frac{1}{n} \log x.$

Estas propiedades se cumplen solo cuando la base de los logaritmos en ambos lados de la igualdad es la misma. Los logaritmos también cumplen las siguientes identidades:.

- $\log_b x = \frac{\log_c x}{\log_c b}$, donde $c > 0$ y es un número real.
- $\log_b x = \frac{1}{\log_x b}.$
- $\log_{b^n} x = \frac{1}{n} \log_b x.$
- $y^{\log x} = x^{\log y}$, sin importar la base del logaritmo.
- $\log 1 = 0$, sin importar la base del logaritmo.
- $\log_b b = 1.$

Por ejemplo, reducir los términos empleando las propiedades de los logaritmos:

- $\log_{10} 0.001 = -3$
- $\ln e^{-5} = -5 \ln e = -5$
- $\log_7 2401 = \frac{\log_c 2401}{\log_c 7} = 4$
- $\log_2 \sqrt[4]{8} = \frac{1}{4} \log_2 8 = \frac{3}{4}$
- $\log_3 (27 \times 9) = \log_3 27 + \log_3 9 = 5$
- $\log_5 \frac{3125}{625} = \log_5 3125 - \log_5 625 = 1$
- $\log_{\sqrt{5}} 125 = \frac{1}{1/2} \log_5 125 = 6$
- $3^{\log_2 8} = 8^{\log_2 3} = 27$
- $\log_2 \frac{4^2 \cdot \sqrt[6]{64}}{2^5 \cdot \sqrt[3]{512}}$, de la propiedad de la división, $\log_2 4^2 \cdot \sqrt[6]{64} - \log_2 2^5 \cdot \sqrt[3]{512}$. De aquí se pueden separar en sumas $\log_2 4^2 + \log_2 \sqrt[6]{64} - \left(\log_2 2^5 + \log_2 \sqrt[3]{512}\right)$. Finalmente, se tiene $2\log_2 4 + \frac{1}{6} \log_2 64 - \left(5\log_2 2 + \frac{1}{3} \log_2 512\right)$. Reduciendo términos: $2(2) + \frac{6}{6} - 5(1) - \frac{9}{3} = -3.$

Ejemplos, encontrar el valor de la incógnita:

- $2^{x-1} = 8$, se despeja $x - 1 = \log_2 8 = 3$, finalmente $x = 4$
- $\log_x 81 = 4$, se despeja $81 = x^4$, finalmente $x = 3$

Ejercicios

Calcule los siguientes logaritmos al reducir términos empleando las propiedades de los logaritmos:

- $\log_9 \sqrt[4]{3}$
- $\log_9 \frac{1}{3}$
- $\log_{\frac{1}{2}} \frac{1}{4}$
- $\log_{\sqrt{2}} \frac{1}{4}$
- $\frac{1}{\log_3 \sqrt[3]{3^{-2}}}$
- $\frac{1}{\log_2 \sqrt{2^{-3}}}$
- $\frac{\log_2 8^{\log_3 27}}{\log_4 16}$
- $\frac{\log_5 5 \cdot \sqrt[5]{5}}{\log_3 3 \cdot \sqrt[3]{3}}$

Encuentre el valor de la incógnita de cada ecuación:

- $\frac{\log_{10}(x+1)}{\log_{10}(x-1)} = 2$
- $\log_2 x^3 = 6$
- $\log_{10}(x + 6) = \log_{10}(2x + 1)$
- $\log_{10}(x + 6) = \log_{10}(x - 3) + 1$
- $2^{x+1} = \frac{1}{4^x}$
- $2 \log_{10} x = 3 + \log_{10} \frac{x}{10}$
- $\frac{9^x \cdot 3^x}{3} = 1$
- $2 \log_2 x - \log_2(x + 16) = 2$

ÓRDENES DE CRECIMIENTO

Las notaciones que normalmente describen el tiempo de ejecución asintótico de un algoritmo son definidas en términos de funciones cuyos dominios son el conjunto de número naturales $\mathbb{N} = \{0,1,2,\ldots\}$. Tales notaciones son convenientes para describir el peor caso de tiempo de ejecución t (*n*), el cual usualmente está definido solamente en entradas de enteros. Se utilizará la notación asintótica principalmente para describir los tiempos de ejecución de los algoritmos. La notación asintótica implica funciones.

El orden de crecimiento del tiempo de ejecución de un algoritmo da una caracterización de la eficiencia del algoritmo y nos permite comparar el desempeño relativo de algoritmos alternativos. Cuando buscamos tamaños de entradas lo suficientemente grandes para obtener el orden de crecimiento del tiempo de ejecución relevante, entonces estamos estudiando la eficiencia asintótica de los algoritmos. Esto es, nos importa conocer cómo el tiempo de ejecución de un algoritmo aumenta con el tamaño de la entrada, conforme el tamaño de la entrada aumenta sin ninguna cota o límite. Usualmente, un algoritmo que es asintóticamente más eficiente será la mejor opción para todo excepto para entradas muy pequeñas.

En la siguiente sección se ofrecen varias definiciones de notación asintótica, se presentan diversas convenciones de notación y el comportamiento de funciones que comúnmente aparecen en el análisis de algoritmos.

NOTACIÓN ASINTÓTICA

La notación que se emplea para describir el tiempo de ejecución asintótica de un algoritmo está definida en términos de funciones cuyo dominio y contradominio son los números naturales $\mathbb{N}$ y los números reales $\mathbb{R}$, respectivamente. Tal notación es conveniente para describir el tiempo de ejecución como una función t (*n*), el cual esta usualmente definida por tamaños de entrada de tipo entero. Se puede extender la notación en el dominio de números reales o restringirlo a un subconjunto de números naturales. Sin embargo, hay que asegurarse del significado preciso de la notación para cuando se haga “mal uso” de la notación.

COTA SUPERIOR ASINTÓTICA

Para denotar la cota superior asintótica de una función g (n) se escribe 0 (f (n)) como el conjunto de funciones:

$$O(f(n)) = \{g(n) | \exists c > 0, n_0 \in \mathbb{N}: n \geq n_0 \Rightarrow g(n) \leq cf(n)\}$$

Es decir, *O (g (n))* es el conjunto de funciones con dominio e imagen en los números naturales, tales que existe una constante mayor a cero, donde a partir de un número natural el crecimiento de *g (n)* es menor o igual al de la función *cf (n)*. Dado que se está trabajando con conjuntos la notación correcta debe ser $g(n) \in O(f(n))$. Pero en la literatura y para facilitar el análisis, se emplea el símbolo de igualdad para indicar que la función es un elemento del conjunto. Esto es, que tanto $g(n) \in O(f(n))$ como $g(n) = O(f(n))$ como representan lo mismo.

Por ejemplo, sea g (n) = $5n^2$ - 10n +1, entonces:

- $g(n) = O(n^2)$, ya que $5n^2 - 10n + 1 \leq 6n^2$, para $n \geq 1$.
- $g(n) = O(n^3)$, ya que $5n^2 - 10n + 1 \leq 2n^3$, para $n \geq 1$.
- $g(n) \neq O(n)$, ya que $5n^2 - 10n + 1 \nleq n$, para $n \geq 3$.

Esta notación se emplea para indicar la cota superior de una función, dentro de un factor constante.

COTA INFERIOR ASINTÓTICA

Para denotar la cota inferior asintótica de una función *g (n)* se escribe Ω*(f (n))* como el conjunto de funciones:

$$\Omega(f(n)) = \{g(n) | \exists c > 0, n_0 \in \mathbb{N}: n \geq n_0 \Rightarrow g(n) \geq cf(n)\}$$

Es decir, Ω*g (n)* es el conjunto de funciones con dominio e imagen en los números naturales, tales que existe una constante mayor a cero, donde a partir de un número natural n_0 el crecimiento de *f (n)* es menor o igual al de la función cg *(n)*. Al igual que con el conjunto *O (g (n))*, para facilitar el análisis, se emplea el símbolo de igualdad para indicar que la función es un elemento del conjunto. Esto es, que tanto *g (n)* $\in$ Ω*(f (n))*como g (n) = Ω(f (n)) representan lo mismo.

Por ejemplo, sea *g (n) = 5n² - 10 + 1* , entonces:

- $g(n) = \Omega(n^2)$, ya que $5n^2 - 10n + 1 \geq 2n^2$, para $n \geq 4$.
- $g(n) \neq \Omega(n^3)$, ya que $5n^2 - 10n + 1 \ngeq n^3$, para $n \geq 1$.
- $g(n) = \Omega(n)$, ya que $5n^2 - 10n + 1 \geq 1000000n$, para $n \geq 200002$.

Notación

Esta notación se emplea para denotar al conjunto de funciones cuyo orden de crecimiento es el mismo. Para denotar que una función *g (n)* tiene el mismo orden de crecimiento de un conjunto de funciones se escribe de este modo:

$$\Theta\big(f(n)\big) = \{g(n) | \exists c_1, c_2 > 0, n_0 \in \mathbb{N}: n \geq n_0 \Rightarrow c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$

Esto es significa que g (n) $\in \Theta$ (f (n) si existen constantes positivas c_1 y c_2 tales que la "mantienen" entre c_1f(n) y c_2 f (n) para un suficientemente grande. Al igual que con los conjuntos O (g (n)) y Ω *(n))* para facilitar el análisis, se emplea el símbolo de igualdad para indicar que la función es un elemento del conjunto. Esto es, que tanto g (n) $\in \Theta$ (f (n)) como g (n) = Θ (fn)) representan lo mismo.

Otra forma de establecer que g (n) $\in \Theta$ (f (n)) es que se cumpla la siguiente condición:

$$g(n) \in O(f(n)) \text{ y } g(n) \in \Omega(f(n)).$$

De los ejemplos anteriores donde g (n) = 5n² - 10n +1 se puede decir que:

- $g(n) = \Theta(n^2)$, se cumple que $g(n) \in O(f(n))$ y $g(n) \in \Omega(f(n))$.
- $g(n) \neq \Theta(n^3)$, se cumple que $g(n) \in O(f(n))$ pero $g(n) \notin \Omega(f(n))$.
- $g(n) \neq \Theta(n)$, de manera similar al anterior se cumple $g(n) \in \Omega(f(n))$ pero $g(n) \notin O(f(n))$.

Notación asintótica justa

La notación asintótica *0 (f (n))* puede o no ser asintóticamente justa. Por ejemplo, la cota *3 0 n2n²= (n²)* es asintóticamente justa, pero la cota 3 n² = no lo es. De aquí que se emplea la notación *o (f (n))* para denotar una **cota superior que no es asintóticamente justa.** Formalmente se define *o (f (n))*, pequeña *o* de *f (n)* , de este modo:

$$o\big(f(n)\big) = \{g(n) | \forall c > 0 \exists, n_0 \in \mathbb{N}: n \geq n_0 \Rightarrow g(n) < cf(n)\}$$

La diferencia entre *O (f (n))* y *O (f (n))* es que en *g (n) = 0 (f (n))* la cota *g (n)* $\leq$ *cf (n)* se cumple para alguna constante c > 0. Pero en *g (n) = 0 (f (n))*, la cota *g (n) < cf (n)* se cumple para cualquier constante c > 0.

Esta relación *g (n) = 0 (f (n))* implica lo siguiente:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

De forma análoga, la notación ω(f(n)) es a la notación Ω(f(n)) como lo son la notación o(f(n)) y O(f(n)). Se utiliza la notación ω (f(n)) para denotar una **cota inferior que no** es asintóticamente justa. Formalmente se define ω(f(n)), pequeña omega, como el conjunto:

$$\omega(f(n)) = \{g(n) | \forall c > 0 \exists, n_0 \in \mathbb{N}: n \geq n_0 \Rightarrow cf(n) < g(n)\}$$

Por ejemplo, $n^2 = \omega(n)$**, pero** $n^2 \neq \omega(n^2)$**. La relación** $g(n) = \omega(f(n))$ **implica que**

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$$

Varias de las propiedades de números reales se pueden aplicar a las comparaciones asintóticas. Para cualquier símbolo $\mathbb{O} \in \{O, \Theta, \Omega, o, \omega\}$ vale la implicación de **transitividad,** esto es, si *g(n)=O(f(n)) y f(n)=O(h(n))*, entonces *g(n)=O(h(n))*.

Para cualquier símbolo $\mathbb{F} \in \{O, \Theta, \Omega\}$ vale la relación de **reflexibilidad**, esto es:

$$f(n) = \mathbb{F}(f(n))$$

Las siguientes relaciones son inmediatas:

- $o(g(n)) \subset O(f(n))$
- $\omega(g(n)) \subset \Omega(f(n))$

Notación asintótica en ecuaciones y/o desigualdades

Como se ha visto, la notación asintótica se puede emplear en funciones matemáticas, pero ¿cómo se deben interpretar? Cuando una notación asintótica aparece en una formula, se debe interpretar como una función desconocida. Por ejemplo, la formula n^2-2n+3=n^2+ Θ (n) significa que n^2-2n+3=n^2+g(n), donde g(n) es alguna función en el conjunto Θ (n).

Utilizar la notación asintótica de esta forma facilita eliminar detalles innecesarios en las ecuaciones. *La cantidad de funciones desconocidas en una expresión es igual al número de veces que la notación asintótica aparece.*

Es decir, el operador en la notación asintótica significa concatenación, y no la usual suma algebraica. Por ejemplo, supóngase que se tiene la serie $\sum_{k=1}^{n} \Theta(n) = \Theta(n) + \cdots + \Theta(n),$ aunque se repite *n* veces se podría pensar erróneamente que $\sum_{k=1}^{n} \Theta(n) = \Theta(n^2).$ Sin embargo, y como se menciona anteriormente, el operador significa concatenación, por lo tanto, $\sum_{k=1}^{n} \Theta(n) = \Theta(n).$

En algunos casos, la notación aparece del lado izquierdo de una ecuación, por ejemplo, $7n^2 + \Theta(n) = \Theta(n^2).$

Esto se debe interpretar como "no importa como son escogidas las funciones desconocidas del lado izquierdo de la igualdad; hay una forma de escoger la función desconocida del lado derecho de la igualdad para hacer valida la ecuación".

Por lo tanto, del ejemplo anterior se tiene que para cualquier función $g(n) \in \Theta(n),$ hay una función $f(n) \in \Theta(n^2)$ tal que $7n^2+g(n)=f(n)$ para todo *n*.

Es posible juntar un número de tales relaciones como el siguiente ejemplo:

7n^2-2n+3=7n^2+ Θ (n)= Θ (n^2)

Por ejemplo, comprobar que

- $2^{n+1} = \Theta(2^n).$
- $2^{2n} \neq \Theta(2^n).$

Para comprobar que $2^{n+1} = \Theta(2^n)$ hay que verificar las definiciones de que $2^{n+1} \in O(2^n)$ y $2^{n+1} \in \Omega(2^n).$ Empezamos primero por verificar que $2^{n+1} = O(2^n).$ Entonces tenemos:

$$\begin{aligned} 2^{n+1} &\leq c2^n \\ 2 \cdot 2^n &\leq c2^n \\ 2 &\leq c \end{aligned}$$

Por lo tanto, se cumple que $2^{n+1} \in O(2^n).$ Por otro lado, falta verificar que $2^{n+1} = \Omega(2^n)$. Entonces se tiene:

$$\begin{aligned} 2^n &\leq c2^{n+1} \\ 2^n &\leq 2c2^n \\ 1 &\leq 2c \end{aligned}$$

Se llega a que $c \geq \frac{1}{2}$ y, por lo tanto, $2^{n+1} \in \Omega(2^n)$. De aquí que tanto $2^{n+1} \in O(2^n)$ como $2^{n+1} \in \Omega(2^n)$ se cumplen: se concluye que $2^{n+1} = \Theta(2^n).$

En el segundo ejemplo, para comprobar que $2^{2n} \neq \Theta(2^n),$ de manera similar al anterior ejemplo, empezamos por verificar que $2^{2n} \neq \Theta(2^n),$ Entonces tenemos:

$$\begin{aligned} 2^{2n} &\leq c2^n \\ (2^n)^2 &\leq c2^n \\ 2^n &\leq c \end{aligned}$$

Es claro que $2^{2n} \notin O(2^n)$, por lo tanto, $2^{2n} \neq \Theta(2^n)$. Por otra parte, se cumple que , ya que $2^{2n} = \Omega(2^n)$,

$$\begin{aligned} 2^n &\leq c2^{2n} \\ 2^n &\leq c(2^n)^2 \\ 1 &\leq c2^n \end{aligned}$$

Si tomamos c = 1, entonces queda la desigualdad 1 $\leq 2^n$, por lo tanto, se cumple que $2^{2n} \in \Omega(2^n)$.

En la reducción de expresiones algebraicas es importante tener presentes las siguientes observaciones. Para cualquier símbolo $\mathbb{F} \in \{O, \Theta, \Omega\}$ vale lo siguiente:

- $\mathbb{F}(g(n)) + \mathbb{F}(f(n)) = \mathbb{F}(\max(g(n), f(n)))$
- $\mathbb{F}(c \cdot g(n)) = \mathbb{F}(g(n))$ para $c > 0$
- $\mathbb{F}(g(n)) \times \mathbb{F}(f(n)) = \mathbb{F}(g(n) \times f(n))$ en general se cumple, aunque es importante seleccionar la función que domine por mucho.

Ejercicios

Determine si son falsos o verdaderos los siguientes enunciados:

- $n^{\frac{1}{\log n}} = \Theta(1)$
- $\sqrt{2}^{\log_2 n} = \Theta(\sqrt{n})$
- $\sqrt{n}\log_2 n = O(n)$
- $n^2 + n\sqrt{n} = O(n^2)$
- $n! = \Omega(2^n)$
- $3^n = \Omega(n2^n)$
- $\log 3^n = O(\log 2^n)$

Dadas las funciones *f (n)* y *g (n)* de cada inciso, compruebe si se cumple o no la relación $f(n) = \Theta(g(n))$:

- $f(n) = n, g(n) = \left(\frac{3}{2}\right)^n$
- $f(n) = n\log_2 n + \sqrt{n}, g(n) = (\log_2 n)^{\log_2 n}$
- $f(n) = \sqrt{\log_2 n}, g(n) = n^{2/3}$
- $f(n) = \sqrt{n}, g(n) = \log_2 n$
- $f(n) = n\sqrt{n}, g(n) = n^2 + 4n - 3$
- $f(n) = 10\log n + 1, g(n) = \log n^2 - 3$
- $f(n) = 2^n - n^2, g(n) = n^4 + n$

¿Encuentre dos funciones *f (n)* y *g (n)* que satisfagan las siguientes relaciones, si es que existen:

- $g(n) = o(f(n))$ y $g(n) \neq \Theta(f(n))$
- $g(n) = \Omega(f(n))$ y $g(n) \neq O(f(n))$
- $g(n) = \Theta(f(n))$ y $g(n) = o(f(n))$

Enliste de menor a menor el orden de crecimiento de las funciones de cada inciso:

- $n^2, n!, \left(\frac{3}{2}\right)^n, n2^n, n^{\frac{1}{\log n}}$
- $n^3, \log_2 n!, \ln\ln n, \ln n, n\ln n$
- $n^{\log_2\log_2 n}, 1, 2^{\log_2 n}, e^n, (\log_2 n)^{\log_2 n}$
- $7n^5 - n^3 + n, \log_2 n, \sqrt{n}, e^n$

Para las siguientes funciones g (n) encuentre funciones f (n) tales que g (n) = Θ (f (n)):

- $g(n) = \sum_{k=1}^{n} k + 1$
- $g(n) = \sum_{k=1}^{n} 2(4)^k + 3(2)^k$
- $g(n) = \sum_{k=1}^{n} \left(\frac{2}{3}\right)^k + 2k$
- $g(n) = \sum_{k=1}^{n} 7^{k-1} - 2^k + 5^k$

De cada par de funciones *g (n) y f (n)*, indique qué relación O, o, Ω, ω o Θ hay entre *g (n)* con *f (n)*:

- $g(n) = n^{100}, f(n) = 2^n$
- $g(n) = 5^n, f(n) = 50^n$
- $g(n) = \sqrt{n}, f(n) = \ln n$
- $g(n) = e^n, f(n) = n^5$
- $g(n) = n^{\log n}, f(n) = \log^n n$

RECURRENCIAS

Las recurrencias van de la mano con el paradigma de *divide y conquistarás*, porque proporcionan una manera natural de caracterizar los tiempos de ejecución de algoritmos que aplican este paradigma. Una recurrencia es una ecuación o desigualdad que describe una función en términos de su valor en entradas más pequeñas. Por ejemplo, como se observa en la sección 2.4, el tiempo de ejecución del algoritmo de ordenamiento *merge sort* se describe por la siguiente recurrencia:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & n > 1 \end{cases}$$

Las recurrencias pueden tomar muchas formas. Por ejemplo, un algoritmo puede dividir subproblemas en tamaños desiguales, tales como 2/3 a 1/3. Si la división y combinación de pasos toma tiempo lineal, tal algoritmo daría esta recurrencia:

$$T(n) = T\left(\frac{2}{3}n\right) + T\left(\frac{n}{3}\right) + \Theta(n)$$

Los subproblemas no son necesariamente restringidos al ser una fracción constante del tamaño original del problema. Por ejemplo, una versión recursiva de búsqueda lineal crearía solo un subproblema conteniendo solamente un elemento menos que el problema original. Cada llamada recursiva tomaría tiempo constante más el tiempo para el llamado recursivo, dando esta recurrencia:

$$T(n) = T(n-1) + \Theta(1)$$

A continuación, veremos tres métodos para resolver recurrencias, en donde se obtengan soluciones asintóticas *0* y/o Θ :

1. El método de sustitución, en donde se propone una cota y se emplea inducción matemática para comprobar la solución propuesta.

2. El método iterativo, en el cual —como su nombre lo indica— se itera la función unas cuantas veces para facilitar dar una propuesta de solución. Comúnmente se emplean técnicas de límites de sumas para resolver las recurrencias.

3. El método maestro da solución a recurrencias de la form $T(n) = aT(n/b) + f(n)$, donde $a \geq 1, b > 1$ y *f (n)* es una función dada.

Método de selección y Generalización

Es el método más simple y sencillo (Ramírez Benavides, 30):

- Se va evaluando la recurrencia para ciertos valores
- Se deduce, a partir del comportamiento mostrado, una ecuación que represente el comportamiento de la recurrencia.
- Se generaliza la solución, encontrando un patrón
- Se demuestra que la ecuación efectivamente resuelve a la recurrencia.

Considere la siguiente relación de recurrencia

$$a(n) = \begin{cases} 1 & n = 1 \\ 3a\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

- Construir una tabla con los valores que toma RR para diferentes soluciones de n
- TIP: Notar que la RR solo queda definida cuando n es potencia de dos

2^1	$3a\left(\frac{2}{2}\right) + 2^1 = 3a(1) + 2^1 = 3 \times 1 + 2^1$
2^2	$3a\left(\frac{4}{2}\right) + 2^2 = 3a(2) + 2^2 = 3(3 \times 1 + 2^1) + 2^2 = 3^2 \times 1 + 3^1 \times 2 + 2^2$
2^3	$3a\left(\frac{8}{2}\right) + 2^3 = 3a(4) + 2^3 = 3(3^2 \times 1 + 3^1 \times 2 + 2^2) + 2^3$ $= 3^3 \times 1 + 3^2 \times 2 + 3^1 \times 2^2 + 2^3$
2^4	$3a\left(\frac{16}{2}\right) + 2^4 = 3a(8) + 2^4 = 3(3^3 \times 1 + 3^2 \times 2 + 3^1 \times 2^2 + 2^3) + 2^4$ $= 3^4 \times 1 + 3^3 \times 2 + 3^2 \times 2^2 + 3^1 \times 2^3 + 2^4$

$$a(2^k) = 3^k 2^0 + 3^{k-1} 2^1 + \cdots + 3^1 2^{k-1} + 3^0 2^k = \sum_{i=0}^{k} 3^{k-i} 2^i$$

En la expresión anterior se puede notar que a queda en términos de una sumatoria, po lo que se requiera manipulación algorítmica para dejarla en términos exclusivamente del argumeno:

$$a(2^k) = \sum_{i=0}^{k} 3^{k-i}2^i = 3^k \sum_{i=0}^{k} \left(\frac{2}{3}\right)^i = 3^{k+1} - 2^{k+1}$$

Para resolver la fórmula se utiliza la serie geométrica donde r≠1, la suma de los primeros n términos de una serie geométrica es:

$$a + ar^1 + ar^2 + ar^3 + \cdots + ar^{n-1} = \sum_{k=0}^{n-1} ar^k = a\frac{1-r^n}{1-r}$$

Con a=1 y r=2/3:

$$a(2^k) = 3^k \sum_{i=0}^{k} \left(\frac{2}{3}\right)^i = 3^k \left(\frac{1-\left(2/3\right)^{k+1}}{1-2/3}\right) = 3^k \left(\frac{\frac{3^{k+1}-2^{k+1}}{3^{k+1}}}{1/3}\right)$$
$$= 3^{k+1}\left(\frac{3^{k+1}-2^{k+1}}{3^{k+1}}\right) = 3^{k+1} - 2^{k+1}$$

Con la solución se generaliza:

a(n)=3n+1-2n+1, n≥0

MÉTODO ITERATIVO

En el método iterativo la función que se quiere resolver se itera unas cuantas veces, de tal forma que facilite encontrar patrones de series, los cuales se pueden resolver empleando límites de series. La función se itera hasta algún valor específico de paro. Por ejemplo, considérese la siguiente recurrencia:

$$T(n) = 2T\left(\frac{n}{4}\right) + n$$

Se itera la ecuación unas cuantas veces hasta que sea visible algún patrón de las series que se presenten; de esta forma, tenemos:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{4}\right) + n \\ &= 2\left(2T\left(\frac{n}{4}\cdot\frac{1}{4}\right) + \frac{n}{4}\right) + n = 2^2\left(\frac{n}{4^2}\right) + \frac{n}{2} + n \\ &= 2^2\left(2T\left(\frac{n}{4^2}\cdot\frac{1}{4}\right) + \frac{n}{4^2}\right) + \frac{n}{2} + n = 2^3T\left(\frac{n}{4^3}\right) + \frac{n}{2^2} + \frac{n}{2} + n \\ \vdots &= \vdots \\ &= 2^kT\left(\frac{n}{4^k}\right) + n\sum_{i=0}^{k-1}\frac{1}{2^i} \end{aligned}$$

Nótese que el patrón de la recurrencia se puede expresar como en la última línea de la ecuación. Si se itera la ecuación hasta que $\frac{n}{4^k} = 1$, entonces tenemos $k = \log_4 n$ definimos a $T(1) = c;$ la suma $\sum_{i\geq 0}\frac{1}{2^i} = 2.$ Al sustituir estos valores en la ecuación tenemos:

$$T(n) = 2^{\log_4 n}T(1) + n(2)$$

Por propiedad de los logaritmos $2^{\log_4 n} = n^{\log_4 2}$ por lo tanto,

$$T(n) = cn^{1/2} + 2n$$

Empleando la notación asintótica, podemos establecer que $cn^{1/2} = \Theta(n^{1/2})$ y $2n = \Theta(n)$, entonces

$$T(n) = \Theta\left(n^{1/2}\right) + \Theta(n) = \Theta(n)$$

Veamos otro ejemplo. Considérese la siguiente recurrencia:

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

Al iterar la ecuación se tiene

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-1-1) + (n-1) + n = T(n-2) + 2n - 1 \\ &= T(n-2-1) + (n-2) + 2n - 1 = T(n-3) + 3n - 3 \\ \vdots &= \vdots \\ &= T(n-k) + kn - \sum_{i=1}^{k} i \end{aligned}$$

La recurrencia se comporta como en la última línea de la ecuación. Si se itera la ecuación hasta que $n = k$ y T (0)= Θ (1) definimos ; la suma se puede ver que representa la suma de la sucesión de números enteros positivos, por lo que esta se puede representar como un polinomio de segundo grado, por lo tanto, $\sum_{i\geq 1} i = \Theta(n^2)$. Al sustituir en la ecuación, se tiene:

$$\begin{aligned} T(n) &= \Theta(1) + n(n) - \Theta(n^2) \\ &= \Theta(1) + \Theta(n^2) - \Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$

MÉTODO MAESTRO

El método maestro es una "receta de cocina" para resolver, como se mencionó anteriormente, recurrencias de la forma

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

donde, a $\geq$ 1, b > 0 son constantes y *f (n)* es una función asintóticamente positiva. Una recurrencia de esta forma describe el tiempo de ejecución de un algoritmo que divide un problema de tamaño *n* en *a* subproblemas, cada uno de tamaño *n/b*. Los *a* subproblemas se resuelven recursivamente, cada uno en tiempo *T (n/b)*. La función *f (n)* comprende el costo de dividir el problema y combinar los resultados de los subproblemas. El método maestro depende del teorema maestro:

Teorema maestro: Sean las constantes a $\geq$ 1, b > 0 y , la función *f (n) y T (n)* definida en los enteros no negativos por la recurrencia

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

donde se interpreta a n/b, ya sea como [n/b] o [n/b]. Entonces, T (n) tiene las siguientes cotas asintóticas:

1. Si $f(n) = O(n^{\log_b a - \epsilon})$ para algún $\epsilon > 0$ constante, entonces $T(n) = \Theta(n^{\log_b a})$.
2. Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} \log n)$.
3. Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguna constante $\epsilon > 0$, y si $af(n/b) \leq cf(n)$ para alguna constante $0 < c < 1$ y todo n suficientemente grande, entonces $T(n) = \Theta(f(n))$.

Nótese que en los tres casos no se cubren todas las posibilidades para *f (n)*. Existe un vacío entre los casos 1 y 2 cuando *f (n)* es menor que $n^{\log_b a}$ pero no polinomialmente menor. Similarmente, existe un vacío entre los casos 2 y 3 cuando *f (n)* es mayor que $n^{\log_b a}$ pero no polinomialmente mayor.

Si la función *f (n)* 9 cae en uno de esos vacíos, o si la condición de regularidad en el caso 3 no se cumple, entonces no se puede usar el método maestro para resolver la recurrencia. Para emplear el método maestro, solo hay que determinar en qué caso cae la recurrencia.

Por ejemplo, considérese la recurrencia $T(n) = 9T(n/3) + n.$ Tenemos que a= 9, b = 3 y *f (n)* =*n*; de aquí que $\log_b a = \log_3 9 = 2.$ Dado *f (n) = 0 (n* $^{2-n}$*),* y proponiendo que $\in$ = 9, entonces se puede ver que se cumple que f (n) = 0 ($n^{1.1}$), por lo que se aplica el caso 1 y se concluye que la solución es T (n)= 0 (n^2)

Considérese la recurrencia $T(n) = T(2n/3) + 1,$ entonces se tiene $a = 1, b = 3/2$ y $f(n) = 1;$ de esta forma $\log_{3/2} 1 = 0,$ por lo tanto, n° =1. Se puede ver que se cumple que 1 = 0 (1), por lo que se aplica el caso 2 y finalmente la solución de la recurrencia es $T(n) = \Theta(\log n).$

Ahora véase la siguiente recurrencia: $T(n) = 2T(n/4) + n,$ se tiene que $a = 2, b = 4$ y $f(n) =$ *n*= esta forma $\log_4 2 = 1/2.$ Dado que $f(n) = \Omega(n^{1/2+\epsilon}),$ si tomamos $\epsilon = 1/5,$ aplica el caso 3 si se puede demostrar la condición de regularidad de *.f (n)* Para un suficientemente grande, tenemos que 2(n/4) $\leq$ *cn*; reduciendo términos, se concluye que la condición de regularidad se cumple para 1/2 < c $\leq$ 1. Por lo tanto, *T(n) = 0(n).*

Ejercicios

Resuelva los ítems empleando cualquier método.

- $T(n) = 2T\left(\frac{n}{4}\right) + 1$
- $T(n) = T\left(\frac{n}{2}\right) + 1$
- $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$
- $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$
- $T(n) = 2T\left(\frac{n}{4}\right) + n^2$
- $T(n) = 2T\left(\frac{n}{2}\right) + n^4$
- $T(n) = T\left(\frac{7}{10}n\right) + 10$
- $T(n) = 16T\left(\frac{n}{4}\right) + n^2$

- $T(n) = 7T\left(\frac{n}{3}\right) + n^2$
- $T(n) = 3T\left(\frac{n}{3} + 5\right) + \frac{n}{2}$
- $T(n) = 7T\left(\frac{n}{2}\right) + n^2$
- $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$
- $T(n) = T(n-2) + n^2$, donde $T(n) = c$ para $n \leq 2$.
- $T(n) = 4T\left(\frac{n}{3}\right) + n \log n$
- $T(n) = 3T\left(\frac{n}{3}\right) + \frac{n}{\log n}$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$
- $T(n) = 3T\left(\frac{n}{3} - 2\right) + \frac{n}{2}$
- $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$
- $T(n) = T(n-1) + \frac{1}{n}$, donde $T(n) = c$ para $n \leq 1$.
- $T(n) = T(n-1) + \log n$, donde $T(n) = c$ para $n \leq 1$.
- $T(n) = T(n-2) + \frac{1}{\log n}$, donde $T(n) = c$ para $n \leq 2$.
- $T(n) = \sqrt{n}T\left(\sqrt{n}\right) +$

REFERENCIAS

Abellanas, M. y Lodares, D. (1990). *Análisis de algoritmos y teoría de grafos.* México: Macrobit-Ra-Ma.

Alfonseca Cubero, E., Alfonseca Moreno, M. y Moriyon, R. (2007). *Teoría de autómatas y lenguajes formales.* México: MacGraw Hill.

Areán Álvarez, L. F. (2014). *¿Existen problemas irresolubles? Matemáticas, complejidad y computabilidad.* España: National Geographic.

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J. and Houston, L. A. (2007). *Object-Oriented Analysis and Design with Applications.* United States: Addison-Wesley.

Bueno de Arjona, G. (1987). *Introducción a la programación lineal y al análisis de sensibilidad.* México: Trillas.

Cairó, O. y Guardati, S. (2000). *Estructura de datos.* México: McGraw Hill.

Complejidad Algorítmica. (s. f.). Recuperado de http://www.slideshare.net/joemmanuel/complejidad-computacional.

Dasgupta, S., Papadimitriou, C. and Vazirani, U. (2008). *Algorithms.* New York: McGraw Hill.

Deutsch, D. (s. f.). *Lectures on Cuantum Computation*. Recuperado de http://www.quiprocone.org/Protected/DD_lectures.htm.

Dewdney, A. K. (1989). *The Turing Omnibus. 61 Excursions in Computer Science.* New York: Computer Science Press.

Docentes Educación Navarra (29 de diciembre de 2018). *Unidad 4. Programación lineal*. Recuperado de http://docentes.educacion.navarra.es/mpastorg/cd_alumno/modeloG/2bach_CSS/Datos/Unidades/04/CCss_t04_1_mec.pdf.

Fregoso, A. (1997). *Comunicación y lenguaje.* México: Universidad Autónoma Chapingo.

Garey, M. R. and Johnson, D. S. (1975). *Computer and intratability. A guide to the theory of NP Completeness.* U. S. A.: A series of Books in the Mathematical Science.

Goldshlager, L. y Lister, A. (1986). *Introducción moderna a las ciencias de la computación con un enfoque algorítmico.* México: Prentice Hall.

Gómez Fuentes, M. y Cervantes Ojeda, J. (2014). *Introducción al análisis y al diseño de algoritmos.* México: Universidad Autónoma Metropolitana.

Horowitz, E. and Sahni, S. (1978). *Fundamentals of Computer Algorithms.* United States of America: Computer Science Press.

Kewis, H. R. and Papadimitriou, C. H. (1989). *Elements of The Theory of Computation.* U. S. A.: Prentice Hall.

Langholz, G., Francioni, J. and Kandel, A. (1989). *Elements of computer organization.* New York: Prentice Hall.

Levin, G. (2004). *Computación y programación moderna, perspectiva integral de la informática.* México: Addison Wesley.

Loomis, M. E. (2013). *Estructura de datos y organización de archivos.* México: Prentice Hall.

Método de la Burbuja (15 de 10 de 2016). Recuperado de http://c.conclase.net/orden/?cap=burbuja.

Penrose, R. (1989). *La mente nueva del emperador. En torno a la cibernética, la mente y las leyes de la Física.* México: Fondo de Cultura Económica.

Piñeiro, G. E. (2017). *Dos teoremas que revolucionaron las matemáticas. Gödel.* España: RBA, (Serie: Genios Matemáticos).

Serie de Fibonacci (s. f.). Recuperado de http://www.ugr.es/~eaznar/fibo.htm.

Singh, S. (1995). *El enigma de Fermat.* México: Planeta.

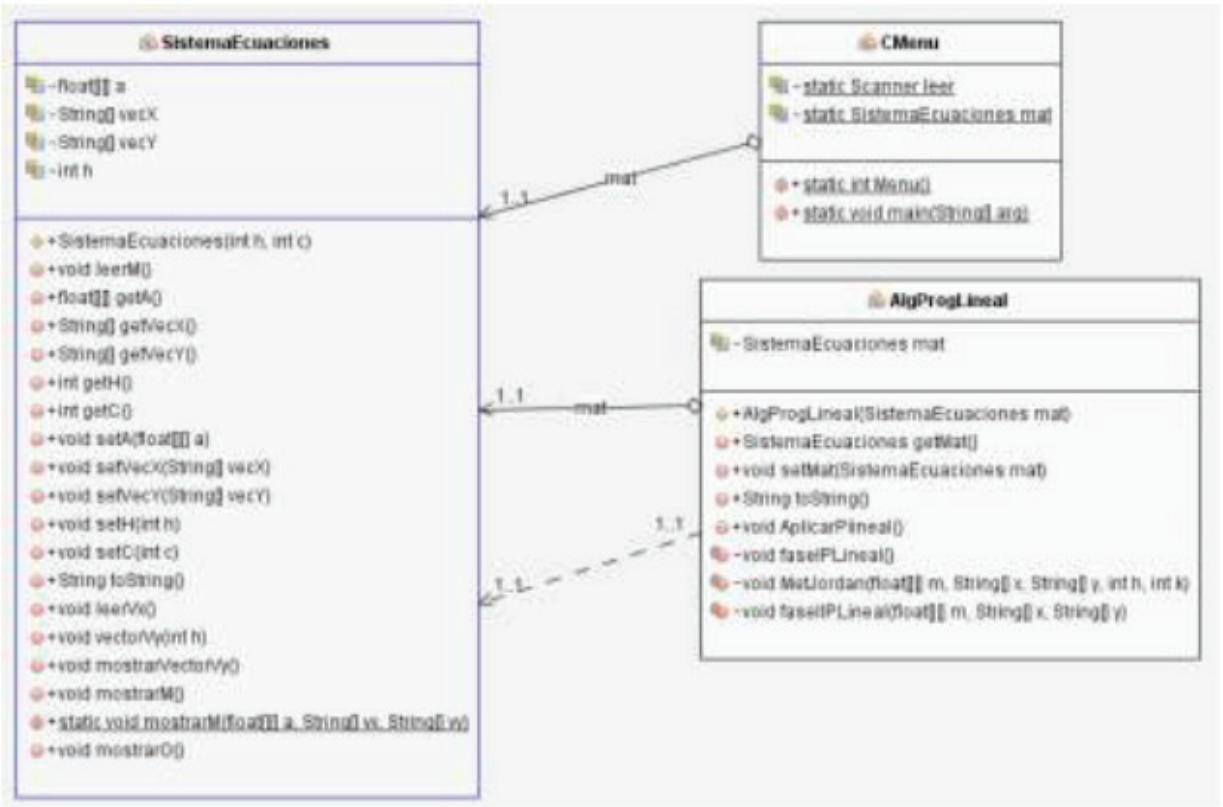
ANEXO A

Programación lineal

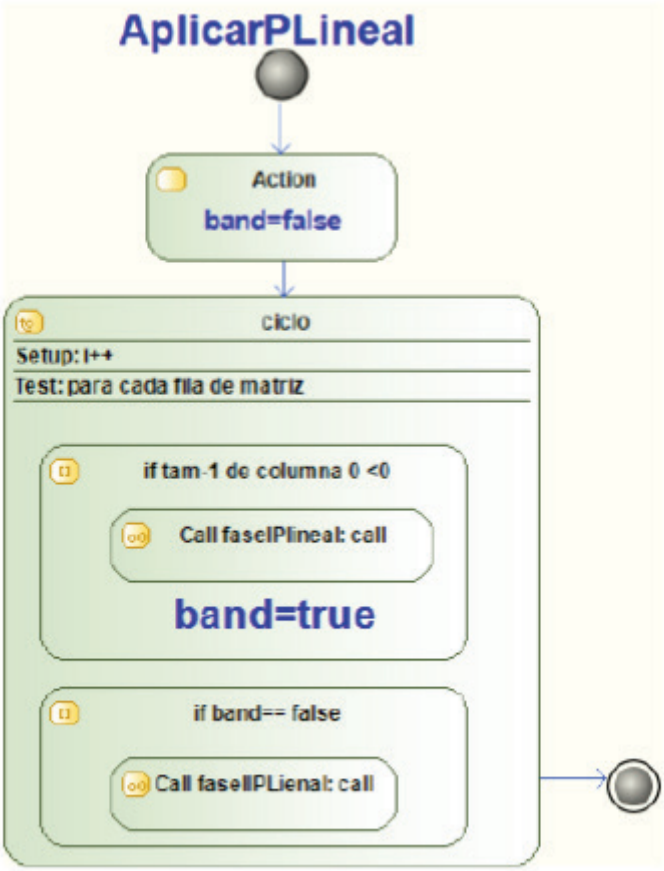
INTRODUCCIÓN

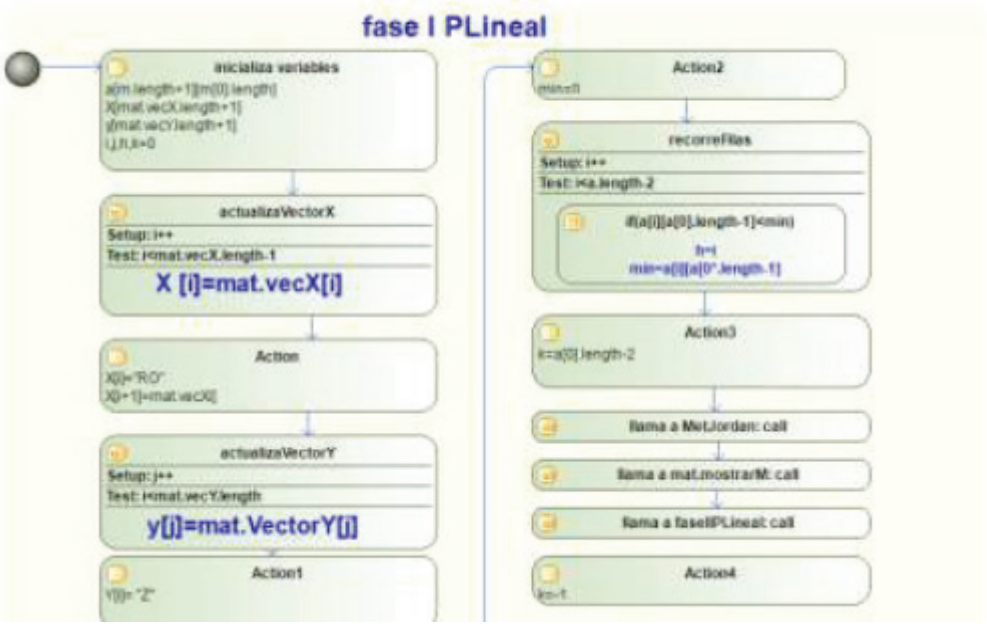
En este anexo se presenta el código de programación lineal por el algoritmo de Tucker. En la primera sección se muestra el diagrama de clases, en la segunda el diagrama de actividades y al final el código completo.

Diagrama de clases

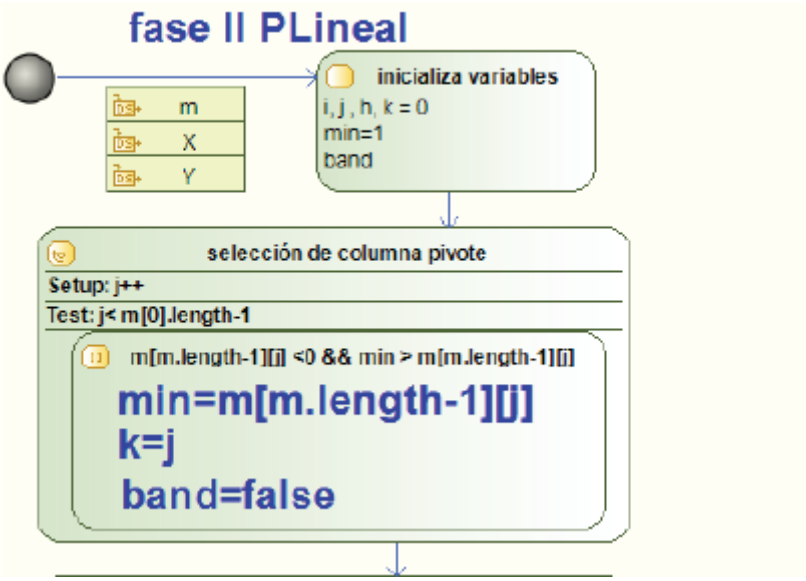


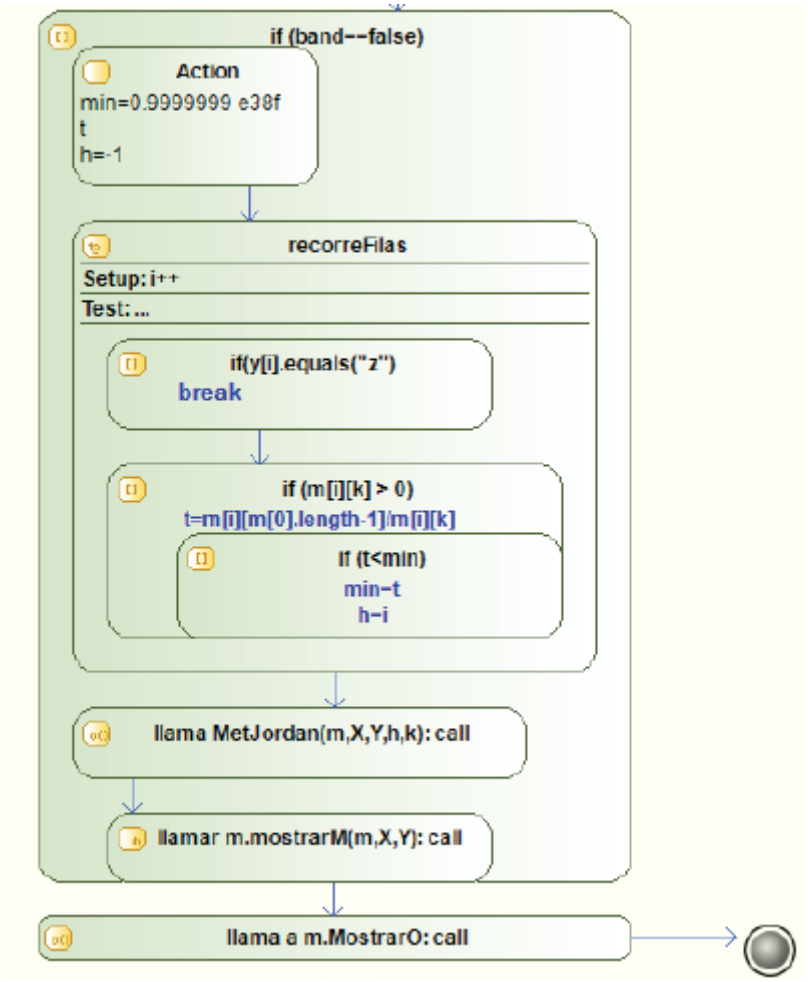
Diagramas de actividades

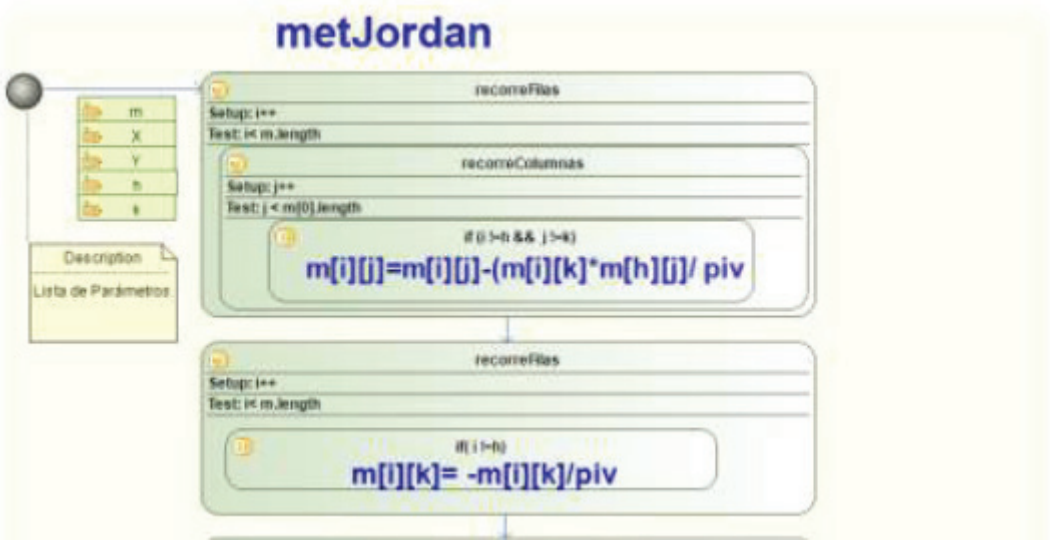


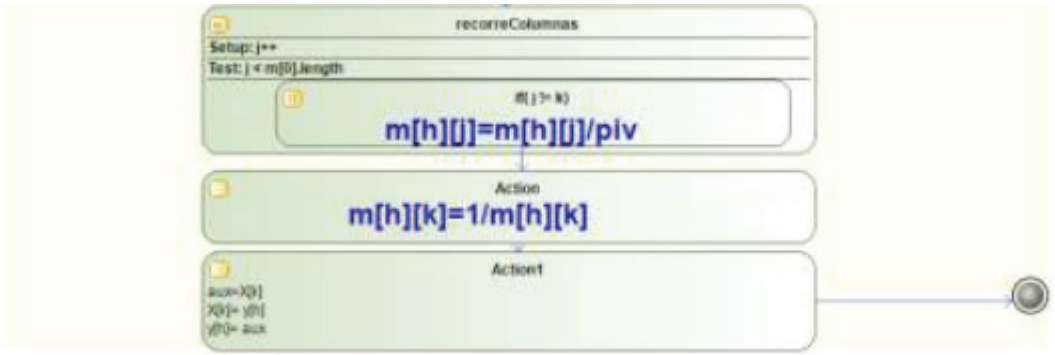



recorridoFilas
Setup: i++
Test: i<mat.a.length
recorridoColumnas
Setup: j++
Test: j<mat.a[0].length-1
a[i][j]= mat.a[i][j]
recorreFila
Setup: i++
Test: i<mat.a.length
a[i][a[0].length-1]=mat.a[i][mat.a[0].length-1]
recorreFilas
Setup: i++
Test: i<mat.a.length
if (mat.a[i][mat.a[0].length -1]<0)
a[i][a[0].length-2]=-1
else a[i][a[0].length-2]=0
recorreColumnas
Setup: j++
Test: j<a[0].length
if (j != a[0].length-2
a[a.length-1][j]=0
else a[a.length-1][j]=1
llama a mat.mostrarM(a,X,Y): call
recorrerX
Setup: i++
Test: i<x.length
if(x[i].compareTo(r)==0)
k=i
break
if (k != -1)
h=0
recorreCol
Setup: j++
Test: j<X.length
if (j !=k)
mat.vectX[h]=X[j]
recorreFila
Setup: i++
Test: i< mat.a.length
a[i][h]=a[i][j]
h++
recorreFila
Setup: i++
Test: i < mat.vecY.length
mat.vecY[i]=y[i]
llama mat.MostrarM(a, X, Y): call
llama faseIIPLineal(a, X, Y): call
else mostrar "Sin Solución"


fase II PLineal
m
X
Y
inicializa variables
i, j , h, k = 0
min=1
band
selección de columna pivote
Setup: j++
Test: j< m[0].length-1
m[m.length-1][j] <0 && min > m[m.length-1][j]
min=m[m.length-1][j]
k=j
band=false


if (band==false)
Action
min=0.9999999 e38f
t
h=-1
recorreFilas
Setup: i++
Test: ...
if(y[i].equals("z")
break
if (m[i][k] > 0)
t=m[i][m[0].length-1]/m[i][k]
If (t<min)
min=t
h=i
llama MetJordan(m,X,Y,h,k): call
llamar m.mostrarM(m,X,Y): call
llama a m.MostrarO: call


metJordan
m
X
Y
h
k
Description
Lista de Parámetros
recorreFilas
Setup: i++
Test: i< m.length
recorreColumnas
Setup: j++
Test: j < m[0].length
m[i][j]=m[i][j]-(m[i][k]*m[h][j]/ piv
recorreFilas
Setup: i++
Test: i< m.length
m[i][k]= -m[i][k]/piv



Código en Java

Código de la clase CMenu

```
package progentera;
import java.util.Scanner;
public class CMenu{
 static Scanner leer;
 static SistemaEcuaciones mat;
public static int Menu(){
 leer=new Scanner(System.in);
              int i;
              do{
                     System.out.println("\tMenu");
                     System.out.println("1. Programación Lineal");
                     System.out.println("3. Salir");
                     i=leer.nextInt();
              }while(i<1 && i>3);
              return i;
       }

public static void main(String[] arg){
              int x,h,c;
              do{ x=Menu();
              if (x<3){ System.out.print("Da el número de inecuaciones:");
                     h=leer.nextInt()+1;
                     System.out.print("Da el número de incognitas:");
                     c=leer.nextInt()+1;
                     mat= new SistemaEcuaciones(h,c);
 mat.leerVx();
 mat.vectorVy(h);
 mat.leerM();
                     mat.mostrarM();
                     switch(x){
                     case 1: AlgProgLineal calculos = new AlgProgLineal(mat);
 calculos.AplicarPlineal();
 System.out.println(calculos.mat.toString());
                                   break;
                                   }
                     }
              }while(x<3);
       }
}
```

Código Aplicar Programación Lineal

```
package progentera;
public class AlgProgLineal{
SistemaEcuaciones mat;
public AlgProgLineal(SistemaEcuaciones mat) { this.mat = mat; }
public SistemaEcuaciones getMat() { return mat; }
public void setMat(SistemaEcuaciones mat) {this.mat = mat; }
 @Override
public String toString() { return "AlgProgLineal{" + "mat=" + mat + '}'; }

public void AplicarPlineal(){
       boolean band=false;
       for (int i=0;i<mat.a.length-1;i++)
              if (mat.a[i][mat.a[0].length-1]<0){
                           faseIPLineal();
                           band=true;
                           break;}
              if (band==false) faseIIPLineal(mat.a,mat.vecX,mat.vecY);
}

private void faseIPLineal(){
              System.out.println("FASE I");
 String [] x=new String [mat.vecX.length+1];
 String [] y=new String [mat.vecY.length+1];
              int i,j,h=0,k=0;
              float [][] a=new float[mat.a.length+1][mat.a[0].length+1];
              //actualizando los vectores
              for (i=0;i<mat.vecX.length-1;i++) x[i]=mat.vecX[i];
              x[i]="Ro";
              x[i+1]=mat.vecX[i];
              for (i=0;i<mat.vecY.length;i++) y[i]=mat.vecY[i];
              y[i]="Z'";
       //introduciendo los valores a la matriz a
              for (i=0;i<mat.a.length;i++)
                     for (j=0;j<mat.a[0].length-1;j++)
                           a[i][j]=mat.a[i][j];
              for (i=0;i<mat.a.length;i++)
                     a[i][a[0].length-1]=mat.a[i][mat.a[0].length-1];

              for (i=0;i<mat.a.length;i++)
                     if(mat.a[i][mat.a[0].length-1]<0)
                           a[i][a[0].length-2]=-1;
                     else
                           a[i][a[0].length-2]=0;

              for (j=0;j<a[0].length;j++)
                     if(j!=a[0].length-2)
                            a[a.length-1][j]=0;
                     else
                            a[a.length-1][j]=1;
              mat.mostrarM(a,x,y);
              float min=0;
              for (i=0;i<a.length-2;i++)
              if (a[i][a[0].length-1]<min){ h=i;min=a[i][a[0].length-1]; }
              k=a[0].length-2;
              MetJordan(a,x,y,h,k);
              mat.mostrarM();
              faseIIPLineal(a,x,y);
              String r="Ro";
       //Elimina a Ro del tableau de Tucker
              k=-1;
              for (i=0;i<x.length-1;i++)
                     if(x[i].compareTo(r)==0){    k=i; break; }
              if (k!=-1){h=0;
                     for (j=0;j<x.length;j++)
                            if (j!=k){ mat.vecX[h]=x[j];
                            for (i=0;i<mat.a.length;i++) a[i][h]=a[i][j];
                            h++; }
                     for (i=0;i<mat.vecY.length;i++)     mat.vecY[i]=y[i];
                     System.out.println("La matriz antes de fase II");
                     mat.mostrarM(a,x,y);
                     System.out.println();
                     faseIIPLineal(a,x,y);
              }
              else System.out.println("Sin solución");
```

```
	}

private void MetJordan(float[][] m, String [] x, String [] y,int h, int k){
	float piv; int i,j;
 System.out.println("Pivote=("+h+","+k+")");
	piv=m[h][k];
	for (i=0;i<m.length;i++)
	 for (j=0;j<m[0].length;j++)
		if(i!=h && j!=k) m[i][j]=m[i][j]-(m[i][k]*m[h][j])/piv;
	for (i=0;i<m.length;i++)
	 if (i!=h) m[i][k]=-m[i][k]/piv;
	for (j=0;j<m[0].length;j++)
 if (j!=k) m[h][j]=m[h][j]/piv;
		m[h][k]=1/m[h][k];
		String aux;
		aux=x[k];
		x[k]=y[h];
 y[h]=aux;
	}


private void faseIIPLineal(float [][] m, String [] x, String [] y){
		int i,j,h=0,k=0;
		float min=1;
		boolean band;
		System.out.println("Fase II");
		do{ band=true;
		//se escoge la columna del pivote
		for (j=0;j<m[0].length-1;j++)
		 if (m[m.length-1][j]<0 && min>m[m.length-1][j]){
			min=m[m.length-1][j];
			System.out.println("min="+min+"\t");
			k=j;
			band=false;
}

if(band==false){
 //se escoge la hilera pivote
 min=0.99999e38f;
	 float t;
				h=-1;
				for (i=0;i<m.length-1;i++){
					if (y[i].equals("z")) break;
					else if (m[i][k]>0){
							t=m[i][m[0].length-1]/m[i][k];
							if (t<min){ min=t; h=i;	}
					}
				}
	//Se realiza un intercambio de Jordan
				MetJordan(m,x,y,h,k);
				mat.mostrarM(m,x,y);
			}
		}while(band==false);
		mat.mostrarO();
	}
}
```

Código Clase SistemaEcuaciones

```
package progentera;
import java.util.Scanner;
public class SistemaEcuaciones {
 float [][] a; int h,c;
 String [] vecX; String [] vecY;

public SistemaEcuaciones(int h, int c){    //crea la matriz
       a=new float [h][c];
 vecX=new String[c];
 vecY=new String[h];
       }
public void leerM(){  //lee la matriz
       Scanner leer= new Scanner(System.in);
 for (int i=0;i<a.length;i++)
       for (int j=0;j<a[0].length;j++){
              System.out.print("a["+i+"]["+j+"]=");
              a[i][j]=leer.nextInt();
       }
       }

 public float[][] getA() { return a; }
 public String[] getVecX() { return vecX; }
 public String[] getVecY() { return vecY; }
 public int getH() { return h; }
 public int getC() { return c; }
 public void setA(float[][] a) {this.a = a; }
 public void setVecX(String[] vecX) { this.vecX = vecX; }
 public void setVecY(String[] vecY) { this.vecY = vecY; }

 public void setH(int h) {this.h = h;}
 public void setC(int c) {this.c = c;}

 @Override
 public String toString() { int i;
 StringBuffer matriz= new StringBuffer();
 for (i=0;i<a[0].length;i++)
              matriz.append(String.valueOf(a[i])+"\t");
 return "SistemaEcuaciones{" + "a=" + matriz + ", vecX=" + vecX + ", vecY=" + vecY + ", h=" +
h + ", c=" + c + '}'; }

public void leerVx(){        // da valores a variables básicas
       String c="1"; int i;
 Scanner leer= new Scanner(System.in);
       for (i=0;i<vecX.length-1;i++){
              System.out.print("Da el nombre de la variable "+i+"=>");
              vecX[i]=leer.next();  }
       vecX[i]="";
       vecX[i]=vecX[i].concat(c);
}
public void vectorVy(int h){        //da valores a variables no básicas.
       char a; String c;     int i;
       for (i=0;i<vecY.length-1;i++) vecY[i]="y";
       vecY[vecY.length-1]="";
       if (vecY.length<=10){
              for (i=0;i<vecY.length-1;i++){
                     a=(char)(i+48);
                     c=""+a;
                     vecY[i]=vecY[i].concat(c);
              }
```

```
                c=""+'z';
                vecY[i]=vecY[i].concat(c);
        }
        else{
                for (i=0;i<10;i++){
                        a=(char)(i+48);
                        c=""+a;
                        vecY[i]=vecY[i].concat(c);
                }
                int j=vecY.length/10;
                for (i=1;i<j;i++)
                        for (int k=0;k<10;k++){
                                a=(char)(i+48);
                                c=""+a;
                                vecY[i*10+k]=vecY[i*10+k].concat(c);
                                a=(char)(k+48);
                                c=""+a;
                                vecY[i*10+k]=vecY[i*10+k].concat(c);
                        }
                int s=vecY.length%10;
                for (i=0;i<s-1;i++){
                        a=(char)(j+48);
                        c=""+a;
                        vecY[j*10+i]=vecY[j*10+i].concat(c);
                        a=(char)(i+48);
                        c=""+a;
                        vecY[j*10+i]=vecY[j*10+i].concat(c);
                }
                c=""+'Z';
                vecY[j*10+s-1]=vecY[j*10+s-1].concat(c);
        }
}
public void mostrarVectorVy(){
        for (int i=0;i<vecY.length;i++)
                        System.out.println(vecY[i]+"\t");
}

public void mostrarM(){
        int i,j,num,entero=0;
        float x,decimal;
        System.out.println();
        System.out.print("\t");
        for (i=0;i<a[0].length;i++)
                System.out.print(vecX[i]+"\t");
        System.out.println();
        for (i=0;i<a.length;i++){
                System.out.print(vecY[i]+"\t");
                for (j=0;j<a[0].length;j++){
                        num=(int)(a[i][j]*100);
                        entero=num/100;
                        x=entero;
                        entero=num%100;
                        decimal=entero/100f;
                        x+=decimal;
                        System.out.print(x+"\t");
                }
                System.out.println();
        }
}
public static void mostrarM(float[][] a, String[] vx, String [] vy){
        int i,j,num,entero=0;
```

```
        float x,decimal;
        System.out.println();
        System.out.print("\t");
        for (i=0;i<a[0].length;i++)
                System.out.print(vx[i]+"\t");
        System.out.println();
        for (i=0;i<a.length;i++){
                System.out.print(vy[i]+"\t");
                for (j=0;j<a[0].length;j++){
                        num=(int)(a[i][j]*100);
                        entero=num/100;
                        x=entero;
                        entero=num%100;
                        decimal=entero/100f;
                        x+=decimal;
                        System.out.print(x+"\t");
                }
                System.out.println();
        }
}

public void mostrarO(){
        System.out.println("Optimo");
        for (int i=0;i<vecY.length;i++)
                System.out.println(vecY[i]+"="+a[i][a[0].length-1]);

        }

}
```

INTRODUCCIÓN AL ANÁLISIS DE ALGORITMOS

Se terminó de imprimir en septiembre de 2019 en los
talleres de Editorial Centro de Estudios e
Investigaciones para el Desarrollo Docente. CENID AC
Av. México #2798. Piso 5-B, Torre Diamante
Circunvalación Vallarta
C.P. 44680 Guadalajara, Jalisco, México
teléfono: 01 (33) 1061 8187

Tiraje: 1000