AREA INFORMÁTICA

INACAP
www.inacap.cl

MANUAL DE ANÁLISIS Y DISEÑO DE ALGORITMOS

Versión 1.0

Dirección de Área Informática
www.informatica.inacap.cl

AREA INFORMÁTICA

INACAP
www.inacap.cl

Colaboraron en el presente manual:

Víctor Valenzuela Ruz
v_valenzuela@inacap.cl
Docente
Ingeniería en Gestión Informática
INACAP Copiapó

Copyright

Prefacio

"Lo que debemos aprender a hacer lo aprendemos haciéndolo".

Aristóteles, Ethica Nicomachea II (325 A.C.)

El presente documento ha sido elaborado originalmente como apoyo a la asignatura de "*Análisis y Diseño de Algoritmos*" del séptimo semestre de la carrera de Ingeniería en Gestión Informática, del Instituto Nacional de Capacitación (INACAP). Este documento engloba la mayor parte de la materia de este curso troncal e incluye ejemplos resueltos y algunos ejercicios que serán desarrollados en clases.

El manual ha sido concebido para ser leído en forma secuencial, pero también para ser de fácil consulta para verificar algún tema específico.

No se pretende que estos apuntes sustituyan a la bibliografía de la asignatura ni a las clases teóricas, sino que sirvan más bien como complemento a las notas que el alumno debe tomar en clases. Asimismo, no debe considerarse un documento definitivo y exento de errores, si bien ha sido elaborado con detenimiento y revisado exhaustivamente.

El autor pretende que sea mejorado, actualizado y ampliado con cierta frecuencia, lo que probablemente desembocará en sucesivas versiones, y para ello nadie mejor que los propios lectores para plantear dudas, buscar errores y sugerir mejoras.

El Autor.

Copiapó, Enero 2003

Índice General

Presentación

El curso de *Análisis y Diseño de Algoritmos* (ADA) tiene como propósito fundamental proporcionar al estudiante las estructuras y técnicas de manejo de datos más usuales y los criterios que le permitan decidir, ante un problema determinado, cuál es la estructura y los algoritmos óptimos para manipular los datos.

El curso está diseñado para proporcionar al alumno la madurez y los conocimientos necesarios para enfrentar, tanto una gran variedad de los problemas que se le presentarán en su vida profesional futura, como aquellos que se le presentarán en los cursos más avanzados.

El temario gira en torno a dos temas principales: estructuras de datos y análisis de algoritmos. Haciendo énfasis en la abstracción, se presentan las estructuras de datos más usuales (tanto en el sentido de útiles como en el de comunes), sus definiciones, sus especificaciones como tipos de datos abstractos (TDA's), su implantación, análisis de su complejidad en tiempo y espacio y finalmente algunas de sus aplicaciones. Se presentan también algunos algoritmos de ordenación, de búsqueda, de recorridos en gráficas y para resolver problemas mediante recursión y retroceso mínimo analizando también su complejidad, lo que constituye una primera experiencia del alumno con el análisis de algoritmos y le proporcionará herramientas y madurez que le serán útiles el resto de su carrera.

Hay que enfatizar que el curso no es un curso de programación avanzada, su objetivo es preparar al estudiante brindándole una visión amplia de las herramientas y métodos más usuales para la solución de problemas y el análisis de la eficiencia de dichas soluciones. Al terminar el curso el alumno poseerá un nutrido "arsenal" de conocimientos de los que puede echar mano cuando lo requiera en su futura vida académica y profesional. Sin embargo, dadas estas características del curso, marca generalmente la frontera entre un programador principiante y uno maduro, capaz de analizar, entender y programar sistemas de *software* más o menos complejos.

Para realizar las implantaciones de las distintas estructuras de datos y de los algoritmos que se presentan en el curso se hará uso del lenguaje de programación MODULA-2, C++ y Java.

Capítulo 1

Introducción

1.1 Motivación y Objetivos

La representación de información es fundamental para las Ciencias de la Computación.

La Ciencia de la Computación (*Computer Science*), es mucho más que el estudio de cómo usar o programar las computadoras. Se ocupa de algoritmos, métodos de calcular resultados y máquinas autómatas.

Antes de las computadoras, existía la **computación**, que se refiere al uso de métodos sistemáticos para encontrar soluciones a problemas algebraicos o simbólicos.

Los babilonios, egipcios y griegos, desarrollaron una gran variedad de métodos para calcular cosas, por ejemplo el área de un círculo o cómo calcular el máximo común divisor de dos números enteros (teorema de Euclides).

En el siglo XIX, Charles Babbage describió una máquina que podía liberar a los hombres del tedio de los cálculos y al mismo tiempo realizar cálculos confiables.

La motivación principal de la computación por muchos años fue la de desarrollar cómputo numérico más preciso. La Ciencia de la Computación creció del interés en sistemas formales para razonar y la mecanización de la lógica, así cómo también del procesamiento de datos de negocios. Sin embargo, el verdadero impacto de la computación vino de la habilidad de las computadoras de representar, almacenar y transformar la información.

La computación ha creado muchas nuevas áreas como las de correo electrónico, publicación electrónica y multimedia.

La solución de problemas del mundo real, ha requerido estudiar más de cerca cómo se realiza la computación. Este estudio ha ampliado la gama de problemas que pueden ser resueltos.

Por otro lado, la construcción de algoritmos es una habilidad elegante de un gran significado práctico. Computadoras más poderosas **no** disminuyen el significado de algoritmos veloces. En la mayoría de las aplicaciones no es el hardware el cuello de botella sino más bien el *software* inefectivo.

Este curso trata de tres preguntas centrales que aparecen cuando uno quiere que un computador haga algo: ¿Es posible hacerlo? ¿Cómo se hace? y ¿Cuán rápido puede hacerse? El curso da conocimientos y métodos para responder estas preguntas, al mismo tiempo intenta aumentar la capacidad de encontrar algoritmos efectivos. Los problemas para resolver, son un entrenamiento en la solución algorítmica de problemas, y para estimular el aprendizaje de la materia.

Objetivos Generales:

1. Introducir al alumno en el análisis de complejidad de los algoritmos, así como en el diseño e implementación de éstos con las técnicas y métodos más usados.

2. Desarrollar habilidades en el uso de las técnicas de análisis y diseño de algoritmos computacionales.

3. Analizar la eficiencia de diversos algoritmos para resolver una variedad de problemas, principalmente no numéricos.

4. Enseñar al alumno a diseñar y analizar nuevos algoritmos.

5. Reconocer y clasificar los problemas de complejidad polinómica y no polinómica.

Metas del curso:

Al finalizar este curso, el estudiante debe estar capacitado para:

- analizar, diseñar e implementar algoritmos iterativos y recursivos correctos.

- medir la eficiencia de algoritmos iterativos y recursivos, y de tipos o clases de datos orientados por objetos.

- utilizar la técnica de diseño de algoritmos más adecuada para una aplicación en particular.

- definir la aplicabilidad de las diferentes técnicas de búsqueda y ordenamiento, del procesamiento de cadenas de caracteres, de los métodos geométricos, del uso de los algoritmos de grafos, de los algoritmos paralelos y de los algoritmos de complejidad no polinómica.

- diferenciar entre los algoritmos de complejidad polinómica y no polinómica.

1.2 Algunas Notas sobre la Historia de los Algoritmos

El término proviene del matemático árabe *Al'Khwarizmi*, que escribió un tratado sobre los números. Este texto se perdió, pero su versión latina, *Algoritmi de Numero Indorum*, sí se conoce.

El trabajo de *Al'Khwarizmi* permitió preservar y difundir el conocimiento de los griegos (con la notable excepción del trabajo de Diofanto) e indios, pilares de nuestra civilización. Rescató de los griegos la rigurosidad y de los indios la simplicidad (en vez de una larga demostración, usar un diagrama junto a la palabra *Mira*). Sus libros son intuitivos y prácticos y su principal contribución fue simplificar las matemáticas a un nivel entendible por no expertos. En particular muestran las ventajas de usar el sistema decimal indio, un atrevimiento para su época, dado lo tradicional de la cultura árabe.

La exposición clara de cómo calcular de una manera sistemática a través de algoritmos diseñados para ser usados con algún tipo de dispositivo mecánico similar a un ábaco, más que con lápiz y papel, muestra la intuición y el poder de abstracción de *Al'Khwarizmi*. Hasta se preocupaba de reducir el número de operaciones necesarias en cada cálculo. Por esta razón, aunque no haya sido él el inventor del primer algoritmo, merece que este concepto esté asociado a su nombre.

Los babilonios que habitaron en la antigua Mesopotania, empleaban unas pequeñas bolas hechas de semillas o pequeñas piedras, a manera de "cuentas" y que eran agrupadas en carriles de caña. Más aún, en 1.800 A.C. un matemático babilónico inventó los algoritmos que le permitieron resolver problemas de cálculo numérico.

En 1850 A.C., un algoritmo de multiplicación similar al de expansión binaria es usado por los egipcios.

La teoría de las ciencias de la computación trata cualquier objeto computacional para el cual se puede crear un buen modelo. La investigación en modelos formales de computación se inició en los 30's y 40's por Turing, Post, Kleene, Church y otros. En los 50's y 60's los lenguajes de programación, compiladores y sistemas operativos estaban en desarrollo, por lo tanto, se convirtieron tanto en el sujeto como la base para la mayoría del trabajo teórico.

El poder de las computadoras en este período estaba limitado por procesadores lentos y por pequeñas cantidades de memoria. Así, se desarrollaron teorías (modelos, algoritmos y análisis) para hacer un uso eficiente de ellas. Esto dio origen al desarrollo del área que ahora se conoce como "Algoritmos y Estructuras de Datos". Al mismo tiempo se hicieron estudios para comprender la complejidad inherente en la solución de algunos problemas. Esto dió origen a lo que se conoce como la jerarquía de problemas computacionales y al área de "Complejidad Computacional".

1.3 Fundamentos Matemáticos

Monotonicidad

Una función *f* es *monótona* si es creciente o decreciente. Es ***creciente*** si rige la implicación siguiente:

$$\forall\, n_0, n_1 : (n_1 = n_2) \rightarrow f(n_1) = f(n_2)$$

Es ***decreciente*** si rige la implicación siguiente:

$$\forall\, n_0, n_1 : (n_1 = n_2) \rightarrow f(n_1) = f(n_2)$$

Una función *f(n)* es **monotónicamente creciente** si $m \leq n$ implica que $f(m) \leq f(n)$. Similarmente, es **monotónicamente decreciente** si $m \leq n$ implica que $f(m) \geq f(n)$. Una función *f(n)* es **estrictamente creciente** si $m < n$ implica que $f(m) < f(n)$ y **estrictamente decreciente** si $m < n$ implica que $f(m) > f(n)$.

Conjuntos

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

Un **orden lineal** tiene las siguientes propiedades:

- Para cualesquier elemento *a* y *b* en el conjunto *S*, exactamente uno de $a < b$, $a = b$, o $a > b$ es verdadero.
- Para todos los elementos *a*, *b* y *c* en el conjunto *S*, si $a < b$, y $b < c$, entonces $a < c$. Esto es conocido como la propiedad de **transitividad**.

Permutación

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

Funciones Piso y Techo

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

Para cualquier número real x, denotamos al mayor entero, menor que o igual a x como $\boldsymbol{floor}(x)$, y al menor entero, mayor que o igual a x como $\boldsymbol{ceiling}(x)$; (floor=piso, ceiling=techo). Para toda x real,

$$x - 1 < \boldsymbol{floor}(x) \leq x \leq \boldsymbol{ceiling}(x) < x + 1$$

Para cualquier entero n,

$$\boldsymbol{ceiling}(x/2) + \boldsymbol{floor}(x/2) = n,$$

y para cualquier entero n y enteros $a \neq 0$ y $b \neq 0$,

$$\boldsymbol{ceiling}(\boldsymbol{ceiling}(n/a)/b) = \boldsymbol{ceiling}(n/ab) \quad \text{y}$$
$$\boldsymbol{floor}(\boldsymbol{floor}(n/a)/b) = \boldsymbol{floor}(n/ab).$$

Las funciones ***floor*** y ***ceiling*** son monótonas crecientes.

Operador módulo

Esta función regresa el residuo de una división entera. Generalmente se escribe $n \bmod m$, y el resultado es el entero r, tal que $n = qm + r$ para q un entero y/0 menor o igual que r y $r < m$.

Por ejemplo: $5 \bmod 3 = 2$ y $25 \bmod 3 = 1$

Polinomios

Dado un entero positivo d, un **polinomio en *n* de grado *d*** es una función $p(n)$ de la forma:

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

donde las constantes $a_0, a_1, \ldots, a_d$ son los **coeficientes** del polinimio y $ad \neq 0$. Un polinomio es **asintóticamente positivo** si y sólo si $a_d > 0$. Para un polinomio asintóticamente positivo $p(n)$ de grado d, tenemos que $p(n) = \boldsymbol{O}(n^d)$. Para cualquier constante real $a \geq 0$, la función n^a es monótona creciente, y para cualquier constante real $a \leq 0$, la función n^a es monótona decreciente. Decimos que una función $f(n)$ es **polinomialmente acotada** si $f(n) = n^{\boldsymbol{O}(1)}$, que es equivalente en decir que $f(n) = \boldsymbol{O}(n^k)$ para alguna constante k.

Exponenciales

Para toda $a \neq 0$, m y n, tenemos las siguientes identidades:

$a^o = 1$
$a^1 = a$
$a^{(-1)} = 1/a$
$(a^m)^n = a^{m\,n}$
$(a^m)^n = (a^n)^m$
$a^m a^n = a^{m+n}$

Para todo n y $a \geq ,1$

$$\lim_{n \to \infty} n^b / a^n = 0$$

de lo que concluimos que $n^b = o(a^n)$.

Entonces, cualquier función exponencial positiva crece más rápido que cualquier polinomio.

$$\exp(x) = \sum_{i=0}^{\infty} x^i / (i!)$$

Logaritmos

Un **logaritmo** de base b para el valor y, es la potencia al cual debe elevarse b para obtener y. $Log_b\, y = x \Leftrightarrow b^x = y$.

Usos para los logaritmos:

- ¿Cuál es el número mínimo de bits necesarios para codificar una colección de objetos? La respuesta es $techo(log_2\, n)$.

- Por ejemplo, si se necesitan 1000 códigos que almacenar, se requerirán al menos $techo(log2\ 1000) = 10$ bits para tener 1000 códigos distintos. De hecho con 10 bits hay 1024 códigos distintos disponibles.

- Análisis de algoritmos que trabajan rompiendo un problema en subproblemas más pequeños. Por ejemplo, la búsqueda binaria de un valor dado en una lista ordenada por valor. ¿Cuántas veces puede una lista de tamaño n ser divida a la mitad hasta que un sólo elemento quede en la lista final? La respuesta es $log_2\, n$.

Los logaritmos tienen las siguientes **propiedades**:

- $\log nm = \log n + \log m$
- $\log n/m = \log n - \log m$
- $\log n^r = r \log n$
- $\log_a n = \log_b n / \log_b a$ (para a y b enteros)

Esta última propiedad dice que el logaritmo de n en distintas bases, está relacionado por una constante (que es logaritmo de una base en la otra). Así que análisis de complejidad que se hacen en una base de logaritmos, pueden fácilmente traducirse en otra, simplemente con un factor de proporcionalidad.

Factoriales

La notación $n!$ se define para los enteros $n \geq 0$ como:

$$n! = \begin{cases} 1 & \text{si } n=0 \\ n \cdot (n-1)! & \text{si } n>0 \end{cases}$$

Entonces, $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots n$.

Una cota superior débil de la función factorial es $n! \leq n^n$, pues cada uno de los n términos en el producto factorial es a lo más n. La **aproximación de *Stirling***, proporciona una cota superior ajustada, y también una cota inferior:

$$n! = \text{sqrt}\{2pn\} \, (n/e)^n \, (1 + \mathbf{Q}(1/n))$$

donde e es la base del logaritmo natural. Usando la aproximación de *Stirling*, podemos demostrar que:

$$n! = \boldsymbol{o}(n^n)$$
$$n! = (2^n)$$
$$lg(n!) = \mathbf{Q}(n \, lg \, n)$$

Números de *Fibonacci*

Los **números de *Fibonacci*** se definen por la siguiente recurrencia:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{para } i \geq 2$$

Entonces cada número de *Fibonacci* es la suma de los números previos, produciendo la sucesión:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Recursión

Un algoritmo es **recursivo** si se llama a sí mismo para hacer parte del trabajo. Para que este enfoque sea exitoso, la llamada debe ser en un problema menor que al originalmente intentado.

En general, un algoritmo recursivo tiene dos partes:

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

Ejemplos de aplicación: cálculo del factorial, torres de *Hanoi* y función de *Ackermann*.

Sumatorias y recurrencias

Las **sumatorias** se usan mucho para el análisis de la complejidad de programas, en especial de ciclos, ya que realizan conteos de operaciones cada vez que se entra al ciclo.

Cuando se conoce una ecuación que calcula el resultado de una sumatoria, se dice que esta ecuación representa una **solución en forma cerrada**.

Ejemplos de soluciones en forma cerrada:

- $\Sigma\, i = n(n+1)/2$
- $\Sigma\, I^2 = (2n^3+3n^2+n)/6$
- $\Sigma_1^{\log n}\ n = n \log n$
- $\Sigma^{\infty}\, a^i = 1/(1-a)$

El tiempo de ejecución para un algoritmo recursivo está más fácilmente expresado por una expresión recursiva.

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

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

La secuencia de *Fibonacci*:

- Fib(n)=Fib(n-1)+Fib(n-2); Fib(1)=Fib(2)=1
- 1,1,2,3,5,8,13,...

Técnicas de Demostración Matemática

Prueba por contradicción

Es la forma más fácil de refutar un teorema o enunciado, es mediante un contra-ejemplo. Desafortunadamente, no importa el número de ejemplos que soporte un teorema, esto no es suficiente para probar que es correcto.

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

Prueba por inducción matemática

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

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

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

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

Ejemplo de demostración del teorema:

La suma de los primeros n números enteros es $n(n+1)/2$.

Estimación

Consiste en hacer estimaciones rápidas para resolver un problema. Puede formalizarse en tres pasos:

- Determine los parámetros principales que afectan el problema.
- Derive una ecuación que relacione los parámetros al problema.
- Seleccione valores para los parámetros, y aplique la ecuación para obtener una solución estimada.

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

Se estiman que 500 páginas de un libro requieren cerca de una pulgada en la repisa del librero, con lo cual da 2000 pulgadas de repisa. Lo cual da 167 pies de repisa (aprox. 200 pies). Si una repisa es alrededor de 4 pies de ancho, entonces se necesitan 50 repisas. Si un librero contiene 5 repisas, entonces se necesitan 10 libreros.

Capítulo 2

Algoritmos y Problemas

2.1 Definición de Algoritmo

El concepto intuitivo de algoritmo, lo tenemos prácticamente todos: *Un algoritmo es una serie finita de pasos para resolver un problema.*

Hay que hacer énfasis en dos aspectos para que un algoritmo exista:

1. El número de pasos debe ser finito. De esta manera el algoritmo debe terminar en un tiempo finito con la solución del problema,

2. El algoritmo debe ser capaz de determinar la solución del problema.

De este modo, podemos definir algoritmo como un "*conjunto de reglas operacionales inherentes a un cómputo*". Se trata de un método sistemático, susceptible de ser realizado mecánicamente, para resolver un problema dado.

Sería un error creer que los algoritmos son exclusivos de la informática. También son algoritmos los que aprendemos en la escuela para multiplicar y dividir números de varias cifras. De hecho, el algoritmo más famoso de la historia se remonta a la antigüedad: se trata del algoritmo de Euclides para calcular el máximo común divisor.

Siempre que se desee resolver un problema hay que plantearse qué algoritmo utilizar. La respuesta a esta cuestión puede depender de numerosos factores, a saber, el tamaño del problema, el modo en que está planteado y el tipo y la potencia del equipo disponible para su resolución.

Características de un algoritmo

1. *Entrada*: definir lo que necesita el algoritmo
2. *Salida*: definir lo que produce.
3. *No ambiguo*: explícito, siempre sabe qué comando ejecutar.
4. *Finito*: El algoritmo termina en un número finito de pasos.
5. *Correcto*: Hace lo que se supone que debe hacer. La solución es correcta
6. *Efectividad*: Cada instrucción se completa en tiempo finito. Cada instrucción debe ser lo suficientemente básica como para que en principio pueda ser ejecutada por cualquier persona usando papel y lápiz.

7. *General*: Debe ser lo suficientemente general como para contemplar todos los casos de entrada.

Así podemos, decir que un ***Algoritmo*** **es un conjunto finito de instrucciones precisas para resolver un problema**.

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

Un **problema** es una función o asociación de entradas con salidas. Un problema puede tener muchos algoritmos.

Por tanto, un **algoritmo** es un procedimiento para resolver un problema cuyos pasos son concretos y no ambiguos. El algoritmo debe ser correcto, de longitud finita y debe terminar para todas las entradas. Un **programa** es una instanciación de un algoritmo en un lenguaje de programación.

2.2 Formulación y Resolución de Problemas

Los algoritmos son los procedimientos que se construyen para la resolución de cualquier problema. De este modo, cuando se refiere a la construcción de un programa, nos estamos refiriéndo a la construcción de un algoritmo. El algoritmo no es un concepto proveniente del campo de la computación, sino que es un término matemático.

Los algoritmos los encontramos, o mejor, los ejecutamos a lo largo de nuestras actividades diarias; por ejemplo, cuando hacemos una llamada telefónica, tenemos en cuenta un conjunto de instrucciones mínimas y el orden en el cual debemos ejecutarlas para conseguir comunicarnos con alguien en particular; o cuando consultamos un diccionario, cuando se prepara un menú, etc.

Podemos conceptuar que un algoritmo es un procedimiento que contiene un conjunto finito de pasos que se deben ejecutar en un orden específico y lógico, para solucionar los problemas.

Un algoritmo puede ser caracterizado por una función lo cual asocia una salida: s= f (E) a cada entrada E.

Se dice entonces que un algoritmo calcula una función f. Entonces la entrada es una variable independiente básica en relación a la que se producen las salidas del algoritmo, y también los análisis de tiempo y espacio.

Cuando se tiene un problema para el cual debemos especificar un algoritmo solución, tendremos en cuenta varios puntos:

- Si no se conoce un método para solucionar el problema en cuestión, debemos hacer un análisis del mismo para llegar a una solución, después de evaluar alternativas, exigencias y excepciones.

- Si se conoce un "buen" método de solución al problema, se debe especificar exacta y completamente el método o procedimiento de solución en un lenguaje que se pueda interpretar fácilmente.

- Los problemas pueden agruparse en conjunto de problemas que son semejantes.

- En algún sentido, en general, un método para solucionar todos los problemas de un conjunto dado, se considera superior a un método para solucionar solamente un problema del conjunto.

- Hay criterios para determinar que tan "buena" es una solución, distintos de ver si trabaja o no, o hasta qué punto es general la aplicabilidad del método. Estos criterios involucran aspectos tales como: eficiencia, elegancia, velocidad, etc.

Al definir con exactitud un método de solución para un problema, este debe estar en capacidad de encontrarla si existe; en caso de que no exista, el método debe reconocer esta situación y suspender cualquier acción.

Formular y resolver problemas constituye una actividad esencial de la vida. Los modelos educativos se han dedicado a plantear problemas y enseñar como solucionarlos. Sin embargo, no se enseña nunca técnicas de cómo resolver problemas en general.

La formulación y resolución de problemas ha sido preocupación de los psicólogos cuando hablan de la inteligencia del hombre. Ellos han concebido esta capacidad como el concurso de ciertas destrezas mentales relacionadas con lo que ya se ha aprendido y que incluyen:

- Capacidad de analizar los problemas.
- La rapidez y precisión con que acude al pensamiento.
- Una organización de ideas para garantizar que se posee la información esencial que asegura el aprendizaje.
- Una retención clara del incremento de conocimiento.

No se trata de almacenar problemas y sus correspondientes soluciones, sino de desarrollar una capacidad para hacer frente con éxito a situaciones nuevas o desconocidas en la vida del hombre. Ante situaciones nuevas, el que no sabe buscar soluciones se sentirá confuso y angustiado y entonces no busca una estrategia y dará una primera solución para poner punto final a su agonía.

El que sabe buscar soluciones, selecciona la estrategia que le parece más cercana a la requerida y hace una hábil adaptación que se ajusta a la nueva demanda.

Al movernos dentro de la informática no siempre encontramos los problemas a punto de resolverlos, es necesario empezar por hacer su formulación. Se exige hacerlo en términos concisos y claros partiendo de las bases o supuestos que se dispone para especificar sus requerimientos.

Sólo a partir de una buena formulación será posible diseñar una estrategia de solución. Es necesario aprender a desligar estos dos procesos. No se deben hacer formulaciones pensando paralelamente en posibles soluciones. Es necesario darle consistencia e independencia para determinar con precisión a qué se quiere dar solución y luego canalizar una gama de alternativas posibles.

Si ya se ha realizado la formulación del problema podemos cuestionarla con el fin de entender bien la naturaleza del problema.

En nuestra área, el análisis de un problema tiene dos etapas claramente definidas y relacionadas:

- Formulación o planteamiento del problema.
- Resolución del problema.

A su vez, la formulación la podemos descomponer en tres etapas:

- Definición del problema.
- Supuestos: aserciones y limitaciones suministradas.
- Resultados esperados.

La fase de **planteamiento del problema** lo que pretende un algoritmo es sintetizar de alguna forma una tarea, cálculo o mecanismo antes de ser transcrito al computador. Los pasos que hay que seguir son los siguientes:

- Análisis previo del problema.
- Primera visión del método de resolución.
- Descomposición en módulos.
- Programación estructurada.
- Búsqueda de soluciones parciales.
- Ensamblaje de soluciones finales.

La fase de **resolución del problema** se puede descomponer en tres etapas:

- Análisis de alternativas y selección de la solución.
- Especificación detallada del procedimiento solución.

- Adopción o utilización de una herramienta para su implementación, si es necesaria.

Hay que notar que el computador es un medio y no es el fin en la solución de problemas.

En otras palabras, no es el computador el que soluciona los problemas, somos nosotros quienes lo hacemos y de alguna manera le contamos como es la cosa para que él con su velocidad y exactitud trabaje con grandes volúmenes de datos.

En el campo de las ciencias de la computación la solución de problemas se describe mediante el diseño de procedimientos llamados ***algoritmos***, los cuales posteriormente se implementan como ***programas***. Los ***programas*** son procedimientos que solucionan problemas y que se expresan en un lenguaje conocido por el computador.

Se pueden dar problemas que por su tamaño es necesario subdividirlos en problemas más simples para solucionarlos, utilizando la filosofía de "Dividir para conquistar". Se parte del principio de que es más fácil solucionar varios problemas simples como partes de un todo que seguir una implantación de "Todo o Nada". Desarrollar la capacidad de formular y resolver problemas nos prepara para enfrentar situaciones desconocidas.

2.3 Razones para Estudiar los Algoritmos

Con el logro de computadores cada vez más rápidos se podría caer en la tentación de preguntarse si vale la pena preocuparse por aumentar la eficiencia de los algoritmos. ¿No sería más sencillo aguardar a la siguiente generación de computadores?. Vamos a demostrar que esto no es así.

Supongamos que se dispone, para resolver un problema dado, de un algoritmo que necesita un tiempo exponencial y que, en un cierto computador, una implementación del mismo emplea $10^{-4} \times 2^n$ segundos. Este programa podrá resolver un ejemplar de tamaño $n=10$ en una décima de segundo. Necesitará casi 10 minutos para resolver un ejemplar de tamaño 20. Un día entero no bastará para resolver uno de tamaño 30. En un año de cálculo ininterrumpido, a duras penas se resolverá uno de tamaño 38. Imaginemos que necesitamos resolver ejemplares más grandes y compramos para ello un computador cien veces más rápido. El mismo algoritmo conseguirá resolver ahora un ejemplar de tamaño n en sólo $10^{-6} 2^n$ segundos. ¡Qué decepción al constatar que, en un año, apenas se consigue resolver un ejemplar de tamaño 45!

En general, si en un tiempo dado se podía resolver un ejemplar de tamaño n, con el nuevo computador se resolverá uno de tamaño $n+7$ en ese mismo tiempo.

Imaginemos, en cambio, que investigamos en algorítmica y encontramos un algoritmo capaz de resolver el mismo problema en un tiempo cúbico. La implementación de este algoritmo en el computador inicial podría necesitar, por ejemplo, 10^{-2} n^3 segundos. Ahora se podría resolver en un día un ejemplar de un tamaño superior a 200. Un año permitiría alcanzar casi el tamaño 1500.

Por tanto, el nuevo algoritmo no sólo permite una aceleración más espectacular que la compra de un equipo más rápido, sino que hace dicha compra más rentable.

Dado estos argumentos, podemos resumir las siguientes razones para justificar el estudiar los algoritmos:

1. Evitar reinventar la rueda.

   Para algunos problemas de programación ya existen buenos algoritmos para solucionarlos. Para esos algoritmos ya fueron analizadas sus propiedades. Por ejemplo, confianza en su correctitud y eficiencia.

2. Para ayudar cuando desarrollen sus propios algoritmos.

   No siempre existe un algoritmo desarrollado para resolver un problema. No existe regla general de creación de algoritmos. Muchos de los principios de proyecto de algoritmos ilustrados por algunos de los algoritmos que estudiaremos son importantes en todos los problemas de programación. El conocimiento de los algoritmos bien definidos provee una fuente de ideas que pueden ser aplicadas a nuevos algoritmos.

3. Ayudar a entender herramientas que usan algoritmos particulares. Por ejemplo, herramientas de compresión de datos:

   - *Pack* usa Códigos *Huffman*.
   - *Compress* usa *LZW*.
   - *Gzip* usa *Lempel-Ziv*.

4. Útil conocer técnicas empleadas para resolver problemas de determinados tipos.

2.4 Formas de Representación de Algoritmos

Existen diversas formas de representación de algoritmos, pero no hay un consenso con relación a cuál de ellas es mejor.

Algunas formas de representación de algoritmos tratan los problemas a un nivel lógico, abstrayéndose de detalles de implementación, muchas veces relacionados con un lenguaje de programación específico. Por otro lado, existen formas de representación de algoritmos que poseen una mayor

riqueza de detalles y muchas veces acaban por oscurecer la idea principal, el algoritmo, dificultando su entendimiento.

Dentro de las formas de representación de algoritmos más conocidas, sobresalen:

- La descripción narrativa
- El Flujograma convencional
- El diagrama Chapin
- El pseudocódigo, o también conocido como lenguaje estructurado.

2.5 La Máquina de *Turing*

Alan Turing, en 1936 desarrolló su **Máquina de *Turing*** (la cual se cubre en los cursos denominados *Teoría de la Computación o Teoría de Automátas*), estableciendo que cualquier algoritmo puede ser representado por ella.

Turing mostró también que existen problemas matemáticos bien definidos para los cuales no hay un algoritmo. Hay muchos ejemplos de problemas para los cuales no existe un algoritmo. Un problema de este tipo es el llamado problema de paro (*halting problem*):

> *Dado un programa de computadora con sus entradas, ¿parará este alguna vez?*

Turing probó que no hay un algoritmo que pueda resolver correctamente todas las instancias de este problema.

Alguien podría pensar en encontrar algunos métodos para detectar patrones que permitan examinar el programa para cualquier entrada. Sin embargo, siempre habrá sutilezas que escapen al análisis correspondiente.

Alguna persona más suspicaz, podría proponer simplemente correr el programa y reportar éxito si se alcanza una declaración de fin. Desgraciadamente, este esquema no garantiza por sí mismo un paro y en consecuencia el problema no puede ser resuelto con los pasos propuestos. Como consecuencia de acuerdo con la definición anterior, ese último procedimiento no es un algoritmo, pues no se llega a una solución.

Muchas veces aunque exista un algoritmo que pueda solucionar correctamente cualquier instancia de un problema dado, no siempre dicho algoritmo es satisfactorio porque puede requerir de tiempos exageradamente excesivos para llegar a la solución.

Capítulo 3

Eficiencia de Algoritmos

3.1 Introducción

Un objetivo natural en el desarrollo de un programa computacional es mantener tan bajo como sea posible el consumo de los diversos recursos, aprovechándolos de la mejor manera que se encuentre. Se desea un buen uso, eficiente, de los recursos disponibles, sin desperdiciarlos.

Para que un programa sea práctico, en términos de requerimientos de almacenamiento y tiempo de ejecución, debe organizar sus datos en una forma que apoye el procesamiento eficiente.

Siempre que se trata de resolver un problema, puede interesar considerar distintos algoritmos, con el fin de utilizar el más eficiente. Pero, ¿cómo determinar cuál es "el mejor"?. La estrategia **empírica** consiste en programar los algoritmos y ejecutarlos en un computador sobre algunos ejemplares de prueba. La estrategia **teórica** consiste en determinar matemáticamente la cantidad de recursos (tiempo, espacio, etc.) que necesitará el algoritmo ***en función del tamaño*** del ejemplar considerado.

El tamaño de un ejemplar x corresponde formalmente al número de dígitos binarios necesarios para representarlo en el computador. Pero a nivel algorítmico consideraremos el tamaño como el número de elementos lógicos contenidos en el ejemplar.

3.2 Concepto de Eficiencia

Un algoritmo es **eficiente** cuando logra llegar a sus objetivos planteados utilizando la menor cantidad de recursos posibles, es decir, minimizando el uso memoria, de pasos y de esfuerzo humano.

Un algoritmo es **eficaz** cuando alcanza el objetivo primordial, el análisis de resolución del problema se lo realiza prioritariamente.

Puede darse el caso de que exista un algoritmo eficaz pero no eficiente, en lo posible debemos de manejar estos dos conceptos conjuntamente.

La eficiencia de un programa tiene dos ingredientes fundamentales: **espacio** y **tiempo**.

- La ***eficiencia en espacio*** es una medida de la cantidad de memoria requerida por un programa.

- La ***eficiencia en tiempo*** se mide en términos de la cantidad de tiempo de ejecución del programa.

Ambas dependen del tipo de computador y compilador, por lo que no se estudiará aquí la eficiencia de los programas, sino la eficiencia de los algoritmos. Asimismo, este análisis dependerá de si trabajamos con máquinas de un solo procesador o de varios de ellos. Centraremos nuestra atención en los algoritmos para máquinas de un solo procesador que ejecutan una instrucción y luego otra.

3.3 Medidas de Eficiencia

Inventar algoritmos es relativamente fácil. En la práctica, sin embargo, no se pretende sólo diseñar algoritmos, si no más bien que **buenos** algoritmos. Así, el objetivo es inventar algoritmos y probar que ellos mismos son buenos.

La calidad de un algoritmo puede ser avalada utilizando varios criterios. Uno de los criterios más importantes es el tiempo utilizado en la ejecución del algoritmos. Existen varios aspectos a considerar en cada criterio de tiempo. Uno de ellos está relacionado con el **tiempo de ejecución** requerido por los diferentes algoritmos, para encontrar la solución final de un problema o cálculo particular.

Normalmente, un problema se puede resolver por métodos distintos, con diferentes **grados de eficiencia**. Por ejemplo: búsqueda de un número en una guía telefónica.

Cuando se usa un computador es importante limitar el consumo de recursos.

Recurso Tiempo:

- Aplicaciones informáticas que trabajan "**en tiempo real**" requieren que los cálculos se realicen en el menor tiempo posible.

- Aplicaciones que manejan un **gran volumen de información** si no se tratan adecuadamente pueden necesitar tiempos impracticables.

Recurso Memoria:

- Las máquinas tienen una **memoria limitada**.

3.4 Análisis *A Priori* y Prueba *A Posteriori*

El análisis de la eficiencia de los algoritmos (memoria y tiempo de ejecución) consta de dos fases: Análisis *A Priori* y Prueba *A Posteriori.*

El **Análisis *A Priori (o teórico)*** entrega una función que limita el tiempo de cálculo de un algoritmo. Consiste en obtener una expresión que indique el comportamiento del algoritmo en función de los parámetros que influyan. Esto es interesante porque:

- La predicción del costo del algoritmo puede evitar una implementación posiblemente laboriosa.
- Es aplicable en la etapa de diseño de los algoritmos, constituyendo uno de los factores fundamentales a tener en cuenta.

En la **Prueba *A Posteriori (experimental o empírica)*** se recogen estadísticas de tiempo y espacio consumidas por el algoritmo mientras se ejecuta. La estrategia empírica consiste en programar los algoritmos y ejecutarlos en un computador sobre algunos ejemplares de prueba, haciendo medidas para:

- una máquina concreta,
- un lenguaje concreto,
- un compilador concreto y
- datos concretos

La estrategia teórica tiene como ventajas que no depende del computador ni del lenguaje de programación, ni siquiera de la habilidad del programador. Permite evitar el esfuerzo inútil de programar algoritmos ineficientes y de despediciar tiempo de máquina para ejecutarlos. También permite conocer la eficiencia de un algoritmo cualquiera que sea el tamaño del ejemplar al que se aplique.

3.5 Concepto de Instancia

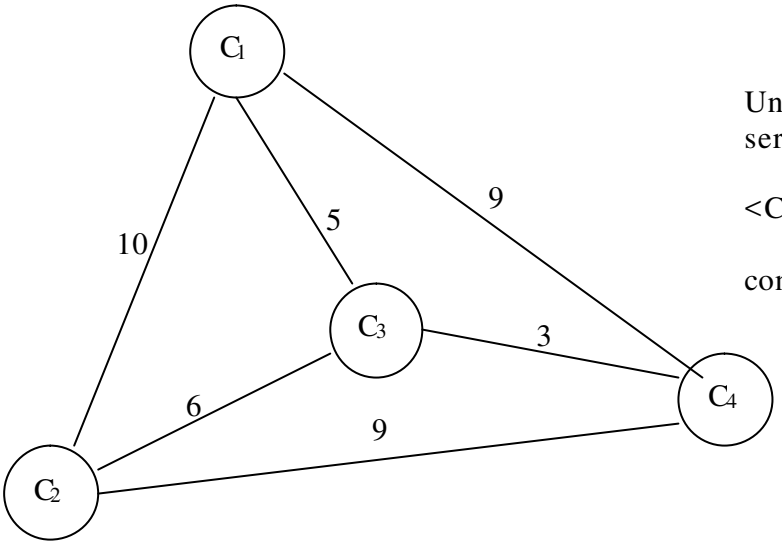
Un **problema computacional** consiste en una caracterización de un conjunto de datos de entrada, junto con la especificación de la salida deseada en base a cada entrada.

Un problema computacional tiene una o más **instancias**, que son valores particulares para los datos de entrada, sobre los cuales se puede ejecutar el algoritmo para resolver el problema.

Ejemplo: el problema computacional de *multiplicar dos números enteros* tiene por ejemplo, las siguientes instancias: multiplicar 345 por 4653, multiplicar 2637 por 10000, multiplicar -32341 por 12, etc.

Un problema computacional abarca a otro problema computacional si las instancias del segundo pueden ser resueltas como instancias del primero en forma directa.

Ejemplo: Problema del Vendedor Viajero.



Una instancia de solución sería:

<C1, C2, C4, C3>

con una longitud de 27.

3.6 Tamaño de los Datos

Variable o expresión en función de la cual intentaremos medir la complejidad del algoritmo.

Es claro que para cada algoritmo la cantidad de recurso (tiempo, memoria) utilizados depende fuertemente de los datos de entrada. En general, la cantidad de recursos crece a medida que crece el tamaño de la entrada.

El análisis de esta cantidad de recursos no es viable de ser realizado instancia por instancia.

Se definen entonces las funciones de cantidad de recursos en base al **tamaño (o talla) de la entrada**. Suele depender del número de datos del problema. Este tamaño puede ser la cantidad de dígitos para un número, la cantidad de elementos para un arreglo, la cantidad de caracteres de una cadena, en problemas de ordenación es el número de elementos a ordenar, en matrices puede ser el número de filas, columnas o elementos totales, en algoritmos recursivos es el número de recursiones o llamadas propias que hace la función.

En ocasiones es útil definir el tamaño de la entrada en base a dos o más magnitudes. Por ejemplo, para un grafo es frecuente utilizar la cantidad de nodos y de arcos.

En cualquier caso, se debe elegir la misma variable para comparar algoritmos distintos aplicados a los mismos datos.

3.7 Cálculo de Costos de Algoritmos

Queremos saber la eficiencia de los algoritmos, no del computador. Por ello, en lugar de medir el tiempo de ejecución en microsegundos o algo por el estilo, nos preocuparemos del número de veces que se ejecuta una operación primitiva (de tiempo fijo).

Para estimar la eficiencia de este algoritmo, podemos preguntarnos, "si el argumento es una frase de *N* números, ¿cuántas multiplicaciones realizaremos?" La respuesta es que hacemos una multiplicación por cada número en el argumento, por lo que hacemos *N* multiplicaciones. La cantidad de tiempo que se necesitaría para el doble de números sería el doble.

3.7.1 Cálculo de eficiencia en análisis iterativo

Cuando se analiza la eficiencia, en tiempo de ejecución, de un algoritmo son posibles distintas aproximaciones**:** desde el cálculo detallado (que puede complicarse en muchos algoritmos si se realiza un análisis para distintos contenidos de los datos de entrada, casos más favorables, caso peor, hipótesis de distribución probabilística para los contenidos de los datos de entrada, etc.) hasta el análisis asintótico simplificado aplicando reglas prácticas.

En estos apuntes se seguirá el criterio de análisis asintótico simplificado, si bien nunca se ha de dejar de aplicar el sentido común. Como en todos los modelos simplificados se ha mantener la alerta para no establecer hipótesis simplificadoras que no se correspondan con la realidad.

En lo que sigue se realizará una exposición basada en el criterio de exponer en un apartado inicial una serie de reglas y planteamientos básicos aplicables al análisis de eficiencia en algoritmos iterativos, en un segundo apartado se presenta una lista de ejemplos que permitan comprender algunas de las aproximaciones y técnicas expuestas.

3.7.2 Cálculo de eficiencia en análisis recursivo

Retomando aquí el socorrido ejemplo del factorial, tratemos de analizar el coste de dicho algoritmo, en su versión **iterativa** , se tiene:

```
PROCEDURE Factorial(n : CARDINAL) : CARDINAL
BEGIN
  VAR Resultado,i : CARDINAL ;
  Resultado :=1 ;
  FOR i :=1 TO n DO
    Resultado :=Resultado*i;
```

```
  END ;
  RETURN Resultado
END Factorial ;
```

Aplicando las técnicas de análisis de coste en algoritmos iterativos de forma rápida y mentalmente (es como se han de llegar a analizar algoritmos tan simples como éste), se tiene: hay una inicialización antes de bucle, de coste constante. El bucle se repite un número de veces *n* y en su interior se realiza una operación de coste constante. Por tanto el algoritmo es de coste lineal o expresado con algo más de detalle y rigor, si la función de coste del algoritmo se expresa por $T(n)$, se tiene que $T(n)$? T.

Una versión **recursiva** del mismo algoritmo, es:

```
PROCEDURE Factorial(n: CARDINAL): CARDINAL;
BEGIN
  IF n=0 THEN
    RETURN 1
  ELSE
    RETURN ( n* Factorial(n-1) )
  END
END Factorial;
```

Al aplicar el análisis de coste aprendido para análisis de algoritmos iterativos se tiene: hay una instrucción de alternativa, en una de las alternativas simplemente se devuelve un valor (operación de coste constante). En la otra alternativa se realiza una operación de coste constante (multiplicación) con dos operandos. El primer operando se obtiene por una operación de coste constante (acceso a la variable *n*), el coste de la operación que permite obtener el segundo operando es el coste que estamos calculando!, es decir es el coste de la propia función factorial (solo que para parámetro ***n*-1**).

Por tanto, para conocer el orden de la función de coste de este algoritmo ¿debemos conocer previamente el orden de la función de coste de este algoritmo?, entramos en una **recurrencia**.

Y efectivamente, el asunto está en saber resolver recurrencias. Si $T(n)$ es la función de coste de este algoritmo se puede decir que $T(n)$ es igual a una operación de coste constante c cuando n vale **0** y a una operación de coste $T(n-1)$ más una operación de coste constante (el acceso a n y la multiplicación) cuando n es mayor que **0**, es decir:

$$T(n) = \begin{cases} c & \text{si } n = 0 \\ T(n-1) + c & \text{si } n > 0 \end{cases}$$

Se trata entonces de encontrar soluciones a recurrencias como ésta. Entendiendo por solución una función simple $f(n)$ tal que se pueda asegurar que $T(n)$ es del orden de $f(n)$.

En este ejemplo puede comprobarse que $T(n)$ es de orden lineal, es decir del orden de la función $f(n)=n$, ya que cualquier función lineal $T(n)= an+b$ siendo a y b constantes, es solución de la **recurrencia**:

$T(0)= b$, es decir una constante

$T(n)= a.n+b= an-a+ b+a= a(n-1)+b+a= T(n-1)+a$, es decir el coste de $T(n)$ es igual al de $T(n-1)$ más una constante.

3.8 Principio de Invarianza

Dado un algoritmo y dos implementaciones I_1 y I_2 (máquinas distintas o códigos distintos) que tardan $T_1(n)$ y $T_2(n)$ respectivamente, el principio de invarianza afirma que existe una constante real $c>0$ y un número natural n_0 tales que para todo $n>=n_0$ se verifica que $T_1(n)<=c\cdot T_2(n)$.

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.

Asumimos que un algoritmo tarda un tiempo del orden de $T(n)$ si existen una constante real $c>0$ y una implementación I del algoritmo que tarda menos que $c\cdot T(n)$, para todo n tamaño de entrada.

El comportamiento de un algoritmo puede variar notablemente para diferentes secuencias de entrada. Suelen estudiarse tres casos para un mismo algoritmo: caso mejor, caso peor, caso medio.

3.9 Análisis Peor Caso, Mejor Caso y Caso Promedio

Puede analizarse un algoritmo particular o una clase de ellos. Una clase de algoritmo para un problema son aquellos algoritmos que se pueden clasificar por el tipo de operación fundamental que realizan.

Ejemplo:

Problema: Ordenamiento
Clase: Ordenamiento por comparación

Para algunos algoritmos, diferentes entradas (*inputs*) para un tamaño dado pueden requerir diferentes cantidades de tiempo.

Por ejemplo, consideremos el problema de encontrar la posición particular de un valor K, dentro de un arreglo de n elementos. Suponiendo que sólo ocurre una vez. Comentar sobre el mejor, peor y caso promedio.

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

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

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

Tipos de análisis:

- **Peor caso:** indica el mayor tiempo obtenido, teniendo en consideración todas las entradas posibles.
- **Mejor caso:** indica el menor tiempo obtenido, teniendo en consideración todas las entradas posibles.
- **Media:** indica el tiempo medio obtenido, considerando todas las entradas posibles.

Como no se puede analizar el comportamiento sobre todas las entradas posibles, va a existir para cada problema particular un análisis en él:

- peor caso
- mejor caso
- caso promedio (o medio)

El caso promedio es la medida más realista de la *performance*, pero es más difícil de calcular pues establece que todas las entradas son igualmente probables, lo cual puede ser cierto o no. Trabajaremos específicamente con el peor caso.

Ejemplo

Sea *A* una lista de *n* elementos $A_1, A_2, A_3, \ldots, A_n$. Ordenar significa permutar estos elementos de tal forma que los mismos queden de acuerdo con un orden preestablecido.

Ascendente $A_1 <= A_2 <= A_3 \ldots\ldots\ldots <= A_n$

Descendente $A_1 >= A_2 >= \ldots\ldots\ldots >= A_n$

Caso peor: Que el vector esté ordenado en sentido inverso.

Caso mejor: Que el vector esté ordenado.

Caso medio: Cuando el vector esté desordenado aleatoriamente.

Capítulo 4

Análisis de Algoritmos

4.1 Introducción

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

- El diseño de un algoritmo que sea fácil de entender, codificar y depurar (Ingeniería de *Software*).
- El diseño de un algoritmo que haga uso eficiente de los recursos de la computadora (Análisis y Diseño de algoritmos).

El **análisis de algoritmos** nos permite medir la dificultad inherente de un problema y evaluar la eficiencia de un algoritmo.

4.2 Tiempos de Ejecución

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

Así, un trozo sencillo de programa como:

*S*1; *FOR i*:= 1 *TO N DO S*2 *END*;

requiere:

$T(N) := t1 + t2*N$

siendo $t1$ el tiempo que lleve ejecutar la serie "*S*1" de sentencias, y $t2$ el que lleve la serie "*S*2".

Prácticamente todos los programas reales incluyen alguna sentencia condicional, haciendo que las sentencias efectivamente ejecutadas dependan de los datos concretos que se le presenten. Esto hace que más que un valor $T(N)$ debamos hablar de un rango de valores:

$Tmin(N) <= T(N) <= Tmax(N)$

los extremos son habitualmente conocidos como "*caso peor*" y "*caso mejor*".

Entre ambos se hallará algún "*caso promedio*" o más frecuente. Cualquier fórmula $T(N)$ incluye referencias al parámetro N y a una serie de constantes "T_i" 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 computador que lo ejecuta.

Dado que es fácil cambiar de compilador y que la potencia de los computadores crece a un ritmo vertiginoso (en la actualidad, se duplica anualmente), intentaremos analizar los algoritmos con algún nivel de independencia de estos factores; es decir, buscaremos estimaciones generales ampliamente válidas.

No se puede medir el tiempo en segundos porque no existe un computador estándar de referencia, en su lugar medimos el número de **operaciones básicas o elementales**.

Las operaciones básicas son las que realiza el computador en tiempo acotado por una constante, por ejemplo:

- Operaciones aritméticas básicas
- Asignaciones de tipos predefinidos
- Saltos (llamadas a funciones, procedimientos y retorno)
- Comparaciones lógicas
- Acceso a estructuras indexadas básicas (vectores y matrices)

Es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentencias, por ejemplo, las que más influyen en el tiempo de ejecución.

Para medir el tiempo de ejecución de un algoritmo existen varios métodos. Veamos algunos de ellos:

a) Benchmarking

La técnica de *benchmark* considera una colección de entradas típicas representativas de una carga de trabajo para un programa.

b) Profiling

Consiste en asociar a cada instrucción de un programa un número que representa la fracción del tiempo total tomada para ejecutar esa instrucción particular. Una de las técnicas más conocidas (e informal) es la *Regla 90-10*, que afirma que el 90% del tiempo de ejecución se invierte en el 10% del código.

c) Análisis

Consiste en agrupar las entradas de acuerdo a su tamaño, y estimar el tiempo de ejecución del programa en entradas de ese tamaño, *T(n)*. Esta es la técnica que se estudiará en el curso.

De este modo, el **tiempo de ejecución** puede ser definido como una función de la entrada. Denotaremos *T(n)* como el tiempo de ejecución de un algoritmo para una entrada de tamaño *n*.

4.3 Concepto de Complejidad

La complejidad (o costo) de un algoritmo es una medida de la cantidad de recursos (tiempo, memoria) que el algoritmo necesita. La complejidad de un algoritmo se expresa en función del tamaño (o talla) del problema.

La función de complejidad tiene como variable independiente el tamaño del problema y sirve para medir la complejidad (espacial o temporal). Mide el tiempo/ espacio relativo en función del tamaño del problema.

El comportamiento de la función determina la eficiencia. No es única para un algoritmo: depende de los datos. Para un mismo tamaño del problema, las distintas presentaciones iniciales de los datos dan lugar a distintas funciones de complejidad. Es el caso de una ordenación si los datos están todos inicialmente desordenados, parcialmente ordenados o en orden inverso.

Ejemplo: $f(n)= 3n^2+2n+1$, en donde n es el tamaño del problema y expresa el tiempo en unidades de tiempo.

¿Una computadora más rápida o un algoritmo más rápido?

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

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

Si la razón de crecimiento es **lineal** (es decir, $T(n)$=cn) entonces por ejemplo, 100.000 números serán procesados en la nueva máquina en el mismo tiempo que 10.000 números en la antigua computadora.

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

Por ejemplo, supongamos que una computadora resuelve un problema de tamaño n en una hora. Ahora supongamos que tenemos una computadora

10 veces más rápida, ¿de qué tamaño es el problema que podemos resolver?

$f(n)$	n	n'	cambio	n'/n
$10n$	1000	10000	$n' = 10n$	10
$20n$	500	5000	$n' = 10n$	10
$5n \log n$	250	1842	$\sqrt{(10)}n$ $< n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{(10)}n$	3.16
2^n	13	16	$n' = n+3$	----

En la tabla de arriba, $f(n)$ es la **razón de crecimiento** de un algoritmo.

Supongamos que tenemos una computadora que puede ejecutar 10.000 operaciones básicas en una hora. La segunda columna muestra el máximo valor de n que puede ejecutarse con 10.000 operaciones básicas en una hora. Es decir $f(n)$="total de operaciones básicas en un intervalo de tiempo". Por ejemplo, $f(n)$= 10.000 operaciones básicas por hora.

Si suponemos que tenemos una computadora 10 veces más rápida, entonces podremos ejecutar 100.000 operaciones básicas en una hora. Por lo cual $f(n')$=100.000 operaciones básicas por hora.

Ejercicio: Comentar sobre la razón n'/n según el incremento de velocidad de la computadora y la razón de crecimiento del algoritmo en cuestión.

Nota: los factores constantes nunca afectan la mejora relativa obtenida por una computadora más rápida.

4.4 Órdenes de Complejidad

Se dice que $O(f(n))$ define un **"orden de complejidad"**. Escogeremos como representante de este orden a la función $f(n)$ más sencilla del mismo.

Así tendremos:

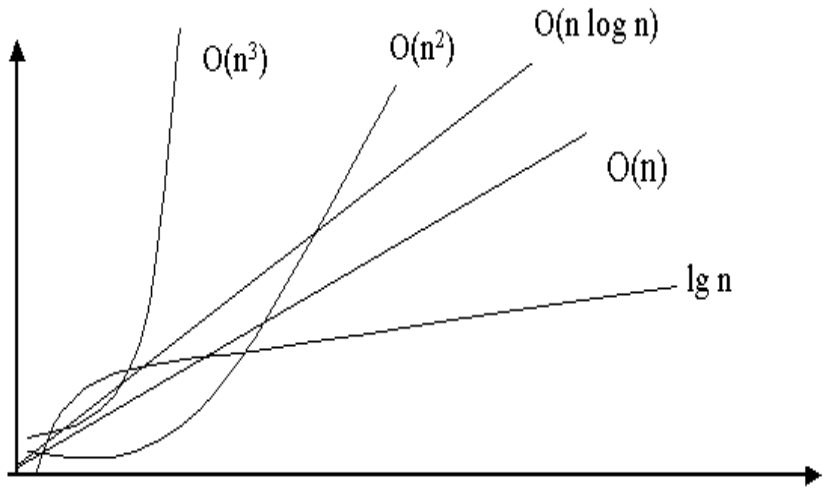
$O(1)$	orden constante
$O(\log n)$	orden logarítmico
$O(n)$	orden lineal
$O(n^2)$	orden cuadrático
$O(n^a)$	orden polinomial ($a > 2$)
$O(a^n)$	orden exponencial ($a > 2$)
$O(n!)$	orden factorial

Es más, se puede identificar una *jerarquía de órdenes de complejidad* que coincide con el orden de la tabla anterior; jerarquía en el sentido de que cada orden de complejidad superior tiene a los inferiores como

subconjuntos. Si un algoritmo *A* se puede demostrar de un cierto orden O_1, es cierto que también pertenece a todos los órdenes superiores (la relación de orden cota superior es transitiva); pero en la práctica lo útil es encontrar la "menor cota superior", es decir el menor orden de complejidad que lo cubra.

Antes de realizar un programa conviene elegir un buen algoritmo, donde por bueno entendemos que utilice pocos recursos, siendo usualmente los más importantes el tiempo que lleve ejecutarse y la cantidad de espacio en memoria que requiera. Es engañoso pensar que todos los algoritmos son "más o menos iguales" y confiar en nuestra habilidad como programadores para convertir un mal algoritmo en un producto eficaz. Es asimismo engañoso confiar en la creciente potencia de las máquinas y el abaratamiento de las mismas como remedio de todos los problemas que puedan aparecer.

Un ejemplo de algunas de las funciones más comunes en análisis de algoritmos son:



La mejor técnica para diferenciar la eficiencia de los algoritmos es el estudio de los **órdenes de complejidad**. El orden de complejidad se expresa generalmente en términos de la cantidad de datos procesados por el programa, denominada ***n***, que puede ser el tamaño dado o estimado.

En el análisis de algoritmos se considera usualmente el caso peor, si bien a veces conviene analizar igualmente el caso mejor y hacer alguna estimación sobre un caso promedio. Para independizarse de factores coyunturales, tales como el lenguaje de programación, la habilidad del codificador, la máquina de soporte, etc. se suele trabajar con un **cálculo asintótico** que indica cómo se comporta el algoritmo para datos muy grandes y salvo algún coeficiente multiplicativo. Para problemas pequeños es cierto que casi todos los algoritmos son "más o menos iguales", primando otros aspectos como esfuerzo de codificación, legibilidad, etc. Los órdenes de complejidad sólo son importantes para grandes problemas.

4.5 Notación Asintótica

El interés principal del análisis de algoritmos radica en saber cómo crece el tiempo de ejecución, cuando el tamaño de la entrada crece. Esto es la *eficiencia asintótica* del algoritmo. Se denomina "asintótica" porque analiza el comportamiento de las funciones en el *límite*, es decir, su tasa de crecimiento.

La notación asintótica se describe por medio de una función cuyo dominio es los números naturales (N) estimado a partir de tiempo de ejecución o de espacio de memoria de algoritmos en base a la longitud de la entrada. Se consideran las funciones asintóticamente no negativas.

La notación asintótica captura el **comportamiento** de la función para valores grandes de *N*.

Las notaciones no son dependientes de los tres casos anteriormente vistos, es por eso que una notación que determine el peor caso puede estar presente en una o en todas las situaciones.

4.5.1 La *O* Mayúscula

La notación *O* se utiliza para comparar funciones. Resulta particularmente útil cuando se quiere analizar la complejidad de un algoritmo, en otras palabras, la cantidad de tiempo que le toma a un computador ejecutar un programa.

Definición: Sean f y g funciones con dominio en $\boldsymbol{R} \leq 0$ o $\boldsymbol{N}$ es imagen en $\boldsymbol{R}$. Si existen constantes C y k tales que:

$$\forall\, x > k, |f(x)| \leq C\,|g(x)|$$

es decir, que para $x > k, f$ es menor o igual a un multiplo de g, decimos que:

$$f(x) = O(g(x))$$

La definición formal es:

$$f(x) = O(g(x)) \Longleftrightarrow \exists\, k, N \mid \forall\, x > N, |f(x)| \leq k\,|g(x)|$$

¿Qué quiere decir todo esto? Básicamente, que una función es siempre menor que otra función (por ejemplo, el tiempo en ejecutar tu programa es menor que x^2) si no tenemos en cuenta los factores constantes (eso es lo que significa la k) y si no tenemos en cuenta los valores pequeños (eso es lo que significa la N).

¿Por qué no tenemos en cuenta los valores pequeños de N? Porqué para entradas pequeñas, el tiempo que tarda el programa no es significativo y casi siempre el algoritmo será suficientemente rápido para lo que queremos.

Así $3N^3 + 5N^2 - 9 = O(N^3)$ no significa que existe una función $O(N^3)$ que es igual a $3N^3 + 5N^2 - 9$.

Debe leerse como:

"$3N^3+5N^2-9$ es *O-Grande* de N^3"

que significa:

"$3N^3+5N^2-9$ está **asintóticamente** dominada por N^3"

La notación puede resultar un poco confusa. Veamos algunos ejemplos.

Ejemplo 1: Muestre que $3N^3 + 5N^2 - 9 = O(N^3)$.

De nuestra experiencia anterior pareciera tener sentido que $C = 5$.
Encontremos k tal que:

$$3N^3 + 5N^2 - 9 \leq 5N^3 \text{ for } N > k :$$

1. Agrupamos: $5N^2 = 2N^3 + 9$
2. Cuál k asegura que $5N^2 = N^3$ para $N > k$?
3. $k = 5$
4. Así que para $N > 5$, $5N^2 = N^3 = 2N^3 + 9$.
5. $C = 5$, $k = 5$ (no es único!)

Ejemplo 2: Muestre que $N^4 \neq O(3N^3 + 5N^2 - 9)$.

Hay que mostrar que no existen C y k tales que para $N > k$, $C*(3N^3 + 5N^2 - 9) \geq N^4$ es siempre cierto (usamos límites, recordando el curso de Cálculo).

$$\lim_{x \to \infty} x^4 / C(3x^3 + 5x^2 - 9) = \lim_{x \to \infty} x / C(3 + 5/x - 9/x^3)$$

$$= \lim_{x \to \infty} x / C(3 + 0 - 0) = (1/3C) \cdot \lim_{x \to \infty} x = \infty$$

Así que sin importar cual sea el C que se elija, N^4 siempre alcanzará y rebasará $C*(3N^3 + 5N^2 - 9)$.

Podemos considerarnos afortunados porque sabemos calcular límites! Límites serán útiles para probar relaciones (como veremos en el próximo teorema).

Lema: Si el límite cuando $x \to \infty$ del cociente $|f(x)/g(x)|$ existe (es finito) entonces $f(x) = O(g(x))$.

Ejemplo 3: $3N^3 + 5N^2 - 9 = O(N^3)$. Calculamos:

$$\lim_{x \to \infty} x^3 / (3x^3 + 5x^2 - 9) = \lim_{x \to \infty} 1 / (3 + 5/x - 9/x^3) = 1/3$$

Listo!

4.5.2 La *o* Minúscula

La cota superior asintótica dada por la notación $\boldsymbol{O}$ puede o no ser ajustada asintóticamente. La cota $2n^2 = \boldsymbol{O}(n^2)$ es ajustada asintóticamente, pero la cota $2n = \boldsymbol{O}(n^2)$ no lo es. Utilizaremos la notación $\boldsymbol{o}$ para denotar una cota superior que no es ajustada asintóticamente.

Definimos formalmente $\boldsymbol{o}(g(n))$ ("o pequeña") como el conjunto:

$\boldsymbol{o}(g(n)) = \{f(n)$: para cualquier constante positiva $c > 0$, existe una constante $n_0 > 0$ tal que: $0 \leq f(n) \leq cg(n)$ para toda $n \geq n_0 \}$.

Por ejemplo, $2n = o(n^2)$, pero $2n^2$ no pertenece a $o(n^2)$.

Las notaciones de $\boldsymbol{O}$ y $\boldsymbol{o}$ son similares. La diferencia principal es, que en $f(n) = \boldsymbol{O}(g(n))$, la cota $0 \leq f(n) \leq cg(n)$ se cumple para *alguna* constante $c > 0$, pero en $f(n) = \boldsymbol{o}(g(n))$, la cota $0 \leq f(n) \leq cg(n)$ se cumple para *todas* las constantes $c > 0$. Intuitivamente en la notación $\boldsymbol{o}$, la función $f(n)$ se vuelve insignificante con respecto a $g(n)$ a medida que n se acerca a infinito, es decir:

$$\lim_{x \to \infty} (f(n)/g(n)) = 0.$$

4.5.3 Diferencia entre *O* y *o*

Para *o* la desigualdad se mantiene para todas las constantes positivas, mientras que para *O* la desigualdad se mantiene sólo para algunas constantes positivas.

4.5.4 Las Notaciones Ω y Θ

Ω Es el reverso de *O*.

$$f(x) = \Omega(g(x)) \rightarrow\leftarrow g(x) = O(f(x))$$

Ω Grande dice que asintóticamente $f(x)$ *domina a* $g(x)$.

Θ Grande dice que ambas funciones se dominan mutuamente, en otras palabras, son asintóticamente equivalentes.

$$f(x) = \Theta(g(x))$$
$$\rightarrow\leftarrow$$
$$f(x) = O(g(x)) \wedge f(x) = \Omega(g(x))$$

$f = \Theta(g)$: *"f es de orden g"*

4.5.5 Propiedades y Cotas más Usuales

<u>Propiedades de las notaciones asintóticas</u>

Relaciones de orden

La notación asintótica nos permite definir relaciones de orden entre el conjunto de las funciones sobre los enteros positivos:

- $f \in O(g(x)) \leftrightarrow f \leq g$ (se dice que *f* es asintóticamente menor o igual que *g*)
- $f \in o(g(x)) \leftrightarrow f < g$
- $f \in \Theta(g(x)) \leftrightarrow f = g$
- $f \in \Omega(g(x)) \leftrightarrow f \geq g$
- $f \in \omega(g(x)) \leftrightarrow f > g$

Relaciones entre cotas

1. $f(n) \in O(f(n))$
2. $f(n) \in O(f(n)) \wedge g(n) \in O(f(n)) \Rightarrow f(n) \in O(h(n))$
3. $f(n) \in O(g(n)) \Rightarrow g(n) \in \Omega(h(n))$
4. $\lim_{n \to \infty} f(n)/g(n) = 0 \Rightarrow O(f(n)) \subset O(g(n))$
5. $O(f(n)) = O(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge g(n) \in O(f(n))$
6. $O(f(n)) \subset O(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge g(n) \notin O(f(n))$
7. $\Omega(f(n)) \subset \Omega(g(n)) \Leftrightarrow f(n) \in \Omega(g(n)) \wedge g(n) \notin \Omega(f(n))$

Relaciones de inclusión

Son muy útiles a la hora de comparar funciones de coste en el caso asintótico.

$\forall x, y, a, \varepsilon \in R^{>0}$

1. $\log_a n \in O(log_b n)$
2. $O(\log_a{}^x n) \subset O(\log_a{}^{x+\delta} n)$
3. $O(\log_a{}^x n) \subset O(n)$
4. $O(n^x) \subset O(n^{x+\delta})$
5. $O(n^x \log_a{}^y n) \subset O(n^{x+\delta})$
6. $O(n^x) \subset O(2^n)$

Propiedades de clausura

El conjunto de funciones del orden de $f(n)$ es cerrado respecto de la suma y la multiplicación de funciones:

1. $f_1(n) \in O(g_1(n)) \ \wedge f_2(n) \in O(g_2(n)) \Rightarrow f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$
2. $f_1(n) \in O(g_1(n)) \ \wedge f_2(n) \in O(g_2(n)) \Rightarrow f_1(n) * f_2(n) \in O(g_1(n) * g_2(n))$
3. $f_1(n) \in O(g_1(n)) \wedge f_2(n) \in O(g_2(n)) \Rightarrow f_1(n)+f_2(n) \in O(max(g_1(n), g_2(n))$

como consecuencia es que los polinomios de grado k en n son exactamente del orden de nk

$$a_k n^k + \ldots + a_1 n + a_0 \in \Theta(n^k)$$

Cotas Asintóticas Más Usuales

1. $c \in \Theta(1) \ \forall c \in R \geq 0$
2. $\sum_{i=1}^{n} i = (n/2)(n+1) \in \Theta(n^2)$
3. $\sum_{i=1}^{n} i^2 = (n/3)(n+1)(n+1/2) \in \Theta(n^3)$
4. $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1}) \ \forall k \in N$
5. $\sum_{i=1}^{n} \log i \in \Theta(n \log n)$
6. $\sum_{i=1}^{n} (n-i)^k \in \Theta(n^{k+1}) \ ) \ \forall k \in N$

4.6 Ecuaciones de Recurrencias

4.6.1 Introducción

Como ya hemos indicado, el análisis del tiempo de ejecución de un programa recursivo vendrá en función del tiempo requerido por la(s) llamada(s) recursiva(s) que aparezcan en él. De la misma manera que la verificación de programas recursivos requiere de razonar por inducción, para tratar apropiadamente estas llamadas, el cálculo de su eficiencia requiere un concepto análogo: el de ecuación de recurrencia. Demostraciones por inducción y ecuaciones de recurrencia son por tanto los conceptos básicos para tratar programas recursivos.

Supongamos que se está analizando el coste de un algoritmo recursivo, intentando calcular una función que indique el uso de recursos, por ejemplo el tiempo, en términos del tamaño de los datos; denominémosla $T(n)$ para datos de tamaño n. Nos encontramos con que este coste se define a su vez en función del coste de las llamadas recursivas, es decir, de $T(m)$ para otros tamaños m (usualmente menores que n). Esta manera de expresar el coste T en función de sí misma es lo que denominaremos una *ecuación recurrente* y su resolución, es decir, la obtención para T de una fórmula cerrada (independiente de T) puede ser dificultosa.

4.6.2 Resolución de Recurrencias

Las ecuaciones de recurrencia son utilizadas para determinar cotas asintóticas en algoritmos que presentan recursividad.

Veremos dos técnicas básicas y una auxiliar que se aplican a diferentes clases de recurrencias:

- Método del teorema maestro
- Método de la ecuación característica
- Cambio de variable

No analizaremos su demostración formal, sólo consideraremos su aplicación para las recurrencias generadas a partir del análisis de algoritmos.

4.6.3 Método del Teorema Maestro

Se aplica en casos como:

$$T(n) = \begin{cases} 5 & \text{si } n=0 \\ 9T(n/3) + n & \text{si } n \neq 0 \end{cases}$$

Teorema: Sean $a = 1$, $b > 1$ constantes, $f(n)$ una función y $T(n)$ una recurrencia definida sobre los enteros no negativos de la forma $T(n) = aT(n/b) + f(n)$, donde n/b puede interpretarse como $\lceil n/b \rceil$ o $\lfloor n/b \rfloor$.

Entonces valen:

1. Si $f(n) \in O(m^{\log_b a - \varepsilon})$ para algún $\varepsilon > 0$ entonces $T(n) \in \Theta(n^{\log_b a})$.
2. Si $f(n) \in \Theta(n^{\log_b a})$ entonces $T(n) \in \Theta(n^{\log_b a} \lg n)$.
3. Si $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ para algún $\varepsilon > 0$, y satisface $af(n/b) \leq cf(n)$ para alguna constante $c < 1$, entonces $T(n) \in \Theta(f(n))$.

Ejemplos:

1. Si $T(n)=9T(n/3)+n$ entonces $a=9$, $b=3$, se aplica el caso 1 con $\varepsilon=1$ y $T(n) \in \Theta(n^2)$.

2. Si $T(n)=T(2n/3)+1$ entonces $a=1$, $b=3/2$, se aplica el caso 2 y $T(n) = \Theta(\lg n)$.

3. Si $T(n)=3T(n/4)+n\lg n$ entonces $a=3$, $b=4$, $f(n) \in \Omega(n^{\log_4 3 + 0,2})$ y $3(n/4) \lg(n/4) \leq 3/4n \lg n$, por lo que se aplica el caso 3 y $T(n) \in \Theta(n \lg n)$

4.6.4 Método de la Ecuación Característica

Se aplica a ciertas recurrencias lineales con coeficientes constantes como:

$$T(n) = \begin{cases} 5 & \text{si } n=0 \\ 10 & \text{si } n=1 \\ 5T(n-1) + 8T(n-2)+2n & \text{si } n > 0 \end{cases}$$

En general, para recurrencias de la forma:

$$T(n) = a_1T(n-1) + a_2T(n-2) + \ldots + a_kT(n-k) + b^n p(n)$$

donde a_i, $1 \leq i \leq k$, b son constantes y $p(n)$ es un polinomio en n de grado s.

Ejemplos:

1. En $T(n)=2t(n-1)+3^n$, $a_1=2$, $b=3$, $p(n)=1$, $s=0$.
2. En $T(n)=t(n-1)+ t(n-2)+n$, $a_1=1$, $a_2=1$, $b=1$, $p(n)=n$, $s=1$.

En general, para:

$$T(n) = a_1T(n-1) + a_2T(n-2) + \ldots + a_kT(n-k) + b^n p(n)$$

- Paso 1: Encontrar las raíces no nulas de la ecuación característica:

  $(x^k - a_1x^{k-1} - a_2x^{k-2} - \ldots - a_k)(x-b)^{s+1} = 0$

  Raíces: r_i, $1 \le i \le l \le k$, cada una con multiplicidad m_i.

- Paso 2: Las soluciones son de la forma de combinaciones lineales de estas raíces de acuerdo a su multiplicidad.

  $T(n)$ = -----

- Paso 3: Se encuentran valores para las constantes c_{ij}, tal que:

  $1 \le i \le l, 0 \le j \le m_i - 1$

  y d_i, $0 \le i \le s - 1$ según la recurrencia original y las condiciones iniciales (valores de la recurrencia para n=0,1, …).

Ejemplo: Resolver la ecuación de recurrencia siguiente:

$$T(n) = \begin{cases} 0 & \text{si } n=0 \\ 2T(n-1) + 1 & \text{si } n > 0 \end{cases}$$

donde b=1 y $p(n)$=1 de grado 0.

La **ecuación característica** $(x-2)(x-1)^{0+1} = 0$, con raíces 2 y 1 de multiplicidad 1.

La **solución general** es entonces de la forma: $T(n)=c_{11}2^n + c_{21}1^n$.

A partir de las condiciones iniciales se encuentra el siguiente sistema de ecuaciones que sirve para hallar c_{11} y c_{21}:

$c_{11} + c_{21} = 0$ de $n = 0$

$2c_{11} + c_{21} = 1$ de $n = 1$

de donde $c_{11} = 1$ y $c_{21} = -1$.

La **solución** es entonces: $T(n) = 2^n - 1$.

4.6.5 Cambio de Variable

Dada la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} a & \text{si } n=1 \\ 2T(n/2) + n\log_2 n & \text{si } n = 2^k \end{cases}$$

No se puede aplicar el teorema maestro ni la ecuación característica.

Se define una nueva recurrencia $S(i) = T(2^i)$, con el objetivo de llevarla a una forma en la que se pueda resolver siguiendo algún método anterior.

Entonces el caso general queda:

$$S(i) = T(2^i) = 2T(2^i/2) + 2^i i = 2T(2^{i-1}) + i2^i = 2S(i\text{-}1) + i2^i$$

Con $b=2$ y $p(i) = i$ de grado 1.

La ecuación característica de esta recurrencia es:

$$(x - 2)(x - 2)^{i+1} = 0,$$

con raíz 2 de grado 3.

La solución es entonces $S(i) = c_{11}2^i + c_{12}i2^i + c_{13}i^2 2^i$, con lo que volviendo a la variable original queda:

$$T(n) - 2T(n/2) = n\log_2 n = (c_{12} - c_{13})n + 2c_{13}n(\log_2 n).$$

Se pueden obtener los valores de las constantes sustituyendo esta solución en la recurrencia original:

$$T(n) = c_{11}n + c_{12}(\log_2 n)n + c_{13}(\log_2 n)^2 n.$$

de donde $c_{12} = c_{13}$ y $2c_{12} = 1$.

Por tanto $T(n) \in \Theta(n\log^2 n \mid n$ **es potencia de** $2)$.

Si se puede probar que $T(n)$ es eventualmente no decreciente, por la **regla de las funciones de crecimiento suave** se puede extender el resultado a todos los n (dado que $n\log^2 n$ es de crecimiento suave).

En este caso $T(n) \in \Theta(n\log^2 n)$.

4.7 Ejemplos y Ejercicios

Ejemplos de cálculo del tiempo de ejecución de un programa

Veamos el análisis de un simple enunciado de asignación a una variable entera:

```
a = b;
```

Como el enunciado de asignación toma tiempo constante, está en $\Theta(1)$.

Consideremos un simple ciclo "*for*":

```
sum=0;
for (i=1; i<=n; i++)
    sum += n;
```

La primera línea es $\Theta(1)$. El ciclo "*for*" es repetido n veces. La tercera línea toma un tiempo constante también, el costo total por ejecutar las dos líneas que forman el ciclo "*for*" es $\Theta(n)$. El costo por el entero fragmento de código es también $\Theta(n)$.

Analicemos un fragmento de código con varios ciclos "*for*", algunos de los cuales están anidados.

```
sum=0;
for (j=1; j<=n; j++)
        for (i=1; i<=j; i++)
            sum++;
for (k=1; k<=n; k++)
        A[k]= k-1;
```

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

La asignación toma tiempo constante, llamémosla c_1. El segundo ciclo es similar al ejemplo anterior y toma $c_2n = \Theta(n)$.

Analicemos ahora el primer ciclo, que es un doble ciclo anidado. En este caso trabajemos de adentro hacia fuera. La expresión *sum++* requiere tiempo constante, llamemosle c_3.

Como el ciclo interno es ejecutado j veces, tiene un costo de c_3j. El ciclo exterior es ejecutado n veces, pero cada vez el costo del ciclo interior es diferente.

El costo total del ciclo es c_3 veces la suma de los números 1 a n, es decir $\Sigma^{n}_{i=1} j = n(n+1)/2$, que es $\Theta(n^2)$. Por tanto, $\Theta(c_1+c_2n+c_3n^2)$ es simplemente $\Theta(n^2)$.

Comparemos el análisis asintótico de los siguientes fragmentos de código:

```
sum1=0;
for(i=1; i<=n; i++)
   for(j=1; j<=n; j++)
      sum1++;

sum2=0;
for(i=1; i<=n; i++)
   for(j=1; j<=i; j++)
      sum2++;
```

El primer fragmento de ejecuta el enunciado *sum1*++, precisamente n^2 veces.

Por otra parte, el segundo fragmento de código tiene un costo aproximado de ½ n^2. Así ambos códigos tienen un costo de $\Theta(n^2)$.

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

```
sum1=0;
for(k=1; k<=n; k*=2)
  for(j=1; j<=n; j++)
    sum1++;
```

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

Capítulo 5

Estrategias de Diseño de Algoritmos

5.1 Introducción

A través de los años, los científicos de la computación han identificado diversas técnicas generales que a menudo producen algoritmos eficientes para la resolución de muchas clases de problemas. Este capítulo presenta algunas de las técnicas más importantes como son: recursión, dividir para conquistar, técnicas ávidas, el método de retroceso y programación dinámica.

Se debe, sin embargo, destacar que hay algunos problemas, como los NP completos, para los cuales ni éstas ni otras técnicas conocidas producirán soluciones eficientes. Cuando se encuentra algún problema de este tipo, suele ser útil determinar si las entradas al problema tienen características especiales que se puedan explotar en la búsqueda de una solución, o si puede usarse alguna solución aproximada sencilla, en vez de la solución exacta, difícil de calcular.

5.2 Recursión

La recursividad es una técnica fundamental en el diseño de algoritmos eficientes, que está basada en la solución de versiones más pequeñas del problema, para obtener la solución general del mismo. Una **instancia** del problema se soluciona según la solución de una o más instancias **diferentes** y **más pequeñas** que ella.

Es una herramienta poderosa que sirve para resolver cierto tipo de problemas reduciendo la complejidad y ocultando los detalles del problema. Esta herramienta consiste en que una función o procedimiento se llama a sí mismo.

Una gran cantidad de algoritmos pueden ser descritos con mayor claridad en términos de recursividad, típicamente el resultado será que sus programas serán más pequeños.

La recursividad es una alternativa a la iteración o repetición, y aunque en tiempo de computadora y en ocupación en memoria es la solución recursiva menos eficiente que la solución iterativa, existen numerosas situaciones en las que la recursividad es una solución simple y natural a un problema que en caso contrario ser difícil de resolver. Por esta razón se puede decir que la

recursividad es una herramienta potente y útil en la resolución de problemas que tengan naturaleza recursiva y, por ende, en la programación.

Existen numerosas definiciones de **recursividad**, algunas de las más importantes o sencillas son éstas:

- Un objeto es *recursivo* si figura en su propia definición.

- Una definición *recursiva* es aquella en la que el objeto que se define forma parte de la definición (recuerde la regla gramatical: lo definido nunca debe formar parte de la definición)

La característica importante de la recursividad es que siempre existe un medio de salir de la definición, mediante la cual se termina el proceso recursivo.

Ventajas:

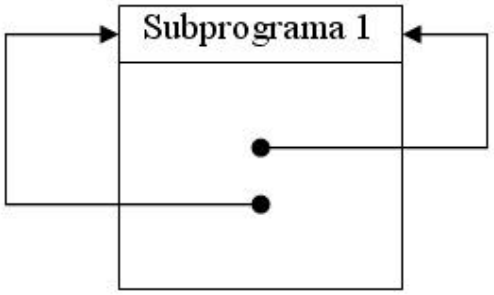
- Puede resolver problemas complejos.
- Solución más natural.

Desventajas:

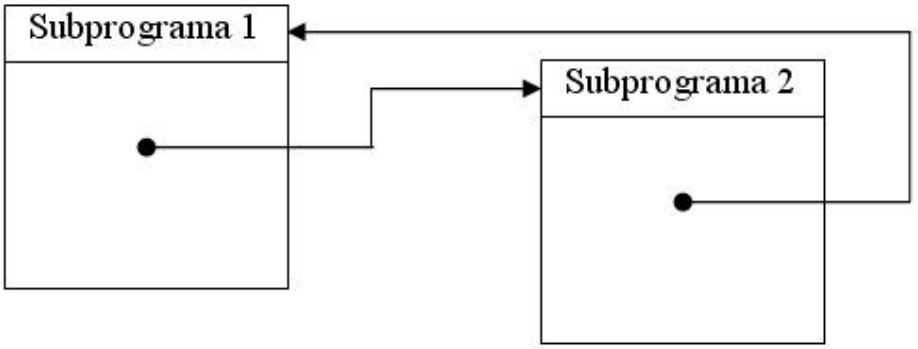
- Se puede llegar a un ciclo infinito.
- Versión no recursiva más difícil de desarrollar.
- Para la gente sin experiencia es difícil de programar.

Tipos de Recursividad

- **Directa o simple:** un subprograma se llama a si mismo una o más veces directamente.



- **Indirecta o mutua:** un subprograma A llama a otro subprograma B y éste a su vez llama al subprograma A.



a) Propiedades para un Procedimiento Recursivo

- Debe de existir cierto criterio (criterio base) en donde el procedimiento no se llama a sí mismo. Debe existir al menos una solución no recursiva.

- En cada llamada se debe de acercar al criterio base (reducir rango). El problema debe reducirse en tamaño, expresando el nuevo problema en términos del propio problema, pero más pequeño.

Los algoritmos de *divide y vencerás* pueden ser procedimientos recursivos.

b) Propiedades para una Función Recursiva

- Debe de haber ciertos argumentos, llamados valores base para los que la función no se refiera a sí misma.

- Cada vez que la función se refiera así misma el argumento de la función debe de acercarse más al valor base.

Ejemplos:

Factorial

$n! = 1 * 2 * 3 \ldots * (n-2) * (n-1) * n$
$0! = 1$
$1! = 1$
$2! = 1 * 2$
$3! = 1 * 2 * 3$
...
$n! = 1 * 2 * 3 \ldots * (n-2) * (n-1) * n$

$\Rightarrow n! = n * (n-1)!$

si $n = 0$ entonces $n! = 1$
si $n > 0$ entonces $n! = n * (n-1)!$

```
Function factorial(n:integer):longint:
begin
   if ( n = 0 ) then
          factorial:=1
   else
          factorial:= n * factorial(n-1);
end;
```

Rastreo de ejecución:

Si $n = 6$

Nivel

1) Factorial(6)= 6 * factorial(5) = 720
2) Factorial(5)=5 * factorial(4) = 120
3) Factorial(4)=4 * factorial(3) = 24
4) Factorial(3)=3 * factorial(2) = 6
5) Factorial(2)=2 * factorial(1) = 2
6) Factorial(1)=1 * factorial(0) = 1
7) Factorial(0)=1

La *profundidad* de un procedimiento recursivo es el número de veces que se llama a sí mismo.

Serie de Fibonacci

0, 1, 1, 2, 3, 5, 8, ...

$F0 = 0$
$F1 = 1$
$F2 = F1 + F0 = 1$
$F3 = F2 + F1 = 2$
$F4 = F3 + F2 = 3$
...
$Fn = Fn-1 + Fn-2$

si $n = 0, 1$ $Fn = n$

si $n > 1$ $Fn = Fn-1 + Fn-2$

```
Procedure fibo(var fib:longint; n:integer);
var
        fibA, fibB : integer;
begin
        if ( n = 0) or ( n = 1) then fib:=n
        else begin
                fibo(fibA,n-1);
                fibo(fibB,n-2);
                fib:=fibA + FibB;
        end;
end;


function fibo ( n:integer) : integer;
begin
        if ( n = 0) or ( n = 1) then
                fibo:=n
         else
                fibo:=fibo(n-1) + fibo(n-2);
       end;
end;
```

No todos los ambientes de programación proveen las facilidades necesarias para la recursión y adicionalmente, a veces su uso es una fuente de ineficiencia inaceptable. Por lo cual se prefiere eliminar la recursión. Para ello, se sustituye la llamada recursiva por un lazo, donde se incluyen algunas asignaciones para recolocar los parámetros dirigidos a la función.

Remover la recursión es complicado cuando existe más de una llamada recursiva, pero una vez hecha ésta, la versión del algoritmo es **siempre** más eficiente.

5.3 Dividir para Conquistar

Muchos algoritmos útiles tienen una estructura *recursiva*, de modo que para resolver un problema se llaman recursivamente a sí mismos una o más veces para solucionar subproblemas muy similares.

Esta estructura obedece a una estrategia *dividir-y-conquistar*, en que se ejecuta tres pasos en cada nivel de la recursión:

- **Dividir**. Dividen el problema en varios subproblemas similares al problema original pero de menor tamaño;

- **Conquistar**. Resuelven recursivamente los subproblemas si los tamaños de los subproblemas son suficientemente pequeños, entonces resuelven los subproblemas de manera directa; y luego,

- **Combinar**. Combinan estas soluciones para crear una solución al problema original.

La técnica *Dividir para Conquistar* (o Divide y Vencerás) consiste en descomponer el caso que hay que resolver en subcasos más pequeños, resolver independientemente los subcasos y por último combinar las soluciones de los subcasos para obtener la solución del caso original.

Caso General

Consideremos un problema arbitrario, y sea *A* un algoritmo sencillo capaz de resolver el problema. *A* debe ser eficiente para casos pequeños y lo denominamos *subalgoritmo básico*.

El caso general de los algoritmos de Divide y Vencerás (DV) es como sigue:

```
función DV(x)
  {
    si x es suficientemente pequeño o sencillo entonces
          devolver A(x)
    descomponer x en casos más pequeños x1, x2, x3 , ... , xl
    para i ← 1 hasta l hacer
          yi ← DV(xi)
    recombinar los yi para obtener una solución y de x
    devolver y
  }
```

El número de subejemplares l suele ser pequeño e independiente del caso particular que haya que resolverse.

Para aplicar la estrategia Divide y Vencerás es necesario que se cumplan tres condiciones:

- La decisión de utilizar el subalgoritmo básico en lugar de hacer llamadas recursivas debe tomarse cuidadosamente.

- Tiene que ser posible descomponer el caso en subcasos y recomponer las soluciones parciales de forma eficiente.

- Los subcasos deben ser en lo posible aproximadamente del mismo tamaño.

En la mayoría de los algoritmos de Divide y Vencerás el tamaño de los l subcasos es aproximadamente m/b, para alguna constante b, en donde m es el tamaño del caso (o subcaso) original (cada subproblema es aproximadamente del tamaño $1/b$ del problema original).

El análisis de tiempos de ejecución para estos algoritmos es el siguiente:

Sea $g(n)$ el tiempo requerido por DV en casos de tamaño n, sin contar el tiempo necesario para llamadas recursivas. El tiempo total $t(n)$ requerido por este algoritmo de Divide y Vencerás es parecido a:

$$t(n) = l\,t(n/b) + g(n) \quad \text{para } l = 1 \text{ y } b = 2$$

siempre que n sea suficientemente grande. Si existe un entero $k = 0$ tal que:

$$g(n) \in \Theta(n^k),$$

se puede concluir que:

$$t(n) \in \begin{cases} T(n^k) & \text{si } l < b^k \\ T(n^k \log n) & \text{si } l = b^k \\ T(n^{\log_b l}) & \text{si } l > b^k \end{cases}$$

Se deja propuesta la demostración de esta ecuación de recurrencia usando análisis asintótico.

5.4 Programación Dinámica

Frecuentemente para resolver un problema complejo se tiende a dividir este en subproblemas más pequeños, resolver estos últimos (recurriendo posiblemente a nuevas subdivisiones) y combinar las soluciones obtenidas para calcular la solución del problema inicial.

Puede ocurrir que la división natural del problema conduzca a un gran número de subejemplares idénticos. Si se resuelve cada uno de ellos sin tener en cuenta las posibles repeticiones, resulta un algoritmo ineficiente; en cambio si se resuelve cada ejemplar distinto una sola vez y se conserva el resultado, el algoritmo obtenido es mucho mejor.

Esta es la idea de la programación dinámica: no calcular dos veces lo mismo y utilizar normalmente una tabla de resultados que se va rellenando a medida que se resuelven los subejemplares.

La programación dinámica es un método *ascendente*. Se resuelven primero los subejemplares más pequeños y por tanto más simples. Combinando las soluciones se obtienen las soluciones de ejemplares sucesivamente más grandes hasta llegar al ejemplar original.

Ejemplo:

Consideremos el cálculo de números combinatorios. El algoritmo sería:

```
función C(n, k)
  si k=0 o k=n entonces devolver 1
  sino devolver C(n-1, k-1) + C(n-1, k)
```

Ocurre que muchos valores $C(i, j)$, con $i < n$ y $j < k$ se calculan y recalculan varias veces.

Un fenómeno similar ocurre con el algoritmo de *Fibonacci*.

La programación dinámica se emplea a menudo para resolver problemas de optimización que satisfacen el *principio de optimalidad*: en una secuencia óptima de decisiones toda subsecuencia ha de ser también óptima.

Ejemplo:

Si el camino más corto de Santiago a Copiapó pasa por La Serena, la parte del camino de Santiago a La Serena ha de ser necesariamente el camino mínimo entre estas dos ciudades.

Podemos aplicar esta técnica en:

- Camino mínimo en un grafo orientado
- Árboles de búsqueda óptimos

5.5 Algoritmos Ávidos

Los algoritmos ávidos o voraces (*Greedy Algorithms*) son algoritmos que toman decisiones de corto alcance, basadas en información inmediatamente disponible, sin importar consecuencias futuras.

Suelen ser bastante simples y se emplean sobre todo para resolver problemas de optimización, como por ejemplo, encontrar la secuencia óptima para procesar un conjunto de tareas por un computador, hallar el camino mínimo de un grafo, etc.

Habitualmente, los elementos que intervienen son:

- un conjunto o lista de ***candidatos*** (tareas a procesar, vértices del grafo, etc.);
- un conjunto de ***decisiones*** ya tomadas (candidatos ya escogidos);
- una ***función*** que determina si un conjunto de candidatos es una ***solución*** al problema (aunque no tiene por qué ser la óptima);

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

- una ***función*** de selección que escoge el candidato aún no seleccionado que es más ***prometedor***;

- una ***función objetivo*** que da el valor/coste de una solución (tiempo total del proceso, la longitud del camino, etc.) y que es la que se pretende maximizar o minimizar;

Para resolver el problema de optimización hay que encontrar un conjunto de candidatos que optimiza la función objetivo. Los algoritmos voraces proceden por pasos.

Inicialmente el conjunto de candidatos es vacío. A continuación, en cada paso, se intenta añadir al conjunto el mejor candidato de los aún no escogidos, utilizando la función de selección. Si el conjunto resultante no es completable, se rechaza el candidato y no se le vuelve a considerar en el futuro. En caso contrario, se incorpora al conjunto de candidatos escogidos y permanece siempre en él. Tras cada incorporación se comprueba si el conjunto resultante es una solución del problema. Un algoritmo voraz es correcto si la solución así encontrada es siempre óptima.

El esquema genérico del algoritmo voraz es:

```
function voraz(C:conjunto):conjunto
    { C es el conjunto de todos los candidatos }
    S <= vacío    { S es el conjunto en el que se construye la solución
    mientras ¬solución(S) y C<> vacío hacer
      x ← el elemento de C que maximiza seleccionar(x)
      C ← C \ {x}
      si completable(S U {x}) entonces S ← S U {x}
si solución(S)
 entonces devolver S
si no devolver no hay solución
```

El nombre voraz proviene de que, en cada paso, el algoritmo escoge el mejor "pedazo" que es capaz de "comer" sin preocuparse del futuro. Nunca deshace una decisión ya tomada: una vez incorporado un candidato a la solución permanece ahí hasta el final; y cada vez que un candidato es rechazado, lo es para siempre.

Ejemplo: Mínimo número de monedas

Se desea pagar una cantidad de dinero a un cliente empleando el menor número posible de monedas. Los elementos del esquema anterior se convierten en:

- *candidato*: conjunto finito de monedas de, por ejemplo, 1, 5, 10 y 25 unidades, con una moneda de cada tipo por lo menos;
- *solución*: conjunto de monedas cuya suma es la cantidad a pagar;
- *completable*: la suma de las monedas escogidas en un momento dado no supera la cantidad a pagar;
- *función de selección*: la moneda de mayor valor en el conjunto de candidatos aún no considerados;
- *función objetivo*: número de monedas utilizadas en la solución.

5.6 Método de Retroceso (*backtracking*)

El *Backtracking* o ***vuelta atrás*** es una técnica de resolución general de problemas mediante una búsqueda sistemática de soluciones. El procedimiento general se basa en la descomposición del proceso de búsqueda en tareas parciales de tanteo de soluciones (*trial and error*).

Las tareas parciales se plantean de forma recursiva al construir gradualmente las soluciones. Para resolver cada tarea, se descompone en varias subtareas y se comprueba si alguna de ellas conduce a la solución del problema. Las subtareas se prueban una a una, y si una subtarea no conduce a la solución se prueba con la siguiente.

La descripción natural del proceso se representa por un árbol de búsqueda en el que se muestra como cada tarea se ramifica en subtareas. El árbol de búsqueda suele crecer rápidamente por lo que se buscan herramientas eficientes para descartar algunas ramas de búsqueda produciéndose la poda del árbol.

Diversas estrategias ***heurísticas***, frecuentemente dependientes del problema, establecen las reglas para decidir en que orden se tratan las tareas mediante las que se realiza la ramificación del árbol de búsqueda, y qué funciones se utilizan para provocar la poda del árbol.

El método de *backtracking* se ajusta a muchos famosos problemas de la inteligencia artificial que admiten extensiones muy sencillas. Algunos de estos problemas son el de las Torres de *Hanoi*, el del salto del caballo o el de la colocación de las ocho reinas en el tablero de ajedrez, el problema del laberinto.

Un caso especialmente importante en optimización combinatoria es el problema de la **selección óptima**. Se trata del problema de seleccionar una solución por sus componentes, de forma que respetando una serie de restricciones, de lugar a la solución que optimiza una función objetivo que se evalúa al construirla. El árbol de búsqueda permite recorrer todas las soluciones para quedarse con la mejor de entre las que sean factibles. Las estrategias heurísticas orientan la exploración por las ramas más prometedoras y la poda en aquellas que no permiten alcanzar una solución factible o mejorar la mejor solución factible alcanzada hasta el momento.

El problema típico es el conocido como **problema de la mochila**. Se dispone de una mochila en la que se soporta un peso máximo P en la que se desean introducir objetos que le den el máximo valor. Se dispone de una colección de n objetos de los que se conoce su peso y su valor;

$$(pi, v\,i);\ i = 1, 2, ..., n.$$

5.7 Método de *Branch and Bound*

Branch and bound (o también conocido como **Ramifica y Poda**) es una técnica muy similar a la de *Backtracking*, pues basa su diseño en el análisis del árbol de búsqueda de soluciones a un problema. Sin embargo, no utiliza la búsqueda en profundidad (*depth first*), y solamente se aplica en problemas de **Optimización**.

Los algoritmos generados por está técnica son normalmente de orden exponencial o peor en su peor caso, pero su aplicación ante instancias muy grandes, ha demostrado ser eficiente (incluso más que *bactracking*).

Se debe decidir de qué manera se conforma el árbol de búsqueda de soluciones. Sobre el árbol, se aplicará una búsqueda en anchura (*breadth-first*), pero considerando **prioridades** en los nodos que se visitan (*best-first*). El criterio de selección de nodos, se basa en un valor óptimo posible (*bound*), con el que se toman decisiones para hacer las **podas** en el árbol.

Modo de proceder de los algoritmos *Branch and Bound*:

- Nodo vivo: nodo con posibilidad de ser ramificado, es decir, no se ha realizado una poda.
- No se pueden comenzar las podas hasta que no se haya descendido a una hoja del árbol
- Se aplicará la poda a los nodos con una cota peor al valor de una solución dada → a los nodos podados a los que se les ha aplicado una poda se les conoce como nodos muertos
- Se ramificará hasta que no queden nodos vivos por expandir
- Los nodos vivos han de estar almacenados en algún sitio → lista de nodos, de modo que sepamos qué nodo expandir

Algoritmo General:

```
Función ramipoda(P:problema): solucion
L={P} (lista de nodos vivos)
S= ∞± (según se aplique máximo o mínimo)
Mientras L ≠ 0
      n=nodo de mejor cota en L
      si solucion(n)
            si (valor(n) mejor que s)
                  s=valor(n)
                  para todo  m ∈ L y Cota(m) peor que S
                        borrar nodo m de L
      sino
            añadir a L los hijos de n con sus cotas
      borrar n de L (tanto si es solución como si ha ramificado)
Fin_mientras
Devolver S
```

Capítulo 6

Algoritmos de Ordenamiento

6.1 Concepto de Ordenamiento

DEFINICIÓN.

Entrada: Una secuencia de n números $<a_1, a_2, .., a_n>$, usualmente en la forma de un arreglo de n elementos.

Salida: Una permutación $<a'_1, a'_2, .., a'_n>$ de la secuencia de entrada tal que $a'_1 = a'_2 = ...= a'_n$.

CONSIDERACIONES. Cada número es normalmente una *clave* que forma parte de un *registro*; cada registro contiene además *datos satélites* que se mueven junto con la clave; si cada registro incluye muchos datos satélites, entonces movemos punteros a los registros (y no los registros).

Un método de ordenación se denomina **estable** si el orden relativo de los elementos no se altera por el proceso de ordenamiento. (Importante cuanto se ordena por varias claves).

Hay dos categorías importantes y disjuntas de algoritmos de ordenación:

- ***Ordenación interna:*** La cantidad de registros es suficientemente pequeña y el proceso puede llevarse a cabo en memoria.
- ***Ordenación externa:*** Hay demasiados registros para permitir ordenación interna; deben almacenarse en disco.

Por ahora nos ocuparemos sólo de ordenación interna.

Los algoritmos de ordenación interna se clasifican de acuerdo con la cantidad de trabajo necesaria para ordenar una secuencia de n elementos:

¿Cuántas **comparaciones** de elementos y cuántos **movimientos** de elementos de un lugar a otro son necesarios?

Empezaremos por los métodos tradicionales (inserción y selección). Son fáciles de entender, pero ineficientes, especialmente para juegos de datos grandes. Luego prestaremos más atención a los más eficientes: *heapsort* y *quicksort*.

6.2 Ordenamiento por Inserción

Este es uno de los métodos más sencillos. Consta de tomar uno por uno los elementos de un arreglo y recorrerlo hacia su posición con respecto a los anteriormente ordenados. Así empieza con el segundo elemento y lo ordena con respecto al primero. Luego sigue con el tercero y lo coloca en su posición ordenada con respecto a los dos anteriores, así sucesivamente hasta recorrer todas las posiciones del arreglo.

Este es el algoritmo:

Sea n el número de elementos en el arreglo A.

INSERTION-SORT(A) :

{Para cada valor de j, inserta $A[j]$ en la posición que le corresponde en la secuencia ordenada $A[1 .. j-1]$}

```
for j ← 2 to n do
        k ← A[j]
        i ← j – 1
        while i > 0 ∧ A[i] > k do
                A[i + 1] ← A[i]
                i ← i – 1
        A[i + 1] ← k
```

Análisis del algoritmo:

Número de comparaciones: $n(n\text{-}1)/2$ lo que implica un $T(n) = O(n^2)$. La ordenación por inserción utiliza aproximadamente $n^2/4$ comparaciones y $n^2/8$ intercambios en el caso medio y dos veces más en el peor de los casos. La ordenación por inserción es lineal para los archivos casi ordenados.

6.3 Ordenamiento por Selección

El método de ordenamiento por selección consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenar todo el arreglo.

SELECTION-SORT(A) :

{Para cada valor de j, selecciona el menor elemento de la secuencia no ordenada $A[j .. n]$ y lo intercambia con $A[j]$}

```
for j ← 1 to n – 1 do
        k ← j
        for i ← j + 1 to n do
                if A[i] < A[k] then k ← i
        intercambie A[j] ↔ A[k]
```

Análisis del algoritmo:

La ordenación por selección utiliza aproximadamente $n^2/2$ comparaciones y n intercambios, por lo cual $T(n) = O(n^2)$.

6.4 Ordenamiento de la Burbuja (*Bubblesort*)

El algoritmo *bubblesort*, también conocido como ordenamiento burbuja, funciona de la siguiente manera: Se recorre el arreglo intercambiando los elementos adyacentes que estén desordenados. Se recorre el arreglo tantas veces hasta que ya no haya cambios que realizar. Prácticamente lo que hace es tomar el elemento mayor y lo va recorriendo de posición en posición hasta ponerlo en su lugar.

Procedimiento ***BubbleSort***

```
for (i = n; i >= 1; i--)
   for (j = 2; j <= i; j++)
   { if (A[j-1] > A[j]) then  intercambie(A, j-1, j);
   }
```

Análisis del algoritmo:

La ordenación de burbuja tanto en el caso medio como en el peor de los casos utiliza aproximadamente $n^2/2$ comparaciones y $n^2/2$ intercambios, por lo cual $T(n) = O(n^2)$.

6.5 Ordenamiento Rápido (*Quicksort*)

Vimos que en un algoritmo de ordenamiento por intercambio, son necesarios intercambios de elementos en sublistas hasta que no son posibles más. En el burbujeo, son comparados ítems correlativos en cada paso de la lista. Por lo tanto para ubicar un ítem en su correcta posición, pueden ser necesarios varios intercambios.

Veremos el sort de intercambio desarrollado por *C.A.R. Hoare* conocido como ***Quicksort***. Es más eficiente que el burbujeo (*bubblesort*) porque los intercambios involucran elementos que están más apartados, entonces menos intercambios son requeridos para poner un elemento en su posición correcta.

La idea básica del algoritmo es elegir un elemento llamado **pivote**, y ejecutar una secuencia de intercambios tal que todos los elementos menores que el pivote queden a la izquierda y todos los mayores a la derecha.

Lo único que requiere este proceso es que todos los elementos a la izquierda sean menores que el pivote y que todos los de la derecha sean mayores luego de cada paso, no importando el ***orden*** entre ellos, siendo precisamente esta flexibilidad la que hace eficiente al proceso.

Hacemos dos búsquedas, una desde la izquierda y otra desde la derecha, comparando el pivote con los elementos que vamos recorriendo, buscando los menores o iguales y los mayores respectivamente.

DESCRIPCIÓN. Basado en el paradigma dividir-y-conquistar, estos son los tres pasos para ordenar un subarreglo $A[p \ldots r]$:

Dividir. El arreglo $A[p \ldots r]$ es particionado en dos subarreglos no vacíos $A[p \ldots q]$ y $A[q+1 \ldots r]$—cada dato de $A[p \ldots q]$ es menor que cada dato de $A[q+1 \ldots r]$; el índice q se calcula como parte de este proceso de partición.

Conquistar. Los subarreglos $A[p \ldots q]$ y $A[q+1 \ldots r]$ son ordenados mediante sendas llamadas recursivas a QUICKSORT.

Combinar. Ya que los subarreglos son ordenados *in situ*, no es necesario hacer ningún trabajo extra para combinarlos; todo el arreglo $A[p \ldots r]$ está ahora ordenado.

```
QUICKSORT (A, p, r) :
  if p < r then
    q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q)
    QUICKSORT(A, q+1, r)
```

```
PARTITION (A, p, r) :
  x ← A[p]; i ← p–1; j ← r+1
  while τ
    repeat j ← j–1 until A[j] = x
    repeat i ← i+1 until A[i] = x
    if i < j then
      intercambie A[i] ↔ A[j]
    else return j
```

Para ordenar un arreglo completo A, la llamada inicial es QUICKSORT$(A, 1, n)$.

PARTITION reordena el subarreglo $A[p \ldots r]$ *in situ*, poniendo datos menores que el ***pivote*** $x = A[p]$ en la parte baja del arreglo y datos mayores que x en la parte alta.

Análisis del algoritmo:

Depende de si la partición es o no balanceada, lo que a su vez depende de cómo se elige los pivotes:

- Si la partición es balanceada, QUICKSORT corre asintóticamente tan rápido como ordenación por mezcla.

- Si la partición es desbalanceada, QUICKSORT corre asintóticamente tan lento como la ordenación por inserción.

El peor caso ocurre cuando PARTITION produce una región con $n-1$ elementos y otra con sólo un elemento.

Si este desbalance se presenta en cada paso del algoritmo, entonces, como la partición toma tiempo $\Theta(n)$ y $\sigma(1) = \Theta(1)$, tenemos la recurrencia:

$$\sigma(n) = \sigma(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(n^2).$$

Por ejemplo, este tiempo $\Theta(n^2)$ de ejecución ocurre cuando el arreglo está completamente ordenado, situación en la cual INSERTIONSORT corre en tiempo $O(n)$.

Si la partición produce dos regiones de tamaño $n/2$, entonces la recurrencia es:

$$\sigma(n) = 2\sigma(n/2) + \Theta(n) = \Theta(n \log n).$$

Tenemos un algoritmo mucho más rápido cuando se da el mejor caso de PARTITION.

El caso promedio de QUICKSORT es mucho más parecido al mejor caso que al peor caso; por ejemplo, si PARTITION siempre produce una división de proporcionalidad 9 a 1:

- Tenemos la recurrencia $\sigma(n) = \sigma(9n/10) + \sigma(n/10) + n$.
- En el árbol de recursión, cada nivel tiene costo n (o a lo más n más allá de la profundidad $\log_{10} n$), y la recursión termina en la profundidad $\log_{10/9} n = \Theta(\log n)$ …⇒
- …el costo total para divisiones 9 a 1 es $\Theta(n \log n)$, asintóticamente lo mismo que para divisiones justo en la mitad.
- Cualquier división de proporcionalidad constante, por ejemplo, 99 a 1, produce un árbol de recursión de profundidad $\Theta(\log n)$, en que el costo de cada nivel es $\Theta(n)$, lo que se traduce en que el tiempo de ejecución es $\Theta(n \log n)$.

En la práctica, en promedio, PARTITION produce una mezcla de divisiones buenas y malas, que en el árbol de recursión están distribuidas aleatoriamente; si suponemos que aparecen alternadamente por niveles:

- En la raíz, el costo de la partición es n y los arreglos producidos tienen tamaños $n-1$ y 1.

- En el siguiente nivel, el arreglo de tamaño $n-1$ es particionado en dos arreglos de tamaño $(n-1)/2$, y el costo de procesar un arreglo de tamaño 1 es 1.
- Esta combinación produce tres arreglos de tamaños 1, $(n-1)/2$ y $(n-1)/2$ a un costo combinado de $2n-1 = \Theta(n)$ …
- …no peor que el caso de una sola partición que produce dos arreglos de tamaños $(n-1)/2 + 1$ y $(n-1)/2$ a un costo de $n = \Theta(n)$, y que es muy aproximadamente balanceada.

Intuitivamente, el costo $\Theta(n)$ de la división mala es absorbido por el costo $\Theta(n)$ de la división buena:

- $\Rightarrow$ la división resultante es buena.

$\Rightarrow$ El tiempo de ejecución de QUICKSORT, cuando los niveles producen alternadamente divisiones buenas y malas, es como el tiempo de ejecución para divisiones buenas únicamente:

$\mathbf{O}(n \log n)$, pero con una constante más grande.

6.6 Ordenamiento por Montículo (*Heapsort*)

DEFINICIONES. Un *heap* (*binario*) es un arreglo que puede verse como un árbol binario completo:

- Cada nodo del árbol corresponde a un elemento del arreglo;
- El árbol está lleno en todos los niveles excepto posiblemente el de más abajo, el cual está lleno desde la izquierda hasta un cierto punto.

Un arreglo A que representa a un heap tiene dos atributos:

- $\lambda[A]$ es el número de elementos en el arreglo;
- $\sigma[A]$ es el número de elementos en el *heap* almacenado en el arreglo— $\sigma[A] = \lambda[A]$;
- $A[1 .. \lambda[A]]$ puede contener datos válidos, pero ningún dato más allá de $A[\sigma[A]]$ es un elemento del *heap*.

La raíz del árbol es $A[1]$; y dado el índice i de un nodo, los índices de su padre, hijo izquierdo, e hijo derecho se calculan como:

$P(i)$: **return** $\lfloor i/2 \rfloor$ $L(i)$: **return** $2i$ $R(i)$: **return** $2i + 1$

Un *heap* satisface la *propiedad de heap*:

- Para cualquier nodo i distinto de la raíz, $A[P(i)] = A[i]$;
- El elemento más grande en un heap está en la raíz;

- Los subárboles que tienen raíz en un cierto nodo contienen valores más pequeños que el valor contenido en tal nodo.

La altura de un *heap* de n elementos es $\Theta(\log n)$.

RESTAURACIÓN DE LA PROPIEDAD DE *HEAP*. Consideremos un arreglo A y un índice i en el arreglo, tal que:

- Los árboles binarios con raíces en $L(i)$ y $R(i)$ son heaps;
- $A[i]$ puede ser más pequeño que sus hijos, violando la propiedad de heap.

HEAPIFY hace "descender" por el *heap* el valor en $A[i]$, de modo que el subárbol con raíz en i se vuelva un *heap*; en cada paso:

- determinamos el más grande entre $A[i]$, $A[L(i)]$ y $A[R(i)]$, y almacenamos su índice en k;
- si el más grande no es $A[i]$, entonces intercambiamos $A[i]$ con $A[k]$, de modo que ahora el nodo i y sus hijos satisfacen la propiedad de *heap*;
- pero ahora, el nodo k tiene el valor originalmente en $A[i]$—el subárbol con raíz en k podría estar violando la propiedad de *heap* …
- …siendo necesario llamar a HEAPIFY recursivamente.

```
HEAPIFY(A, i) :
        l ← L(i);  r ← R(i)
        if l = σ[A] ∧ A[l] > A[i] then k ← l else k ← i
        if r = σ[A] ∧ A[r] > A[k] then k ← r
        if k ? i then
                intercambie A[i] ↔ A[k]
                HEAPIFY(A, k)
```

El tiempo de ejecución de HEAPIFY a partir de un nodo de altura h es $\boldsymbol{O}(h) = \boldsymbol{O}(\log n)$, en que n es el número de nodos en el subárbol con raíz en el nodo de partida.

CONSTRUCCIÓN DE UN HEAP. Usamos HEAPIFY de abajo hacia arriba para convertir un arreglo $A[1 .. n]$—$n = \lambda[A]$—en un *heap*:

- Como los elementos en $A[(\lfloor n/2 \rfloor + 1) .. n]$ son todos hojas del árbol, cada uno es un *heap* de un elemento;
- BUILD-HEAP pasa por los demás nodos del árbol y ejecuta HEAPIFY en cada uno.

- El orden en el cual los nodos son procesados garantiza que los subárboles con raíz en los hijos del nodo i ya son heaps cuando HEAPIFY es ejecutado en el nodo i.

BUILD-HEAP(A) :
$\sigma[A] \leftarrow \lambda[A]$
for $i \leftarrow \lfloor \lambda[A]/2 \rfloor$ ***downto*** 1 do HEAPIFY(A, i)

El tiempo que toma HEAPIFY al correr en un nodo varía con la altura del nodo, y las alturas de la mayoría de los nodos son pequeñas.

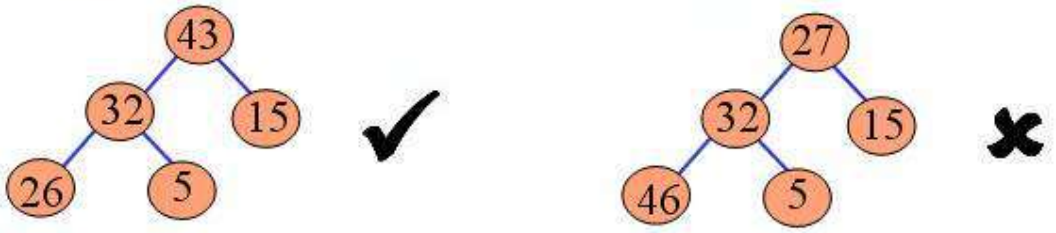
En un *heap* de n elementos hay a lo más $n/2^{h+1}$ nodos de altura h, de modo que el costo total de BUILD-HEAP es:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathbf{O}(h) = \mathbf{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = \mathbf{O}(n)$$

ya que $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$.

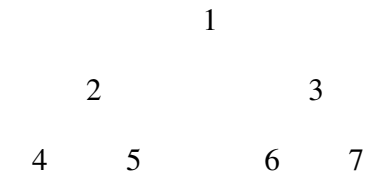
Un *heap* es un árbol binario con las siguientes características:

- Es completo. Cada nivel es completo excepto posiblemente el último, y en ese nivel los nodos están en las posiciones más a la izquierda.
- Los ítems de cada nodo son mayores e iguales que los ítems almacenados en los hijos.



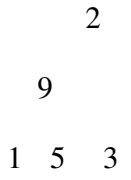
Para implementarlo podemos usar estructuras enlazadas o vectores numerando los nodos por nivel y de derecha a izquierda.

Es decir:



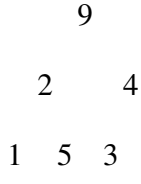
De este modo, es fácil buscar la dirección del padre o de los hijos de un nodo. El padre de *i* es: *i div* 2. Los hijos de *i* son: 2 * *i* y 2 * *i* + 1.

Un algoritmo para convertir un árbol binario completo en un *heap* es básico para otras operaciones de *heap*. Por ejemplo un árbol en el que ambos subárboles son *heap* pero el árbol en si no lo es.



La única razón por la que no es un *heap* es porque la raíz es menor que uno (o dos como en este caso subárboles)

El primer paso es intercambiar la raíz con el mayor de los dos hijos:



Esto garantiza que la nueva raíz sea mayor que ambos hijos, y que uno de estos siga siendo un *heap* (en este caso el derecho) y que el otro pueda ser o no. Si es, el árbol entero lo es, sino simplemente repetimos el *swapdown* en este subárbol.

Swapdown
1. Done := *false*
2. *c*:= 2 ** *r*
3. ***While not*** Done ***and*** *c* < = *n* ***Do***
   a. ***if*** *c* < *n* ***and*** *Heap*[*c*] < *Heap*[*c*+1] ***then*** *c* := *c*+1;
   b. ***if*** *Heap*[r] < *Heap*[*c*] ***then***
      i. Intercambiar *Heap*[*r*] y *Heap*[*c*]
      ii. *r* := *c*
      iii. *c* := 2 ** *c*
   else Done := *true*

Heapify

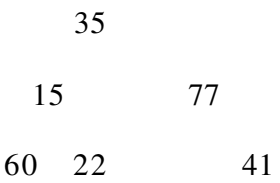
Recibe un Árbol Binario Completo (ABC) almacenado en un arreglo desde posición 1 a posición *n*. Convierte el árbol en un *heap*.

SwapDown (árbol cuya raíz esta en *r*)

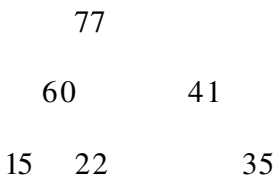
Veamos un ejemplo:

Sean los siguientes elementos: 35, 15, 77, 60, 22, 41

El arreglo contiene los ítems como un ABC.



Usamos *Heapify* para convertirlo en *Heap*.



Esto pone al elemento más grande en la raíz, es decir la posición 1 del vector. Ahora usamos la estrategia de *sort* de selección y posicionamos correctamente al elemento más grande y pasamos a ordenar la sublista compuesta por los otros 5 elementos

En términos del árbol estamos cambiando, la raíz por la hoja más a la derecha. Sacamos ese elemento del árbol. El árbol que queda no es un *heap*. Como solo cambiamos la raíz, los subárboles son *heaps*, entonces podemos usar *swapdown* en lugar del *Heapify* que consume más tiempo.

60

35 41

15 22

Ahora usamos la misma técnica de intercambiar la raíz con la hoja más a la derecha que podamos.

22

35 41

15 60 ----------→ 35,22,41,15,60,77

Volvemos a usar *SwapDown* para convertir en un *Heap*.

41

35 22

15

Y volvemos a hacer el intercambio hoja-raíz.

15

35 22

41 ----------→ 35,15,22,41,60,77

15

35 22

Volvemos a usar *SwapDown*

35

15 22 Intercambiamos hoja-raíz y podamos

22 ----------→ 22,15,35,41,60,77

Finalmente, el árbol de dos elementos es convertido en un *heap* e intercambiado la raíz con la hoja y podado

15

22 ----------→ 15,22,35,41,60,77

El siguiente algoritmo resume el procedimiento descrito.

Consideramos a *X* como un árbol binario completo (ABC) y usamos ***heapify*** para convertirlo en un *heap*.

For *i* = *n* ***down to*** 2 ***do***

- Intercambiar *X*[1] y *X*[*i*], poniendo el mayor elemento de la sublista *X*[1]...*X*[*i*] al final de esta.
- Aplicar *SwapDown* para convertir en *heap* el árbol binario correspondiente a la sublista almacenada en las posiciones 1 a *i* - 1 de *X*.

6.7 Otros Métodos de Ordenamiento

Existen varios otros algoritmos de ordenamiento. A continuación se mencionarán brevemente algunos otros más.

6.7.1 Ordenamiento por Incrementos Decrecientes

Denominado así por su desarrollador *Donald Shell* (1959). El ordenamiento por incrementos decrecientes (o método ***shellsort***) es una generalización del ordenamiento por inserción donde se gana rápidez al permitir intercambios entre elementos que se encuentran muy alejados.

La idea es reorganizar la secuencia de datos para que cumpla con la propiedad siguiente: si se toman todos los elementos separados a una distancia *h*, se obtiene una secuencia ordenada.

Se dice que la secuencia *h* ordenada está constituida por *h* secuencias ordenadas independendientes, pero entrelazadas entre sí. Se utiliza una serie decreciente *h* que termine en 1.

```
for ( h = 1; h <= n/ 9; h = 3*h + 1 );
  for ( ; h > 0; h/ = 3 )
    for ( i = h + 1; i <= n; i+= 1 )
    {  v = A[i];
       j = i;
```

```
    while ( j> h && A[j-h] > v ) then
    {  A[j] = A[j-1];
       j -= h;
    }
    A[j] = v;
  }
```

Análisis del algoritmo:

Esta sucesión de incrementos es fácil de utilizar y conduce a una ordenación eficaz. Hay otras muchas sucesiones que conducen a ordenaciones mejores. Es difícil mejorar el programa anterior en más de un 20%, incluso para ***n*** grande. Existen sucesiones desfavorables como por ejemplo: 64, 32, 16, 8, 4, 2, 1, donde sólo se comparan elementos en posiciones impares cuando $h=1$.

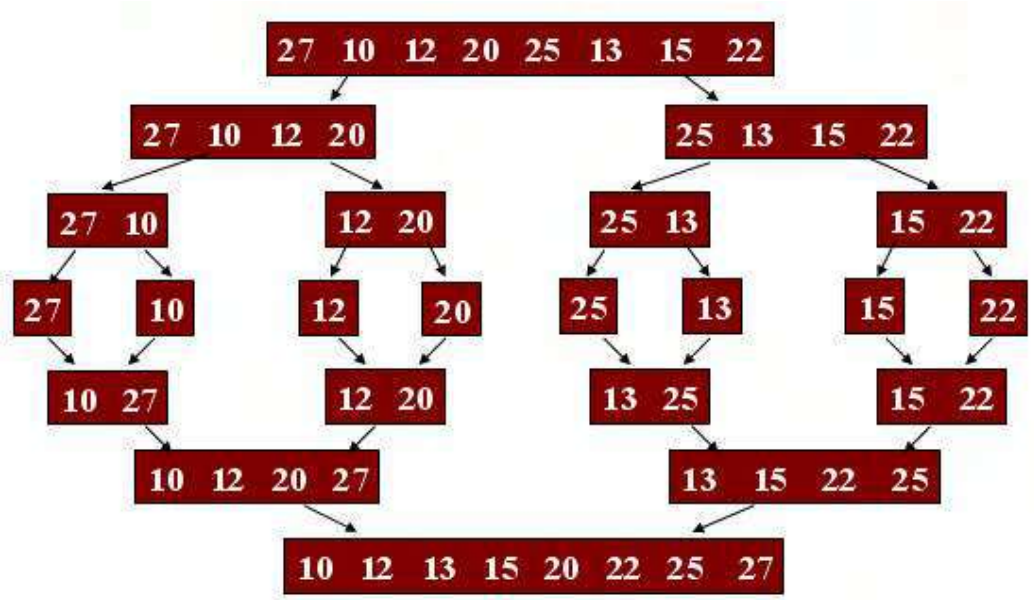
Nadie ha sido capaz de analizar el algoritmo, por lo que es difícil evaluar analíticamente los diferentes incrementos y su comparación con otros métodos, en consecuencia no se conoce su forma funcional del tiempo de ejecución.

Para la sucesión de incrementos anterior se tiene un $T(n) = n(\log n)^2$ y $n^{1,25}$. La ordenación de Shell nunca hace más de $n^{3/2}$ comparaciones (para los incrementos 1,2,13, 40.....) .

6.7.2 Ordenamiento por Mezclas Sucesivas (*merge sort*)

Se aplica la técnica divide-y-vencerás, dividiendo la secuencia de datos en dos subsecuencias hasta que las subsecuencias tengan un único elemento, luego se ordenan mezclando dos subsecuencias ordenadas en una secuencia ordenada, en forma sucesiva hasta obtener una secuencia única ya ordenada. Si $n = 1$ solo hay un elemento por ordenar, sino se hace una ordenación de mezcla de la primera mitad del arreglo con la segunda mitad. Las dos mitades se ordenan de igual forma.

Ejemplo: Se tiene un arreglo de 8 elementos, se ordenan los 4 elementos de cada arreglo y luego se mezclan. El arreglo de 4 elementos, se ordenan los 2 elementos de cada arreglo y luego se mezclan. El arreglo de 2 elementos, como cada arreglo sólo tiene $n = 1$ elemento, solo se mezclan.




```
void ordenarMezcla(TipoEle A[], int izq, int der)
{  if ( izq < der )
   {  centro = ( izq + der ) % 2;
      ordenarMezcla( A, izq, centro );
      ordenarMezcla( A, centro+1, der);
      intercalar( A, izq, centro, der );
   }
}
void intercalar(TipoEle A[], int a, int c, int b )
{  k = 0;
   i = a;
   j = c + 1;
   n = b - a;
   while ( i < c + 1 ) && ( j < b + 1 )
   {  if ( A[i] < A[j] )
      {  B[k] = A[i];
         i = i + 1;
      }
      else
      {  B[k] = A[j];
         j = j + 1;
      }
     k = k + 1;
   };
   while ( i < c + 1 )
   {  B[k] = A[i];
      i++;
      k++;
   };
   while ( j < b + 1 )
```

```
    {  B[k] = A[j];
       j++;
       k++;
    };
    i = a;
    for ( k = 0; k < n; i++ )
    {  A[i] = B[k];
       i++;
    };
};
```

Análisis del algoritmo:

La relación de recurrencia del algoritmo es $T(1) = 1$, $T(n) = 2\ T(n/2) + n$, cuya solución es $T(n) = n \log n$.

Capítulo 7

Algoritmos de Búsqueda

7.1 Introducción

En cualquier estructura de datos puede plantearse la necesidad de saber si un cierto dato está almacenado en ella, y en su caso en que posición se encuentra. Es el problema de la búsqueda.

Existen diferentes métodos de búsqueda, todos parten de un esquema iterativo, en el cual se trata una parte o la totalidad de la estructura de datos sobre la que se busca.

El tipo de estructura condiciona el proceso de búsqueda debido a los diferentes modos de acceso a los datos.

7.2 Búsqueda Lineal

La búsqueda es el proceso de localizar un registro (elemento) con un valor de llave particular. La búsqueda termina exitosamente cuando se localiza el registro que contenga la llave buscada, o termina sin éxito, cuando se determina que no aparece ningún registro con esa llave.

Búsqueda lineal, también se le conoce como búsqueda secuencial. Supongamos una colección de registros organizados como una lista lineal. El algoritmo básico de búsqueda lineal consiste en empezar al inicio de la lista e ir a través de cada registro hasta encontrar la llave indicada (k), o hasta al final de la lista. Posteriormente hay que comprobar si ha habido éxito en la búsqueda.

Es el método más sencillo en estructuras lineales. En estructuras ordenadas el proceso se optimiza, ya que al llegar a un dato con número de orden mayor que el buscado, no hace falta seguir.

La situación óptima es que el registro buscado sea el primero en ser examinado. El peor caso es cuando las llaves de todos los n registros son comparados con k (lo que se busca). El caso promedio es $n/2$ comparaciones.

Este método de búsqueda es muy lento, pero si los datos no están en orden es el único método que puede emplearse para hacer las búsquedas. Si los valores de la llave no son únicos, para encontrar todos los registros con una llave particular, se requiere buscar en toda la lista.

Mejoras en la eficiencia de la búsqueda lineal

1) Muestreo de acceso

Este método consiste en observar que tan frecuentemente se solicita cada registro y ordenarlos de acuerdo a las probabilidades de acceso detectadas.

2) Movimiento hacia el frente

Este esquema consiste en que la lista de registros se reorganicen dinámicamente. Con este método, cada vez que búsqueda de una llave sea exitosa, el registro correspondiente se mueve a la primera posición de la lista y se recorren una posición hacia abajo los que estaban antes que él.

3) Transposición

Este es otro esquema de reorganización dinámica que consiste en que, cada vez que se lleve a cabo una búsqueda exitosa, el registro correspondiente se intercambia con el anterior. Con este procedimiento, entre más accesos tenga el registro, más rápidamente avanzará hacia la primera posición. Comparado con el método de movimiento al frente, el método requiere más tiempo de actividad para reorganizar al conjunto de registros. Una ventaja de método de transposición es que no permite que el requerimiento aislado de un registro, cambie de posición todo el conjunto de registros. De hecho, un registro debe ganar poco a poco su derecho a alcanzar el inicio de la lista.

4) Ordenamiento

Una forma de reducir el número de comparaciones esperadas cuando hay una significativa frecuencia de búsqueda sin éxito es la de ordenar los registros en base al valor de la llave. Esta técnica es útil cuando la lista es una lista de excepciones, tales como una lista de decisiones, en cuyo caso la mayoría de las búsquedas no tendrán éxito. Con este método una búsqueda sin éxito termina cuando se encuentra el primer valor de la llave mayor que el buscado, en lugar de la final de la lista.

En ciertos casos se desea localizar un elemento concreto de un arreglo. En una búsqueda lineal de un arreglo, se empieza con la primera casilla del arreglo y se observa una casilla tras otra hasta que se encuentra el elemento buscado o se ha visto todas las casillas. Como el resultado de la búsqueda es un solo valor, ya sea la posición del elemento buscado o cero, puede usarse una función para efectuar la búsqueda.

En la función BUSQ_LINEAL, nótese el uso de la variable booleana encontrada.

```
Function BUSQ_LINEAL (nombres: tiponom; Tam: integer;
{Regresa el índice de la celda que contiene el elemento buscado o
regresa 0 si no está en el arreglo}
Var   encontrado: boolean;
      indice; integer:
Begin
      encontrado := false;
      indice := 1:
      Reapeat
       If nombres {índice} = sebusca then
        begin
            encontrado : = true;
            BUSQ_LINEAL := indice
        end
       else
            indice := indice + 1
      Until (encontrado) or {indice > tam);

      If not encontrado then BUSQ_LINEAL: = 0
End;
```

7.3 Búsqueda Binaria

Una búsqueda lineal funciona bastante bien para una lista pequeña. Pero si se desea buscar el número de teléfono de Jorge Perez en el directorio telefónico de Santiago, no sería sensato empezar con el primer nombre y continuar la búsqueda nombre por nombre. En un método más eficiente se aprovechará el orden alfabético de los nombres para ir inmediatamente a un lugar en la segunda mitad del directorio.

La búsqueda binaria es semejante al método usado automáticamente al buscar un número telefónico. En la búsqueda binaria se reduce sucesivamente la operación eliminando repetidas veces la mitad de la lista restante. Es claro que para poder realizar una búsqueda binaria debe tenerse una lista ordenada.

Se puede aplicar tanto a datos en listas lineales como en árboles binarios de búsqueda. Los prerrequisitos principales para la búsqueda binaria son:

- La lista debe estar ordenada en un orden especifíco de acuerdo al valor de la llave.
- Debe conocerse el número de registros.

Algoritmo

1. Se compara la llave buscada con la llave localizada al centro del arreglo.
2. Si la llave analizada corresponde a la buscada fin de búsqueda si no.
3. Si la llave buscada es menor que la analizada repetir proceso en mitad superior, sino en la mitad inferior.
4. El proceso de partir por la mitad el arreglo se repite hasta encontrar el registro o hasta que el tamaño de la lista restante sea cero , lo cual implica que el valor de la llave buscada no está en la lista.

El esfuerzo máximo para este algoritmo es de $\log_2 n$. El mínimo de 1 y en promedio ½ $\log_2 n$.

7.4 Árboles de Búsqueda

Son tipos especiales de árboles que los hacen adecuados para almacenar y recuperar información. Como en un árbol binario de búsqueda buscar información requiere recorrer un solo camino desde la raíz a una hoja. Como en un árbol balanceado el más largo camino de la raíz a una hoja es $O(\log n)$. A diferencia de un árbol binario cada nodo puede contener varios elementos y tener varios hijos. Debido a esto y a que es balanceado, permite acceder a grandes conjuntos de datos por caminos cortos.

7.5 Búsqueda por Transformación de Claves (*Hashing*)

Hasta ahora las técnicas de localización de registros vistas, emplean un proceso de búsqueda que implica cierto tiempo y esfuerzo. El siguiente método nos permite encontrar directamente el registro buscado.

La idea básica de este método consiste en aplicar una función que traduce un conjunto de posibles valores llave en un rango de direcciones relativas. Un problema potencial encontrado en este proceso, es que tal función no puede ser uno a uno; las direcciones calculadas pueden no ser todas únicas, cuando $R(k_1) = R(k_2)$. Pero: K_1 diferente de K_2 decimos que hay una **colisión**. A dos llaves diferentes que les corresponda la misma dirección relativa se les llama **sinónimos**.

A las técnicas de cálculo de direcciones también se les conoce como:

- Técnicas de almacenamiento disperso
- Técnicas aleatorias
- Métodos de transformación de llave-a-dirección
- Técnicas de direccionamiento directo
- Métodos de tabla *Hash*
- Métodos de *Hashing*

Pero el término más usado es el de *hashing*. Al cálculo que se realiza para obtener una dirección a partir de una llave se le conoce como **función *hash***.

Ventajas:

1. Se pueden usar los valores naturales de la llave, puesto que se traducen internamente a direcciones fáciles de localizar.
2. Se logra independencia lógica y física, debido a que los valores de las llaves son independientes del espacio de direcciones.
3. No se requiere almacenamiento adicional para los índices.

Desventajas:

1. No pueden usarse registros de longitud variable.
2. El archivo no está ordenado.
3. No permite llaves repetidas.
4. Solo permite acceso por una sola llave.

Costos:

- Tiempo de procesamiento requerido para la aplicación de la función *hash.*
- Tiempo de procesamiento y los accesos E/S requeridos para solucionar las colisiones.

La eficiencia de una función *hash* depende de:

1. La distribución de los valores de llave que realmente se usan.
2. El número de valores de llave que realmente están en uso con respecto al tamaño del espacio de direcciones.
3. El número de registros que pueden almacenarse en una dirección dada sin causar una colisión.
4. La técnica usada para resolver el problema de las colisiones.

Las funciones *hash* más comunes son:

- Residuo de la división
- Método de la Multiplicación
- Medio del cuadrado
- Pliegue

Hashing por Residuo de la División

En este caso, $h(k) = k \bmod m$, en que m = número primo no muy cercano a una potencia de 2—si $m = 2^p$, entonces $h(k)$ es los p bits menos significativos de k.

La idea de este método es la de dividir el valor de la llave entre un número apropiado, y después utilizar el residuo de la división como dirección relativa para el registro (dirección = llave *mod* divisor).

Mientras que el valor calculado real de una dirección relativa, dados tanto un valor de llave como el divisor, es directo; la elección del divisor apropiado puede no ser tan simple. Existen varios factores que deben considerarse para seleccionar el divisor:

1. El rango de valores que resultan de la operación "llave *mod* divisor", va desde cero hasta el divisor 1. Luego, el divisor determina el tamaño del espacio de direcciones relativas. Si se sabe que el archivo va a contener por lo menos *n* registros, entonces tendremos que hacer que divisor > *n*, suponiendo que solamente un registro puede ser almacenado en una dirección relativa dada.

2. El divisor deberá seleccionarse de tal forma que la probabilidad de colisión sea minimizada. ¿Cómo escoger este número? Mediante investigaciones se ha demostrado que los divisores que son números pares tienden a comportase pobremente, especialmente con los conjuntos de valores de llave que son predominantemente impares. Algunas investigaciones sugieren que el divisor deberá ser un número primo. Sin embargo, otras sugieren que los divisores no primos trabajan tan bién como los divisores primos, siempre y cuando los divisores no primos no contengan ningún factor primo menor de 20. Lo más común es elegir el número primo más próximo al total de direcciones.

Independientemente de que tan bueno sea el divisor, cuando el espacio de direcciones de un archivo está completamente lleno, la probabilidad de colisión crece dramáticamente. La saturación de archivo de mide mediante su **factor de carga**, el cual se define como la relación del número de registros en el archivo contra el número de registros que el archivo podría contener si estuviese completamente lleno.

Todas las funciones *hash* comienzan a trabajar probablemente cuando el archivo está casi lleno. Por lo general, el máximo factor de carga que puede tolerarse en un archivo para un rendimiento razonable es de entre el 70% y 80%.

Por ejemplo, si:

- *n* = 2,000 strings de caracteres de 8 bits, en que cada *string* es interpretado como un número entero *k* en base 256, y

- estamos dispuestos a examinar un promedio de 3 elementos en una búsqueda fallida,

Entonces podemos:

- asignar una tabla de mezcla de tamaño $m = 701$ y
- usar la función de mezcla $k(k) = k \bmod 701$.

Hashing por Método de la Multiplicación

En este caso, $h(k) = \lfloor m\ (k\ A \bmod 1) \rfloor$,

en que $0 < A < 1$; el valor de m no es crítico y típicamente se usa una potencia de 2 para facilitar el cálculo computacional de la función.

Por ejemplo, si $k = 123456$, $m = 10{,}000$, y $A = (\sqrt{5} - 1)/2 \approx 0.6180339887$, entonces $h(k) = 41$.

Hashing por Medio del Cuadrado

En esta técnica, la llave es elevada al cuadrado, después algunos dígitos específicos se extraen de la mitad del resultado para constituir la dirección relativa. Si se desea una dirección de n dígitos, entonces los dígitos se truncan en ambos extremos de la llave elevada al cuadrado, tomando n dígitos intermedios. Las mismas posiciones de n dígitos deben extraerse para cada llave.

Ejemplo:

Utilizando esta función *hashing* el tamaño del archivo resultante es de 10^n donde n es el numero de dígitos extraídos de los valores de la llave elevada al cuadrado.

Hashing por Pliegue

En esta técnica el valor de la llave es particionada en varias partes, cada una de las cuales (excepto la última) tiene el mismo número de dígitos que tiene la dirección relativa objetivo. Estas particiones son después plegadas una sobre otra y sumadas. El resultado, es la dirección relativa. Igual que para el método del medio del cuadrado, el tamaño del espacio de direcciones relativas es una potencia de 10.

Comparación entre las Funciones Hash

Aunque alguna otra técnica pueda desempeñarse mejor en situaciones particulares, la técnica del residuo de la división proporciona el mejor desempeño. Ninguna función *hash* se desempeña siempre mejor que las

otras. El método del medio del cuadrado puede aplicarse en archivos con factores de cargas bastantes bajas para dar generalmente un buen desempeño. El método de pliegues puede ser la técnica más fácil de calcular, pero produce resultados bastante erráticos, a menos que la longitud de la llave sea aproximadamente igual a la longitud de la dirección.

Si la distribución de los valores de llaves no es conocida, entonces el método del residuo de la división es preferible. Note que el *hashing* puede ser aplicado a llaves no numéricas. Las posiciones de ordenamiento de secuencia de los caracteres en un valor de llave pueden ser utilizadas como sus equivalentes "numéricos". Alternativamente, el algoritmo *hash* actúa sobre las representaciones binarias de los caracteres.

Todas las funciones *hash* presentadas tienen destinado un espacio de tamaño fijo. Aumentar el tamaño del archivo relativo creado al usar una de estas funciones, implica cambiar la función *hash*, para que se refiera a un espacio mayor y volver a cargar el nuevo archivo.

Métodos para Resolver el Problema de las Colisiones

Considere las llaves K_1 y K_2 que son sinónimas para la función *hash* R. Si K_1 es almacenada primero en el archivo y su dirección es $R(K_1)$, entonces se dice que K_1 esta almacenado en su dirección de origen.

Existen dos **métodos básicos** para determinar dónde debe ser alojado K_2:

- **Direccionamiento abierto.** Se encuentra entre dirección de origen para K_2 dentro del archivo.

- **Separación de desborde (Área de desborde).** Se encuentra una dirección para K_2 fuera del área principal del archivo, en un área especial de desborde, que es utilizada exclusivamente para almacenar registro que no pueden ser asignados en su dirección de origen

Los métodos más conocidos para resolver colisiones son:

Sondeo lineal

Es una técnica de direccionamiento abierto. Este es un proceso de búsqueda secuencial desde la dirección de origen para encontrar la siguiente localidad vacía. Esta técnica es también conocida como método de desbordamiento consecutivo.

Para almacenar un registro por *hashing* con sondeo lineal, la dirección no debe caer fuera del límite del archivo. En lugar de terminar cuando el límite del espacio de dirección se alcanza, se regresa al inicio del espacio y sondeamos desde ahí. Por lo que debe ser posible detectar si la dirección

base ha sido encontrada de nuevo, lo cual indica que el archivo está lleno y no hay espacio para la llave.

Para la búsqueda de un registro por *hashing* con sondeo lineal, los valores de llave de los registros encontrados en la dirección de origen, y en las direcciones alcanzadas con el sondeo lineal, deberá compararse con el valor de la llave buscada, para determinar si el registro objetivo ha sido localizado o no.

El sondeo lineal puede usarse para cualquier técnica de *hashing*. Si se emplea sondeo lineal para almacenar registros, también deberá emplearse para recuperarlos.

Doble *hashing*

En esta técnica se aplica una segunda función *hash* para combinar la llave original con el resultado del primer *hash*. El resultado del segundo *hash* puede situarse dentro del mismo archivo o en un archivo de sobreflujo independiente; de cualquier modo, será necesario algún método de solución si ocurren colisiones durante el segundo *hash*.

La ventaja del método de separación de desborde es que reduce la situación de una doble colisión, la cual puede ocurrir con el método de direccionamiento abierto, en el cual un registro que no está almacenado en su dirección de origen desplazará a otro registro, el que después buscará su dirección de origen. Esto puede evitarse con direccionamiento abierto, simplemente moviendo el registro extraño a otra localidad y almacenando al nuevo registro en la dirección de origen ahora vacía.

Puede ser aplicado como cualquier direccionamiento abierto o técnica de separación de desborde.

Para ambas métodos para la solución de colisiones existen técnicas para mejorar su desempeño como:

1. **Encadenamiento de sinónimos**

   Una buena manera de mejorar la eficiencia de un archivo que utiliza el cálculo de direcciones, sin directorio auxiliar para guiar la recuperación de registros, es el encadenamiento de sinónimos. Mantener una lista ligada de registros, con la misma dirección de origen, no reduce el número de colisiones, pero reduce los tiempos de acceso para recuperar los registros que no se encuentran en su localidad de origen. El encadenamiento de sinónimos puede emplearse con cualquier técnica de solución de colisiones.

   Cuando un registro debe ser recuperado del archivo, solo los sinónimos de la llave objetivo son accesados.

2. Direccionamiento por cubetas

Otro enfoque para resolver el problema de las colisiones es asignar bloques de espacio (cubetas), que pueden acomodar ocurrencias múltiples de registros, en lugar de asignar celdas individuales a registros. Cuando una cubeta es desbordada, alguna nueva localización deberá ser encontrada para el registro. Los métodos para el problema de sobrecupo son básicamente los mismos que los métodos para resolver colisiones.

Comparación entre Sondeo Lineal y Doble *Hashing*

De ambos métodos resultan distribuciones diferentes de sinónimos en un archivo relativo. Para aquellos casos en que el factor de carga es bajo (< 0.5), el sondeo lineal tiende a agrupar los sinónimos, mientras que el doble *hashing* tiende a dispersar los sinónimos más ampliamente a través del espacio de direcciones.

El doble *hashing* tiende a comportarse casi también como el sondeo lineal con factores de carga pequeños (< 0.5), pero actúa un poco mejor para factores de carga mayores. Con un factor de carga > 80 %, el sondeo lineal, por lo general, resulta tener un comportamiento terrible, mientras que el doble *hashing* es bastante tolerable para búsquedas exitosas, pero no así en búsquedas no exitosas.

7.6 Búsqueda en Textos

La búsqueda de patrones en un texto es un problema muy importante en la práctica. Sus aplicaciones en computación son variadas, como por ejemplo la búsqueda de una palabra en un archivo de texto o problemas relacionados con biología computacional, en donde se requiere buscar patrones dentro de una secuencia de ADN, la cual puede ser modelada como una secuencia de caracteres (el problema es más complejo que lo descrito, puesto que se requiere buscar patrones en donde ocurren alteraciones con cierta probabilidad, esto es, la búsqueda no es exacta).

En este capítulo se considerará el problema de buscar la ocurrencia de un patrón dentro de un texto. Se utilizarán las siguientes convenciones:

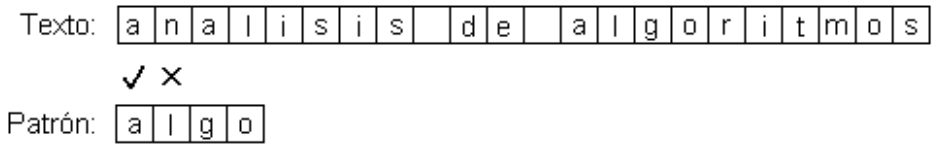
n denotará el largo del texto en donde se buscará el patrón, es decir, *texto* = $a_1 a_2 \ldots a_n$. Donde m denotará el largo del patrón a buscar, es decir, *patrón* = $b_1 b_2 \ldots b_m$.

Por ejemplo:

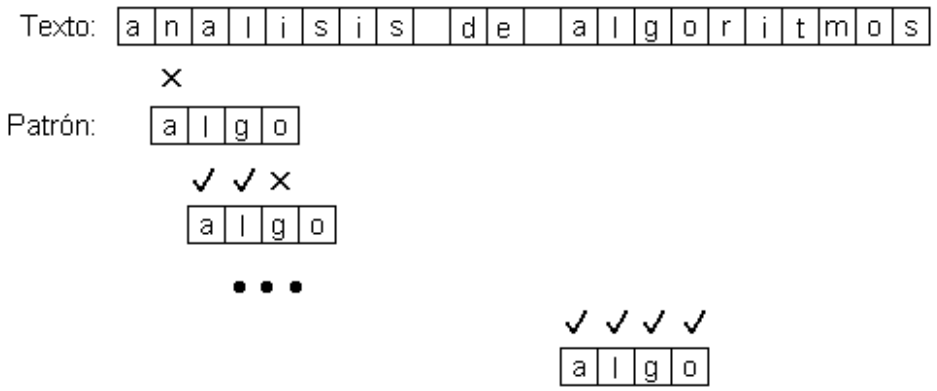
Texto = "analisis de algoritmos" n=22 y m=4
Patrón = "algo"

7.6.1 Algoritmo de Fuerza Bruta

Se alinea la primera posición del patrón con la primera posición del texto, y se comparan los caracteres uno a uno hasta que se acabe el patrón, esto es, se encontró una ocurrencia del patrón en el texto, o hasta que se encuentre una discrepancia.



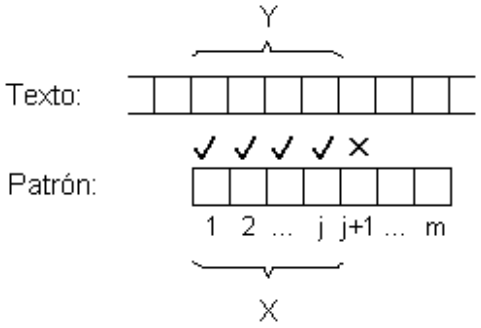
Si se detiene la búsqueda por una discrepancia, se desliza el patrón en una posición hacia la derecha y se intenta calzar el patrón nuevamente.



En el peor caso este algoritmo realiza $O(m \cdot n)$comparaciones de caracteres.

7.6.2 Algorimo *Knuth-Morris-Pratt*

Suponga que se está comparando el patrón y el texto en una posición dada, cuando se encuentra una discrepancia.



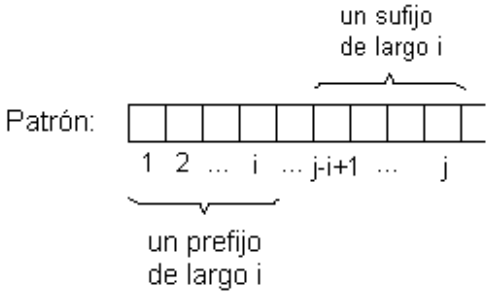
Sea *X* la parte del patrón que calza con el texto, e *Y* la correspondiente parte del texto, y suponga que el largo de *X* es *j*. El algoritmo de fuerza bruta mueve el patrón una posición hacia la derecha, sin embargo, esto puede o no puede ser lo correcto en el sentido que los primeros *j-1* caracteres de *X* pueden o no pueden calzar los últimos *j-1* caracteres de *Y*.

La observación clave que realiza el algoritmo *Knuth-Morris-Pratt* (en adelante KMP) es que *X es igual a Y*, por lo que la pregunta planteada en el párrafo anterior puede ser respondida mirando solamente el patrón de búsqueda, lo cual permite precalcular la respuesta y almacenarla en una tabla.

Por lo tanto, si deslizar el patrón en una posición no funciona, se puede intentar deslizarlo en 2, 3, ..., hasta *j* posiciones.

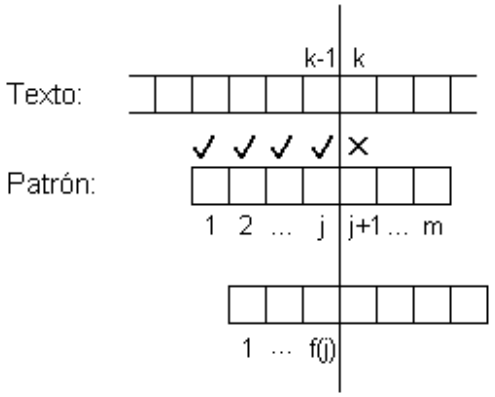
Se define la *función de fracaso* (***failure function***) del patrón como:

$$f(j) = max(i < j \mid b_1 \,..\, b_i = b_{j-i+1} \,..\, b_j)$$



Intuitivamente, *f(j)* es el largo del mayor prefijo de *X* que además es sufijo de *X*. Note que *j = 1* es un caso especial, puesto que si hay una discrepancia en b_1 el patrón se desliza en una posición.

Si se detecta una discrepancia entre el patrón y el texto cuando se trata de calzar b_{j+1}, se desliza el patrón de manera que $b_{f(j)}$ se encuentre donde b_j se encontraba, y se intenta calzar nuevamente.



Suponiendo que se tiene *f(j)* precalculado, la implementación del algoritmo KMP es la siguiente:

```
// n = largo del texto
// m = largo del patrón
// Los índices comienzan desde 1

int k=0;
int j=0;
while (k<n && j<m)
{
  while (j>0 && texto[k]!=patron[j])
  {
     j=f[j];
  }
  if (texto[k+1])==patron[j+1]))
  {
     j++;
  }
  k++;
}
// j==m => calce, j el patrón no estaba en el texto
```

Ejemplo:

Patron = "a a b a a a"
1 2 3 4 5 6

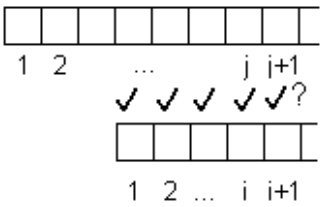
j	1	2	3	4	5	6
f(j)	0	1	0	1	2	2

Texto: "a a a a b a a b a a a b b"
j = 0 1 2
1 2
1 2 3 4 5
2 3 4 5 6 → calce!

El tiempo de ejecución de este algoritmo no es difícil de analizar, pero es necesario ser cuidadoso al hacerlo. Dado que se tienen dos ciclos anidados, se puede acotar el tiempo de ejecución por el número de veces que se ejecuta el ciclo externo (menor o igual a n) por el número de veces que se ejecuta el ciclo interno (menor o igual a m), por lo que la cota es igual a $O(m\,n)$, ¡que es igual a lo que demora el algoritmo de fuerza bruta!.

El análisis descrito es pesimista. Note que el número total de veces que el ciclo interior es ejecutado es menor o igual al número de veces que se puede decrementar j, dado que $f(j)<j$. Pero j comienza desde cero y es siempre mayor o igual que cero, por lo que dicho número es menor o igual al número de veces que j es incrementado, el cual es menor que n. Por lo tanto, el tiempo total de ejecución es $O(n)$. Por otra parte, k nunca es decrementado, lo que implica que el algoritmo nunca se devuelve en el texto.

Queda por resolver el problema de definir la función de fracaso, $f(j)$. Esto se puede realizar inductivamente. Para empezar, $f(1)=0$ por definición. Para calcular $f(j+1)$ suponga que ya se tienen almacenados los valores de $f(1), f(2), ..., f(j)$. Se desea encontrar un $i+1$ tal que el $(i+1)$-ésimo carácter del patrón sea igual al $(j+1)$-ésimo carácter del patrón.



Para esto se debe cumplir que $i=f(j)$. Si $b_{i+1}=b_{j+1}$, entonces $f(j+1)=i+1$. En caso contrario, se reemplaza i por $f(i)$ y se verifica nuevamente la condición.

El algoritmo resultante es el siguiente (note que es similar al algoritmo KMP):

```
// m es largo del patrón
// los índices comienzan desde 1
int[] f=new int[m];
f[1]=0;
int j=1;
int i;
while (j<m)
{
 i=f[j];
 while (i>0 && patron[i+1]!=patron[j+1])
 {
  i=f[i];
 }
 if (patron[i+1]==patron[j+1])
 {
  f[j+1]=i+1;
 }
 else
 {
  f[j+1]=0;
 }
 j++;
}
```

El tiempo de ejecución para calcular la función de fracaso puede ser acotado por los incrementos y decrementos de la variable *i*, que es $O(m)$.

Por lo tanto, el tiempo total de ejecución del algoritmo, incluyendo el preprocesamiento del patrón, es $O(n+m)$.

7.6.3 Algoritmo de Boyer-Moore

Hasta el momento, los algoritmos de búsqueda en texto siempre comparan el patrón con el texto de izquierda a derecha. Sin embargo, suponga que la comparación ahora se realiza de derecha a izquierda: si hay una discrepancia en el último carácter del patrón y el carácter del texto no aparece en todo el patrón, entonces éste se puede deslizar *m* posiciones sin realizar niguna comparación extra. En particular, no fue necesario comparar los primeros *m-1* caracteres del texto, lo cual indica que podría realizarse una búsqueda en el texto con menos de *n* comparaciones; sin embargo, si el carácter discrepante del texto se encuentra dentro del patrón, éste podría desplazarse en un número menor de espacios.

El método descrito es la base del algoritmo *Boyer-Moore*, del cual se estudiarán dos variantes: *Horspool* y *Sunday*.

Boyer-Moore-Horspool (BMH)

El algoritmo BMH compara el patrón con el texto de derecha a izquierda, y se detiene cuando se encuentra una discrepancia con el texto. Cuando esto sucede, se desliza el patrón de manera que la letra del texto que estaba alineada con b_m, denominada c, ahora se alinie con algún b_j, con $j<m$, si dicho calce es posible, o con b_0, un carácter ficticio a la izquierda de b_1, en caso contrario (este es el mejor caso del algoritmo).

Para determinar el desplazamiento del patrón se define la *función siguiente* como:

0 si c no pertenece a los primeros $m-1$ caracteres del patrón (¿Por qué no se considera el carácter b_m?).

j si c pertenece al patrón, donde $j<m$ corresponde al mayor índice tal que $b_j==c$.

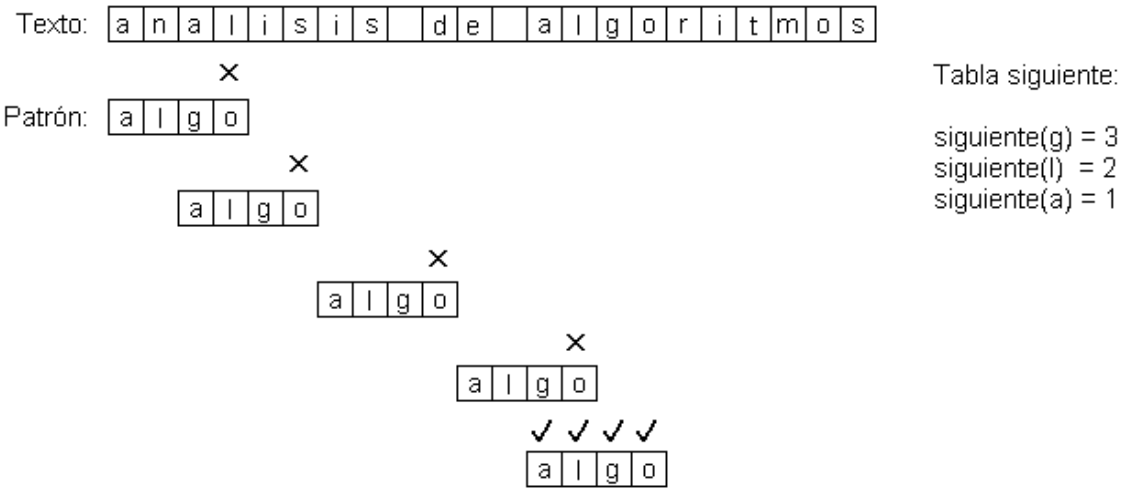
Esta función sólo depende del patrón y se puede precalcular antes de realizar la búsqueda.

El algoritmo de búsqueda es el siguiente:

```
// m  es el largo del patrón
// los índices comienzan desde 1

int k=m;
int j=m;
while (k <=n && j>=1)
{
 if (texto[k-(m-j)]==patron[j])
 {
  j--;
 }
 else
 {
  k=k+(m-siguiente(a[k]));
  j=m;
 }
}
// j==0 => calce!, j>=0 => no hubo calce.
```

Ejemplo de uso del algoritmo BMH:



Se puede demostrar que el tiempo promedio que toma el algoritmo BMH es:

$$O\left(n\left(\frac{1}{m}+\frac{1}{2c}\right)\right)$$

donde c es el tamaño del alfabeto ($c<<n$). Para un alfabeto razonablemente grande, el algoritmo es $O\left(\frac{n}{m}\right)$.

En el peor caso, BMH tiene el mismo tiempo de ejecución que el algoritmo de fuerza bruta.

Boyer-Moore-Sunday (BMS)

El algoritmo BMH desliza el patrón basado en el símbolo del texto que corresponde a la posición del último carácter del patrón. Este siempre se desliza al menos una posición si se encuentra una discrepancia con el texto.

Es fácil ver que si se utiliza el carácter una posición más adelante en el texto como entrada de la función siguiente el algoritmo también funciona, pero en este caso es necesario considerar el patrón completo al momento de calcular los valores de la función siguiente. Esta variante del algoritmo es conocida como *Boyer-Moore-Sunday* (BMS).

¿Es posible generalizar el argumento, es decir, se pueden utilizar caracteres más adelante en el texto como entrada de la función siguiente? La respuesta es no, dado que en ese caso puede ocurrir que se salte un calce en el texto.

Capítulo 8

Teoría de Grafos

Un *grafo* (o multigrafo), es una estructura muy importante utilizada en Informática y también en ciertos ramos de Matemáticas, como por ejemplo, en Investigación de Operaciones. Muchos problemas de difícil resolución, pueden ser expresados en forma de grafo y consecuentemente resuelto usando algoritmos de búsqueda y manipulación estándar.

Entre las múltiples aplicaciones que tienen estas estructuras podemos mencionar:

- Modelar diversas situaciones reales, tales como: sistemas de aeropuertos, flujo de tráfico, etc.

- Realizar planificaciones de actividades, tareas del computador, planificar operaciones en lenguaje de máquinas para minimizar tiempo de ejecución.

Simplemente, un **grafo** es una estructura de datos no lineal, que puede ser considerado como un conjunto de **vértices** (o nodos, dependiendo de la bibliografía) y **arcos** que conectan esos vértices.

Matemáticamente tenemos $G=(V, E)$. Si u y v son elementos de V, entonces un *arco* se puede representar por (u,v).

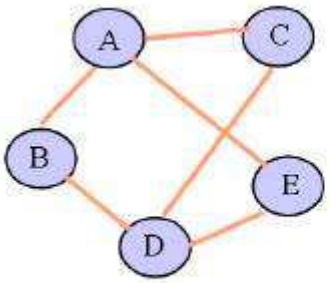
Los grafos también se pueden clasificar como:

- Grafo No Dirigido

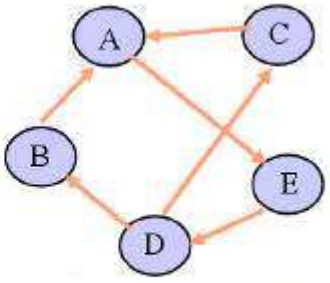
- Grafo Dirigido (o Digrafo)

En el primer caso, los arcos no tienen una dirección y, por lo tanto, (u,v) y (v,u) representan el mismo arco. En un ***Grafo No Dirigido***, dos vértices se dicen adyacentes si existe un arco que une a esos dos vértices.

En el segundo caso, los arcos tienen una dirección definida, y así (u,v) y (v,u) entonces representan arcos diferentes. Un ***Grafo Dirigido*** (o *digrafo*) puede también ser fuertemente conectado si existe un camino desde cualquier vértice hacia otro cualquier vértice.



Grafo No-Dirigido



Grafo Dirigido (Digrafo)

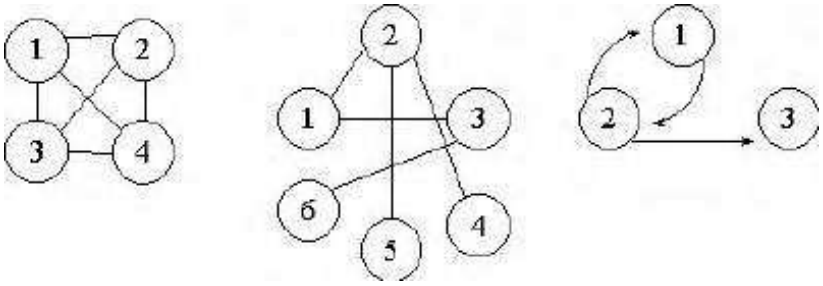
Ejemplos de grafos (dirigidos y no dirigidos):

G1 = (V1, E1)
V1 = {1, 2, 3, 4} E1 = {(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}

G2 = (V2, E2)
V2 = {1, 2, 3, 4, 5, 6} E2 = {(1, 2), (1, 3), (2, 4), (2, 5), (3, 6)}

G3 = (V3, E3)
V3 = {1, 2, 3} E3 = {<1, 2>, <2, 1>, <2, 3>}

Gráficamente estas tres estructuras de vértices y arcos se pueden representar de la siguiente manera:



8.1 Definiciones Básicas

Un grafo puede ser direccional o no direccional.

Un ***grafo direccional*** G es un par (V, E):

- V es un conjunto finito de *vértices* y
- E es un conjunto de *aristas* que representa una relación binaria sobre $V — E \subseteq V \times V$.

En un ***grafo no direccional*** $G = (V, E)$, el conjunto de aristas E consiste en pares no ordenados de vértices.

Una **arista** es un conjunto $\{u, v\}$, en que $u, v \in V$ y $u \neq v$, y que representamos como (u, v).

Si (u, v) es una **arista** en un grafo:

- dirigido, entonces (u, v) es ***incidente*** *desde* o *sale* del vértice u y es *incidente a* o *entra* al vértice v;
- no dirigido, entonces (u, v) es ***incidente*** *sobre* u y v.

v es ***adyacente*** a u; si el grafo es no dirigido, entonces la relación de adyacencia es simétrica.

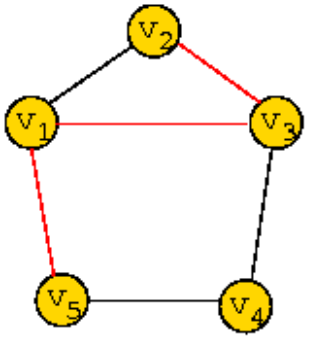
Un **camino (o ruta)** es una secuencia de vértices $v_1,...,v_n$ tal que (v_i,v_{i+1}) pertenece a E. La **longitud** de un camino es la cantidad de arcos que éste contiene.

Un ***camino*** de longitud k desde un vértice u a un vértice u' en un grafo $G = (V, E)$ es una secuencia $v_0, v_1, .., v_k$ de vértices tal que:

- $u = v_0$,
- $u' = v_k$ y
- $(v_{i-1}, v_i) \in E$ para $i = 1, 2, .., k$.

Si hay un **camino** p desde u a u', entonces u' es ***alcanzable*** desde u vía p.

Un camino (en rojo)

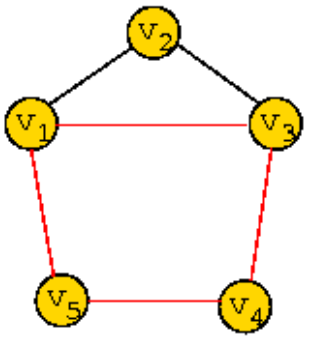


Un **camino simple** es aquel donde todos sus vértices son distintos. Sólo el primero y el último pueden coincidir. Un **ciclo** en un grafo dirigido es el camino de longitud mayor o igual que 1 donde $w_1 = w_n$. Se llamará **ciclo simple** si el camino es simple.

Si $a=\{u,v\}$ es una arista de G escribiremos sólo $a=uv$, y diremos que ***a*** une los vértices ***u*** y ***v*** o que ***u*** y ***v*** son **extremos** de ***a***. Una arista $a=uu$ se llama ***bucle***. Una arista que aparece repetida en E se llama **arista múltiple**.

Un ***ciclo*** es un camino simple y cerrado.

Un ciclo (en rojo)



Un grafo es ***conexo*** si desde cualquier vértice existe un camino hasta cualquier otro vértice del grafo.

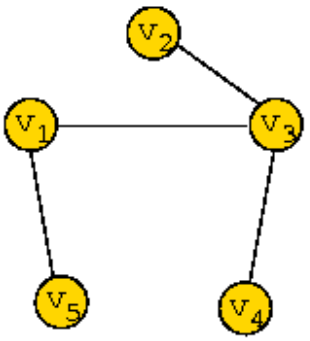
Un **grafo** es **conexo** si para cada par de vértices ***u*** y ***v*** existe un camino de ***u*** a ***v***. Si *G* es un grafo no conexo (o disconexo), cada uno de sus subgrafos conexos maximales se llama **componente conexa** de *G*.

Un digrafo $D=(V,E)$ es **fuertemente conexo** si para todo par de vértices ***u*** y ***v*** existe un camino dirigido que va de ***u*** a ***v***.

Dado un digrafo *D*, podemos considerar el grafo *G* no dirigido que se obtiene al sustituir cada arco (u,v) por la arista (u,v). Si este grafo es conexo, diremos que el digrafo *D* es **débilmente conexo**.

Se dice que un grafo no dirigido es un ***árbol*** si es conexo y acíclico.

Un arbol



En algunos textos llaman *grafo* al que aquí se denomina *grafo simple*, permitiendo la presencia de aristas múltiples en los *multigrafos* y de bucles en los *seudografos*.

Dos vértices son **adyacentes** si son extremos de una misma arista. Dos aristas son **adyacentes** si tienen un extremo común. Un vértice y una arista son **incidentes** si el vértice es extremo de la arista. Un vértice es **aislado** si no tiene otros vértices adyacentes.

Un **grafo completo** es un grafo simple en el que todo par de vértices está unido por una arista. Se representa con K_n al grafo completo de n vértices.

Un grafo $G=(V,E)$ se llama **bipartito** (o bipartido) si existe una partición de V, $V=X \cup Y$, tal que cada arista de G une un vértice de X con otro de Y. (Se designa por $K_{r,s}$ al **grafo bipartito completo** en que $|X|= r$ e $|Y|= s$, y hay una arista que conecta cada vértice de X con cada vértice de Y).

El número de vértices de un grafo G es su **orden** y el número de aristas su **tamaño**. Designaremos el orden con n y el tamaño con q y utilizaremos la notación de grafo (n,q).

Dos grafos $G=(V,E)$ y $G'=(V',E')$ son **isomorfos** si existe una biyección $f:V \rightarrow V'$ que conserva la adyacencia. Es decir, $\forall u,v \in V$, u y v son adyacentes en $G \Leftrightarrow f(u)$ y $f(v)$ son adyacentes en G'.

Un **subgrafo** de $G=(V,E)$ es otro grafo $H=(V',E')$ tal que $V' \subseteq V$ y $E' \subseteq E$. Si $V'= V$ se dice que H es un subgrafo **generador.**

Se llama **grado** de un vértice v al número de aristas que lo tienen como extremo, (cada bucle se cuenta, por tanto, dos veces). Se designa por $d(v)$.

Un **grafo regular** es un grafo simple cuyos vértices tienen todos los mismos grados.

A la sucesión de los grados de los vértices de G se le denomina **sucesión de grados** del grafo G. Una sucesión de enteros no negativos se dice sucesión **gráfica** si es la sucesión de grados de un grafo simple.

8.2 Representaciones de Grafos

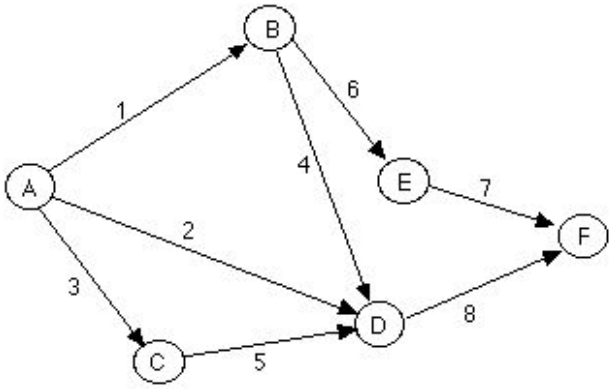
Existen dos formas de mantener un grafo G en la memoria de una computadora, una se llama ***Representación secuencial*** de G, la cual se basa en la matriz de adyacencia.

La otra forma, es la llamada ***Representación enlazada*** de G y se basa en listas enlazadas de vecinos. Independientemente de la forma en que se mantenga un grafo G en la memoria de una computadora, el grafo G normalmente se introduce en la computadora por su definición formal: Un conjunto de nodos y un conjunto de aristas.

8.2.1 Matriz y Lista de Adyacencia

Dado un grafo $G = (V, E)$, podemos representar un grafo a través de una matriz de dimensión V x V, donde el contenido podrá ser de números binarios (0;1) o de número enteros (-1;0;1).

Para ejemplificar, analicemos el grafo **G**:



La **cardinalidad** de vértices del grafo ***G*** es **6**, por tanto, para su representación, deberemos tener una matriz de adyacencias 6x6. Utilizando valores binarios, podemos representarla de esta forma:

ma[*i,j*] =

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	0	0	0	1	1	0
C	0	0	0	1	0	0
D	0	0	0	0	0	1
E	0	0	0	0	0	1
F	0	0	0	0	0	0

Conforme observamos, la matriz de adyacencias está formada siguiendo la siguiente regla: **ma[i,j] = 1, si *i* es adyacente a *j*, y 0 en caso contrario.** Como estamos trabajando con un dígrafo, debemos establecer cual es el origen y cual es el destino. En el ejemplo presentado, el origen está definido por la letra indicadora de línea. Por ejemplo, *A* está conectado con *B*, más no en sentido contrario, por eso *ma[A,B]* es 1 y *ma[B,A]* es 0.

Para resolver esto, podemos hacer una representación alternativa: **ma[*i,j*] = -1 si *i* es origen de adyacencia con *j*, 1 si *i* es el destino de adyacencia con *j*, y es 0 para los demás vértices no envueltos en la adyacencia** . Esto es sintetizado en la siguiente matriz:

$$ma[i,j] = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & -1 & -1 & -1 & 0 & 0 \\ 1 & 0 & 0 & -1 & -1 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Como ejemplo, observemos el elemento *ma[A,B]* , *ma[A,C]* e *ma[A,D]* que poseen valor -1. Esto indica que *A* es origem de arcos para *C*, *D* y *E*. También observemos *ma[F,D]* y *ma[F,E]* , con valor 1, indicando que *F* recibe los arcos de *D* y *E*.

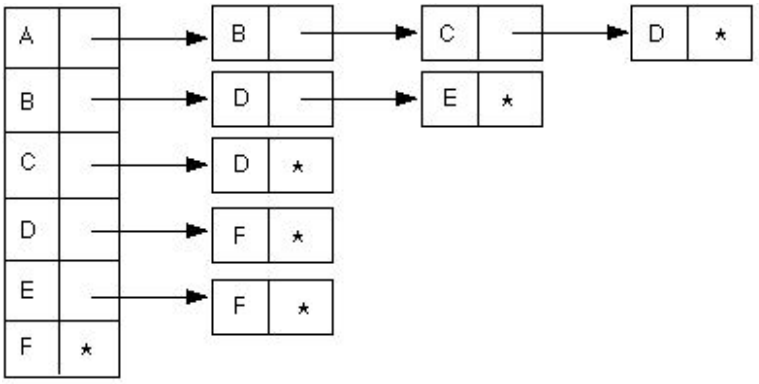
A pesar de la metodología empleada, se observa que para una aplicación dada que necesite de alteración del grafo, sería inadecuada la representación a través de estructuras fijas, exigiendo, entonces, estructuras dinámicas.

Lista de Adyacencias

Para que sea posible remodelar un grafo en tiempo de ejecución, se torna necesaria la utilización dinámica de su representación. Por eso, la representación de adyacencias entre vértices puede ser hecha a través de listas lineales.

Su construcción es realizada por un vector dinámico con listas encadenadas formando un índice de vértices. De cada elemento de índice parte una lista encadenada describiendo los vértices adyacentes conectados.

Como ejemplo, para el grafo **G** presentado anteriormente, visualizaremos la siguiente representación:



La lista encadenada es formada por nodos que contienen el dato del vértice (letra) y el puntero para el vértice adyacente a lo indicado en el índice (vector descriptor de vértices). Eventualmente los nodos pueden exigir otros campos, tales como marcas de visita al vértice, datos adicionales para procesamiento de la secuencia del grafo, etc.

Esta es la forma de representación más flexible para la representación de grafos. Obviamente que aplicaciones específicas permitirán el uso de otras formas de representación. Cabe aquí destacar que es posible la descripción de grafos a través de sus aristas, lo que puede demandar una matriz para su descripción.

La suma de las longitudes de todas las listas es $|E|$, si G es direccional, y $2|E|$ si G es no direccional —la cantidad de memoria necesaria es siempre $\boldsymbol{O}(\max(V, E)) = \boldsymbol{O}(V + E)$.

La *matriz de adyacencias* de $G = (V, E)$ supone que los vértices están numerados 1, 2, .., $|V|$ arbitrariamente, y consiste en una matriz $\mathbf{A} = (a_{ij})$ de $|V| \times |V|$, tal que:

$$a_{ij} = \begin{cases} 1 & \text{si } (i, j) \in E, \\ 0 & \text{en otro caso.} \end{cases}$$

Esta representación requiere $\Theta(V^2)$ memoria, independientemente del número de aristas en G.

La **matriz de adyacencia** siempre es simétrica porque $a_{ij} = a_{ji}$. La **lista de Adyacencia** es una lista compuesta por vértices y una sublista conteniendo las aristas que salen de él.

En el caso de las **listas de adyacencia** el espacio ocupado es $O(V + E)$, muy distinto del necesario en la matriz de adyacencia, que es de $O(V^2)$. La representación por listas de adyacencia, por tanto, será más adecuada para grafos dispersos.

8.2.2 Matriz y Lista de Incidencia

Este tipo de matriz representa un grafo a partir de sus aristas. Como exige muchas veces la utilización de una matriz mayor dado que el método de la matriz de adyacencias, no esta tan utilizada en cuanto a aquella. La matriz alojada deberá tener dimensiones V x E.

El principio de esta representación está en la regla: **mi[*i,j*] = 1 si el vértice *i* incide con la arista *j*, y 0 en caso contrario** . Ejemplificando a partir del grafo ***G*** anteriormente presentado, tendremos una matriz:

$$mi[i,j] = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

8.3 Recorridos de Grafos

En muchas aplicaciones es necesario visitar todos los vértices del grafo a partir de un nodo dado. Algunas aplicaciones son:

- Encontrar ciclos
- Encontrar componentes conexas
- Encontrar árboles cobertores

Hay dos enfoques básicos:

- Recorrido (o búsqueda) en amplitud (***breadth-first search***): Se visita a todos los vecinos directos del nodo inicial, luego a los vecinos de los vecinos, etc.

- Recorrido (o búsqueda) en profundidad (***depth-first search***): La idea es alejarse lo más posible del nodo inicial (sin repetir nodos), luego devolverse un paso e intentar lo mismo por otro camino.

8.3.1 Recorridos en Amplitud

La Búsqueda en Amplitud (BFS) es uno de los algoritmos más simples para recorrer un grafo y es el modelo de muchos algoritmos sobre grafos.

Dado un grafo $G = (V, E)$ y un vértice de partida s, BFS explora sistemáticamente las aristas de G para descubrir todos los vértices alcanzables desde s:

- Calcula la distancia — mínimo número de aristas — de s a cada vértice alcanzable.
- Produce un árbol con raíz s que contiene todos los vértices alcanzables.
- Para cualquier vértice v alcanzable desde s, la ruta de s a v en el árbol contiene el mínimo número de aristas — es una *ruta más corta.*
- Funciona en grafos direccionales y no direccionales.

BFS expande la frontera entre vértices descubiertos y no descubiertos uniformemente en toda la amplitud de la frontera.

Descubre todos los vértices a distancia k antes de descubrir cualquier vértice a distancia $k + 1$.

BFS pinta cada vértice BLANCO, PLOMO o NEGRO, según su estado:

- Todos los vértices empiezan BLANCOs.
- Un vértice es *descubierto* la primera vez que es encontrado, y se pinta PLOMO.
- Cuando se ha descubierto todos los vértices adyacentes a un vértice PLOMO, éste se pinta NEGRO.
- Los vértices PLOMOs pueden tener vértices adyacentes BLANCOs — representan la frontera entre vértices descubiertos y no descubiertos.

BFS construye un *árbol de amplitud*:

- Inicialmente, el árbol sólo contiene su raíz —el vértice s.
- Cuando se descubre un vértice BLANCO v mientras se explora la lista de adyacencias de un vértice u, v y la arista (u, v) se agregan al árbol, y u se convierte en el *predecesor* o padre de v en el árbol.

En la siguiente versión de BFS:

- El grafo se representa mediante listas de adyacencias.
- Para cada vértice u, se almacena: el color en $color[u]$, el predecesor en $\pi[u]$, y la distancia desde s en $d[u]$.
- El conjunto de vértices PLOMOs se maneja en la cola Q.

El árbol de amplitud o *grafo predecesor* de G se define como

$G_\pi = (V_\pi, E_\pi)$, en que:

$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$, y

$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$;

Las aristas en E_π se llaman *aristas de árbol*.

```
BFS(G, s) :
    for each u ∈ V – {s} do
            color[u] ← BLANCO
            d[u] ← ∞
            π[u] ← NIL
    color[s] ← PLOMO;
    d[s] ← 0;
    π[s] ← NIL
    Q ← {s}
    while Q ≠ Ø do
            u ← head[Q]
            for each v ∈ α[u] do
                    if color[v] = BLANCO then
                            color[v] ← PLOMO
                            d[v] ← d[u] + 1
                            π[v] ← u
                            ENQUEUE(Q, v)
            DEQUEUE(Q)
            color[u] ← NEGRO
```

El tiempo total de ejecución de BFS es $O(V + E)$.

8.3.2 Recorridos en Profundidad

A medida que recorremos el grafo, iremos numerando correlativamente los nodos encontrados (1,2,...). Suponemos que todos estos números son cero inicialmente, y utilizamos un contador global n, también inicializado en cero.

La Búsqueda en Profundidad (DFS) busca más adentro en el grafo mientras sea posible:

- Las aristas se exploran a partir del vértice v más recientemente descubierto.
- Cuando todas las aristas de v han sido exploradas, la búsqueda retrocede para explorar aristas que salen del vértice a partir del cual se descubrió v.
- Continuamos hasta que se ha descubierto todos los vértices alcanzables desde el vértice de partida.

- Si quedan vértices no descubiertos, se elige uno como nuevo vértice de partida y se repite la búsqueda.

DFS construye un *subgrafo predecesor*:

- Cuando descubre un vértice v durante la exploración de la lista de adyacencias de un vértice u, asigna u al predecesor $\pi[v]$ de v.
- Este subgrafo o *bosque de profundidad* puede constar de varios *árboles de profundidad*, porque la búsqueda puede repetirse desde varios vértices de partida.

El subgrafo predecesor se define como como $G_\pi = (V, E_\pi)$:

$$E_\pi = \{(\pi[v], v) : v \in V \wedge \pi[v] \neq \text{NIL}\}$$

las aristas en E_π se llaman *aristas de árbol.*

DFS pinta cada vértice BLANCO, PLOMO o NEGRO:

- Todos los vértices empiezan BLANCOs.
- Cuando un vértice es *descubierto*, se pinta PLOMO.
- Cuando un vértice es *terminado* —se ha examinado toda la lista de adyacencias del vértice —se pinta NEGRO.
- Se garantiza que cada vértice forma parte de sólo un árbol de profundidad —éstos son disjuntos.

En la siguiente versión recursiva de DFS:

- G puede ser direccional o no direccional.
- A cada vértice u, se le asocia los tiempos
- $d[u]$ al descubrirlo —cuando u se pinta PLOMO —y
- $f[u]$ al terminar la exploración de $\alpha[u]$ —cuando u se pinta NEGRO.

```
DFS(G) :
    for each u ∈ V do
        color[u] ← BLANCO
        π[u] ← NIL
    t ← 0
    for each u ∈ V do
        if color[u] = BLANCO then
            VISITAR(u)

VISITAR(u) :
    color[u] ← PLOMO
```

$t \leftarrow t + 1$

$d[u] \leftarrow t$

for each $v \in \alpha[u]$ **do**
if $color[v]$ = BLANCO **then**

$\pi[v] \leftarrow u$
VISITAR(v)

$color[u] \leftarrow$ NEGRO

$t \leftarrow t + 1$

$f[u] \leftarrow t$

El tiempo total de ejecución de DFS es $\Theta(V + E)$.

8.4 Grafos con Pesos

Un grafo con peso (o grafo con costo) es un grafo G más una función de *peso* o costo:

$$f: E(G) \rightarrow R$$

En general, salvo mención expresa en contrario, consideramos sólo grafos con pesos no negativos en las aristas.

Definición: Sea G un grafo con peso, y sea H un subgrafo de G. El costo total o peso total de H es:

$$\omega(H) = \sum_{e \in H} \omega(e)$$

8.5 Árboles

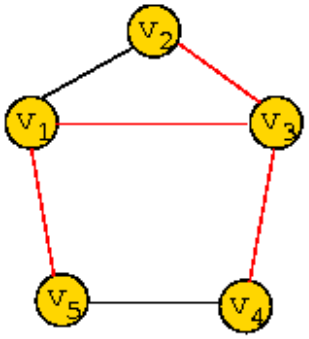
En esta parte del curso vamos estudiar un caso particular de la Teoría de Grafos que son los *árboles*. Primero, vamos definir el concepto de árbol.

Simplemente, un **árbol** es un grafo *conexo* sin ciclos. Notar que como un grafo requiere contener un mínimo de un vértice, un árbol contiene un mínimo de un vértice. También tenemos que un árbol es un grafo simple, puesto arcos y aristas paralelas forman ciclos. Un grafo no conexo que no contiene ciclos será denominado **bosque**. En ese caso tenemos un grafo donde cada componente es un árbol.

Árboles cobertores

Dado un grafo G no dirigido, conexo, se dice que un subgrafo *T* de *G* es un árbol cobertor si es un árbol y contiene el mismo conjunto de nodos que *G*.

Un arbol cobertor (en rojo)



Todo recorrido de un grafo conexo genera un árbol cobertor, consistente del conjunto de los arcos utilizados para llegar por primera vez a cada nodo.

Para un grafo dado pueden existir muchos árboles cobertores. Si introducimos un concepto de "peso" (o "costo") sobre los arcos, es interesante tratar de encontrar un árbol cobertor que tenga costo mínimo.

8.6 Árbol Cobertor Mínimo

En general estamos interesados en el caso en que $H = T$ es un árbol de cobertura de G. Ya que G es finito, la función $w(t)$ alcanza su mínimo en algún árbol T. Nos interesa hallar algún árbol T que minimice el costo total; dicho grafo no tiene por qué se único.

En esta sección veremos dos algoritmos para encontrar un árbol cobertor mínimo para un grafo no dirigido dado, conexo y con costos asociados a los arcos. El costo de un árbol es la suma de los costos de sus arcos.

8.6.1 Algoritmo de Kruskal

Este es un algoritmo del tipo "avaro" ("*greedy*"). Comienza inicialmente con un grafo que contiene sólo los nodos del grafo original, sin arcos. Luego, en cada iteración, se agrega al grafo el arco más barato que no introduzca un ciclo. El proceso termina cuando el grafo está completamente conectado.

En general, la estrategia "avara" no garantiza que se encuentre un óptimo global, porque es un método "miope", que sólo optimiza las decisiones de corto plazo. Por otra parte, a menudo este tipo de métodos proveen buenas heurísticas, que se acercan al óptimo global.

En este caso, afortunadamente, se puede demostrar que el método "avaro" logra siempre encontrar el óptimo global, por lo cual un árbol cobertor encontrado por esta vía está garantizado que es un árbol cobertor mínimo.

Una forma de ver este algoritmo es diciendo que al principio cada nodo constituye su propia componente conexa, aislado de todos los demás nodos.

Durante el proceso de construcción del árbol, se agrega un arco sólo si sus dos extremos se encuentran en componentes conexas distintas, y luego de agregarlo esas dos componentes conexas se fusionan en una sola.

Descripción del Algoritmo

Entrada: Un grafo G con costos, conexo.

Idea: Mantener un subgrafo acíclico cobertor H de modo que en cada paso exista al menos un árbol de cobertura de costo mínimo T, tal que H sea subgrafo de T. Considérense las aristas de G ordenadas por costo en forma no decreciente, los empates se rompen arbitrariamente.

Inicialización: $E(H) = \emptyset$

Iteración: Si la siguiente arista e de G (en el orden dado) une dos componentes de H, entonces agregamos e a $E(H)$. Si no, la ignoramos.

Terminación: Termínese el algoritmo cuando H sea conexo o cuando se acaben las aristas de G.

función Kruskal ($G = <N, A>$: *grafo*; longitud: $A \rightarrow R^+$): *conjunto de aristas*
{Iniciación}
Ordenar A por longitudes crecientes
$n \leftarrow$ el número de nodos que hay en N
$T \leftarrow \emptyset$ {contendrá las aristas del árbol de recubrimiento mínimo}
Iniciar n conjuntos, cada uno de los cuales contiene un elemento distinto de N
{bucle greedy }
repetir
e $\leftarrow$ $\{u, v\} \leftarrow$ arista más corta aun no considerada
$compu \leftarrow buscar(u)$
$compv \leftarrow buscar(v)$
si $compu \neq compv$ **entonces**
$fusionar(compu, compv)$
$T \leftarrow T \cup \{e\}$
hasta que T contenga n - 1 aristas
devolver T

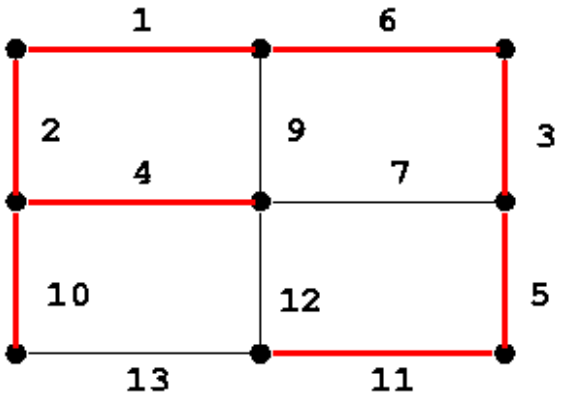
Complejidad del Algoritmo

El algoritmo requiere que las $|E|$ aristas estén ordenadas previamente. Esto toma $O(|E| \ log(|E|))$ operaciones.

El proceso de examinar las $|E|$ aristas toma a lo más $O(|E|)$ iteraciones; de ellas, $n-1$ requieren actualizar la lista de componentes a las que pertenecen los n vértices. Esto puede ser hecho en un total de $O(n log_2 n)$ operaciones, o incluso en forma más astuta en $O(n\alpha(n))$, donde:

$$\alpha(n) \leq \min\left\{ i : \underbrace{\log_2(\log_2(\ldots(\log_2(n))\ldots))}_{i \text{ veces}} \right\} = 1.$$

Ejemplo:



8.6.2 Algoritmo de *Prim*

La característica principal del algoritmo de *Kruskal* es que éste selecciona las mejores aristas sin preocuparse de las conexiones con las aristas selecionadas antes. El resultado es una proliferación de árboles que eventualmente se juntan para formar un árbol.

Ya que sabemos que al final tenemos que producir un solo árbol, por qué no intentar hacer como que el árbol crezca naturalmente hasta la obtención de un árbol generador mínimo? Así, la próxima arista seleccionada sería siempre una que se conecta al árbol que ya existe.

Esa es la idea del **algoritmo de *Prim***. Se caracteriza por hacer la selección en forma local, partiendo de un nodo seleccionado y construyendo el árbol en forma ordenada.

Dado que el arco es seleccionado de aquellos que parten del árbol ya construído, la viabilidad está asegurada. También se puede demostrar que el algoritmo de *Prim* es **correcto**, es decir que devuelve un árbol de recubrimiento minimal en todos los casos.

Al inicio el conjunto B contiene un vértice arbitrario. En cada paso, el algoritmo considera todas las aristas que tocan a B y selecciona a la de menor peso. Después, el algoritmo aumenta en B el vértice ligado por esa arista que no estaba en B. El proceso continúa hasta que B contenga a todos los vértices de G.

Una descripción del algoritmo se describe a continuación:

función *Prim*$(G = (N, A)$: *grafo*$)$: conjunto de aristas
$T := \{\}$
$B :=$ Un vértice de G
Mientras B no contenga todos los vértices
$(u,v) :=$ arista de menor peso tal que $u \in V - B$ e $v \in B$
$T := T \cup \{(u,v)\}$
$B := B \cup \{u\}$
Retornar T

Complejidad del Algoritmo

Para implementar este algoritmo eficientemente, podemos mantener una tabla donde, para cada nodo de V-A, almacenamos el costo del arco más barato que lo conecta al conjunto A. Estos costos pueden cambiar en cada iteración.

Si se organiza la tabla como una cola de prioridad, el tiempo total es $O(m \log n)$. Si se deja la tabla desordenada y se busca linealmente en cada iteración, el costo es $O(n^2)$. Esto último es mejor que lo anterior si el grafo es denso, pero no si está cerca de ser un grafo completo.

8.7 Distancias Mínimas en un Grafo Dirigido

Dado un grafo (o digrafo) ponderado y dos vértices ***s*** y ***t*** se quiere hallar ***d*****(*s*,*t*)** y el **camino** con dicha longitud. Los primeros algoritmos que presentamos obtienen todos los caminos de longitud mínima desde un vértice dado ***s*** al resto de vértices del grafo. El último algoritmo resuelve el problema para un par cualquiera de vértices de G.

Si el vértice ***u*** se encuentra en un camino C de longitud mínima entre los vértices ***s*** y z entonces, la parte de C comprendida entre los vértices ***s*** y ***u*** es un camino de longitud mínima entre ***s*** y ***u***. Por tanto, el conjunto de caminos mínimos desde ***s*** a los restantes vértices del grafo G es un árbol, llamado el **árbol de caminos mínimos** desde ***s***.

Definición: En un grafo con pesos, la ***distancia*** entre dos vértices es:

$$d(u,v) = \min\{\, w(P) : P \text{ es un camino que une } u \text{ y } v \,\}.$$

Note que en particular, esto coincide con la definición usual de distancia, si consideramos la función de peso w donde $w(e)= 1$ para todo $e \in E(G)$.

Nos interesa hallar la distancia más corta entre dos vértices dados u y v, o bien hallar todas las distancias más cortas entre un vértice dado u y todos los otros vértices de G.

En general, no es mucho más eficiente hallar la ruta más corta entre u y v que hallar todas las rutas más cortas entre u y los otros vértices. Para esto usamos el *algoritmo de Dijkstra*.

8.7.1 Algoritmo de Dijkstra

La idea básica del algoritmo es la siguiente: Si P es un camino de longitud mínima $s \to z$ y P contiene al vértice v, entonces la parte $s \to v$ de P es también camino de longitud mínima de s a v. Esto sugiere que si deseamos determinar el camino óptimo de s a cada vértice z de G, podremos hacerlo en orden creciente de la distancia $d(s,z)$.

Descripción del algoritmo

Entrada: Un grafo (o digrafo) G con pesos no negativos, y un vértice de partida u. El peso de una arista xy es $w(xy)$; si xy no es una arista diremos que $w(xy) = \infty$.

Idea: Mantener un conjunto S de vértices a los que conocemos la distancia más corta desde u, junto con el predecesor de cada vértice de S en dicha ruta. Para lograr esto, mantenemos una distancia tentativa $t(z)$ desde u a cada $z \in V(G)$. Si $z \notin S$, $t(z)$ es la distancia más corta encontrada hasta ahora entre u y z. Si $z \in S$, $t(z)$ es la distancia más corta entre u y z. En cualquier caso, $pred[z]$ es el predecesor de z la ruta $u \to z$ en cuestión.

Inicialización: $S = \{u\}$, $t(u)=0$, $t(z) = w(uz)$ y $pred[z] = u$ para $z \neq u$.

Iteración: Sea $v \in V(G) - S$, tal que:

$$t(z) = \min\{t(v) : v \in V(G) - S\}$$

Agrégase v a S.

Para $z \in V(G) - S$ actualícese $t(z) = \min\{t(z), t(v)+w(vz)\}$.

Si cambia $t(z)$, cámbiese $pred[z]$ a v.

Terminación: Termínese el algoritmo cuando $S = V(G)$ o cuando $t(z) = \infty$ para todo $z \in V(G) - S$.

Al terminar, la distancia más corta entre u y v está dada por: $d(u,v) = t(v)$.

Análisis de la complejidad

En cada iteración se añade un vértice a T, luego el número de iteraciones es n. En cada una se elige una etiqueta mínima, la primera vez entre n-1, la segunda entre n-2, ..., luego la complejidad total de estas elecciones es $O(n^2)$. Por otra parte cada arista da lugar a una actualización de etiqueta, que se puede hacer en tiempo constante $O(1)$, en total pues $O(q)$. Por tanto la complejidad total del algoritmo es $O(n^2)$.

8.7.2 Algoritmo de Ford

Es una variante del algoritmo de *Dijkstra* que admite la asignación de pesos negativos en los arcos, aunque no permite la existencia en el digrafo de ciclos de peso negativo.

Descripción del algoritmo

Entrada: Un digrafo ponderado con pesos no negativos en los arcos, un vértice s∈V. El **peso del arco** uv se indica por $w(uv)$, poniendo $w(uv) = \infty$ si uv no es arco.

Salida: La distancia desde ***s*** a cada vértice del grafo.

Clave: Mejorar en cada paso las etiquetas de los vértices, $t(u)$.

Inicialización: Sea $T=\{s\}$, $t(s)=d(s,s)=0$, $t(z)=\infty$ para $z \neq s$.

Iteración: Mientras existan arcos $e=xz$ para los que $t(z) > t(x) + w(e)$ actualizar la etiqueta de z a $min\{t(z), t(x)+w(xz)\}$.

Análisis de la complejidad

En primer lugar debemos observar que cada arco puede considerarse varias veces. Empecemos ordenando los arcos del digrafo D siendo este el orden en que se considerarán los arcos en el algoritmo. Después de la primera pasada se repite el proceso hasta que en una pasada completa no se produzca ningún cambio de etiquetas. Si D no contiene ciclos negativos puede demostrarse que, si el camino mínimo $s \rightarrow u$ contiene k arcos entonces, después de k pasadas se alcanza la etiqueta definitiva para u. Como $k \leq n$ y el número de arcos es q, resulta que la complejidad del

algoritmo de *Ford* es $O(qn)$. Además podemos detectar un ciclo negativo si se produce una mejora en las etiquetas en la pasada número n.

8.7.3 Algoritmo de Floyd-Warshall

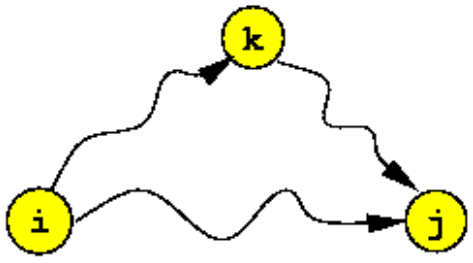
A veces no es suficiente calcular las distancias con respecto a un vértice s, si no que necesitamos conocer la **distancia** entre cada par de vértices. Para ello se puede aplicar reiteradamente alguno de los algoritmos anteriores, variando el vértice s de partida. Así tendríamos algoritmos de complejidad $O(n^3)$ (si usamos el algoritmo de *Dijkstra*) u $O(n^2q)$ (si usamos el algoritmo de *Ford*).

A continuación se describe un algoritmo, debido a *Floyd* y *Warshall*, con una estructura sencilla, que permite la presencia de arcos de peso negativo y que resuelve el mismo problema. (Naturalmente los ciclos de peso negativo siguen estando prohibidos).

La idea básica del algoritmo es la construcción de una sucesión de matrices W^0, W^1, ..., W^n, donde el elemento ij de la matriz W^k nos indique la longitud del camino mínimo $i \rightarrow j$ utilizando como vértices interiores del camino los del conjunto $\{v_1, v_2, ..., v_k\}$. La matriz W^0 es la matriz de pesos del digrafo, con $w^0_{ij} = w(ij)$ si existe el arco $i \rightarrow j$, $w^0_{ii} = 0$ y $w^0_{ij} = \infty$ si no existe el arco $i \rightarrow j$.

Para aplicar este algoritmo, los nodos se numeran arbitrariamente 1,2,...n. Al comenzar la iteración k-ésima se supone que una matriz $D[i,j]$ contiene la distancia mínima entre i y j medida a través de caminos que pasen sólo por nodos intermedios de numeración $<k$.

Estas distancias se comparan con las que se obtendrían si se pasara una vez por el nodo k, y si de esta manera se obtendría un camino más corto entonces se prefiere este nuevo camino, de lo contrario nos quedamos con el nodo antiguo.



Al terminar esta iteración, las distancias calculadas ahora incluyen la posibilidad de pasar por nodos intermedios de numeración $<=k$, con lo cual estamos listos para ir a la iteración siguiente.

Para inicializar la matriz de distancias, se utilizan las distancias obtenidas a través de un arco directo entre los pares de nodos (o infinito si no existe tal arco). La distancia inicial entre un nodo y sí mismo es cero.

Descripción del algoritmo

***Entrada*:** Un digrafo ponderado sin ciclos de peso negativos. El peso del arco uv se indica por $w(uv)$, poniendo $w(uv)=\infty$ si uv no es arco.

Salida: La distancia entre dos vértices cualesquiera del grafo.

Clave: Construimos la matriz W^k a partir de la matriz W^{k-1} observando que $w_{ij}^{k} = \min\{ w_{ij}^{k-1}, w_{ik}^{k-1} * w_{kj}^{k-1} \}$

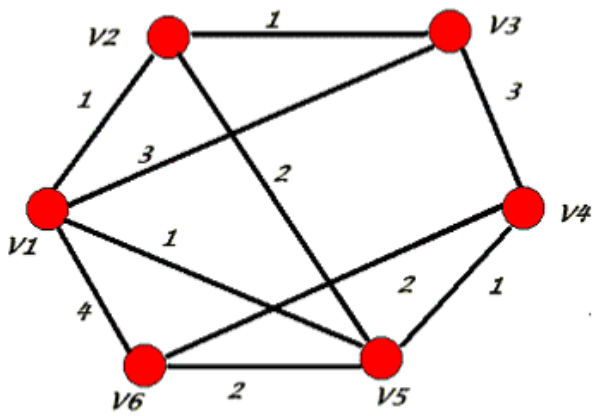
Iteración: Para cada $k=1,\ldots,n$, hacer $w_{ij}^{k} = \min\{ w_{ij}^{k-1}, w_{ik}^{k-1} * w_{kj}^{k-1} \}$ $\forall i,j=1,\ldots,n$
El elemento ij de la matriz W^n nos da la longitud de un camino mínimo entre los vértices i y j.

Análisis de la complejidad

Se deben construir n matrices de tamaño nxn y cada elemento se halla en tiempo constante. Por tanto, la complejidad del algoritmo es $O(n^3)$

El algoritmo de *Floyd-Warshall* es mucho más eficiente desde el punto de vista de almacenamiento dado que puede ser implementado una vez atualizado la distancia de la matriz con cada elección en k; no hay ninguna necesidad de almacenar matrices diferentes. En muchas aplicaciones específicas, es más rápido que cualquier versión de algoritmo de *Dijkstra*.

Ejemplo:



Valores Iniciales de $d(i,j)$

	V1	***V2***	***V3***	***V4***	***V5***	***V6***
V1	0	1	3	X	1	4
V2	1	0	1	X	2	X
V3	3	1	0	3	X	X
V4	X	X	3	0	1	2
V5	1	2	X	1	0	2
V6	4	X	X	2	2	0

Obs: X significa que un vértice cualquiera no tiene ligazón directa con otro.

Menor Camino

	V1	***V2***	***V3***	***V4***	***V5***	***V6***
V1	0	1	2	2	1	3
V2	1	0	1	3	2	4
V3	2	1	0	3	3	5
V4	2	3	3	0	1	2
V5	1	2	3	1	0	2
V6	3	4	5	2	2	0

Capítulo 9

Complejidad Computacional

9.1 Introducción

La complejidad computacional considera globalmente todos los posibles algoritmos para resolver un problema dado.

Estamos interesados en la distinción que existe entre los problemas que pueden ser resueltos por un algoritmo en tiempo polinómico y los problemas para los cuales no conocemos ningún algoritmo polinómico, es decir, el mejor es no-polinómico.

La teoría de la *NP*-Completitud no proporciona un método para obtener algoritmos de tiempo polinómico, ni dice que que estos algoritmos no existan. Lo que muestra es que muchos de los problemas para los cuales no conocemos algoritmos polinómicos están computacionalmente relacionados.

9.2 Algoritmos y Complejidad

En los capítulos anteriores, hemos visto que los algoritmos cuya complejidad es descrita por una función polinomial pueden ser ejecutados para entradas grandes en una cantidad de tiempo razonable, mientras que los algoritmos exponenciales son de poca utilidad excepto para entradas pequeñas.

En esta sección se tratarán los problemas cuya complejidad es descrita por funciones exponenciales, problemas para los cuales el mejor algoritmo conocido requeriría de muchos años o centurias de tiempo de cálculo para entradas moderadamente grandes.

De esta forma se presentarán definiciones que pretenden distinguir entre los **problemas tratables** (aquellos que no son tan duros) y los **problemas intratables** (duros o que consumen mucho tiempo). La mayoría de estos problemas ocurren como problemas de optimización combinatoria.

9.3 Problemas *NP* Completos

Definición Formal de la Clase *NP*

La definición formal de *NP*-completo usa reducciones, o transformaciones, de un problema a otro.

Una definición formal de ***NP*-completo** es:

Un problema ***P*** *es* ***NP*-completo** *si éste está en* ***NP*** *y para cada otro problema* ***P'*** *en* ***NP***, ***P'*** **µ** ***P***

Del teorema anterior y la definición de ***NP* completo** se deduce el siguiente teorema:

Si cualquier problema ***NP*-completo** *esta en P, entonces* ***P=NP***

Este teorema indica, por un lado, que tan valioso sería encontrar un algoritmo polinomialmente acotado para cualquier problema *NP*-completo y, por otro, que tan improbable es que tal algoritmo exista pues hay muchos problemas en *NP* para los cuales han sido buscados algoritmos polinomialmente acotados sin ningún éxito.

Como se señaló antes, el primer problema de decisión que se propuso como un problema *NP*-completo es el **satisfactibilidad** de la lógica proposicional. Todos los problemas *NP* para los cuales se puede comprobar que pueden ser reducibles al problema de satisfctibilidad, son *NP*-completos.

Así tenemos que los siguientes problemas son NP-completos: **rutas** y **circuitos de *Hamilton***, asignación de trabajos con penalizaciones, el agente viajero, el problema de la mochila.

Actualmente para probar que un problema es *NP*-completo, es suficiente con probar que algún otro problema *NP*-completo es polinomialmente reducible a éste debido a que la relación de reducibilidad es transitiva. La lógica de lo dicho es la siguiente:

- Si ***P'*** es ***NP*-completo**, entonces todos los problemas en ***NP*** **µ** ***P'***
- Si se demuestra que ***P'*** **µ** ***P***
- Entonces todos los problemas en ***NP*** **µ** ***P***
- Por lo tanto, ***P*** es ***NP*-completo**

Supóngase un problema *P*, *P** es el **complemento** del problema *P* si el conjunto de símbolos en la cadena s de la fase de adivinación del algoritmo no determinístico que codifican las instancias si de *P** son exactamente aquellas que no están codificando las instancias si de *P*.

Por ejemplo, el problema del ***circuito Hamiltoniano*** es: dado un grafo $G=(V, E)$, es *G* Hamiltoniano. Y el complemento del problema del circuito Hamiltoniano es: dado un grafo $G=(V, E)$, es *G* no Hamiltoniano.

Supóngase un problema *P* en *P*, entonces hay un algoritmo acotado polinomialmente que resuelve *P*. Un algoritmo acotado polinomialmente que solucione el complemento de *P* es exactamente el mismo algoritmo,

únicamente con la sustitución de no por si cuando se obtiene una solución afirmativa y viceversa. El siguiente teorema expresa lo mencionado

*Si P es un problema en P, entonces el complemento P * de P está también en P.*

Los mismos argumentos no pueden ser aplicados para demostrar que un problema en *NP* está también en *NP*. Esto motiva la siguiente definición:
La clase co. *NP* es la clase de todos los problemas que son complemento de problemas en *NP*.

Decir que el complemento de todos los problemas en *NP* está también en *NP* es equivalente a decir que ***NP* = co-NP**. Sin embargo, hay razones para creer que ***NP*¹ co-*NP***. La evidencia es tan circunstancial como la que se estableció para la conjetura de que ***P*¹ *NP***: Muchos investigadores han tratado por largo tiempo, sin éxito, de construir pruebas sucintas para el complemento del problema del circuito *Hamiltoniano*, así como para muchos otros problemas. Sin embargo, puede ser mostrado (como con ***P*¹ *NP***) que si la conjetura es verdadera, está en los problemas ***NP*-completo** testificar su validez.

Si el complemento de un problema *NP-completo* está en NP, entonces ***NP* = co-*NP***.

Así de todos los problemas en NP, Los problemas ***NP*-completo** son aquellos con complementos de estar en NP. Contrariamente, si el complemento de un problema en *NP* está también en *NP*, esto es evidencia de que el problema no está en NP.

Cuando todos los problemas en *NP* se transforman polinomialmente al *problema P*, pero no es posible demostrar $P \subseteq NP$, se dice que *P* es tan difícil como cualquier problema en *NP* y por lo tanto ***NP-hard***.

NP-hard es usado también en la literatura para referirse a los problemas de optimización de los cuales su problema de decisión asociado es *NP*-completo
Problemas que pueden ser solucionados por algoritmos cuyas operaciones pueden ser confinadas dentro de una cantidad de espacio que está acotado por un polinomio del tamaño de la entrada, pertenecen a la clase ***PSPACE***. ***P***, ***NP*** y **co-*NP*** son subconjuntos de ***PSPACE***.

Se dice que un problema es ***PSPACE*-completo** si éste está en *PSPACE* y todos los otros problemas en *PSPACE* son polinomialmente reducibles a él.

9.4 Problemas Intratables

Garey & Johnson, plantean una situación de intratabilidad interesante que describimos a través del siguiente caso:

Si trabajas para una compañía que está pensando entrar a una nueva era globalizadora de gobierno y tu jefe te propone que obtengas un método para determinar si o no cualquier conjunto dado de especificaciones para un nuevo componente pueden ser satisfechas de alguna manera y, si es así, se deberá construir el diseño que satisfaga dichas expecificaciones. La realidad es que después de un tiempo comprobarás que no hay manera de obtener tal método. De modo que tendrás dos alternativas:

1. Renunciar para evitar la pena de que te corran por inepto.

2. Esperar a que te corran por inepto. Sería muy penoso, pero tendrías que confesar que no pudiste resolver el problema por falta de capacidad.

Para evitar la situación anterior tú podrías tratar de tomar una actitud de una persona más valiente y demostrar que en realidad dicho método no se puede obtener. Si eso fuera así entonces tú podrías ir con tu jefe y aclararle que el problema no lo resolviste porque no se puede resolver: Es intratable.

Sin embargo, es muy probable que a tu jefe esta característica del problema le tenga sin cuidado y, de todas maneras, te corran o tengas que renunciar. Es decir regresarás a las alternativas: 1 y 2 antes citadas.

Una situación más prometedora, es tratar de probar que el problema no solo es tan difícil que no se puede resolver, sino que además pertenece a una clase de problemas tales que cientos y miles de personas famosas e inteligentes, tampoco pudieron resolver. Aún así, no estamos muy seguros aún de tu permanencia en la compañía!.

En realidad el conocimiento de que un problema es intratable es de gran utilidad en problemas de tipo computacional. Dado que el problema completo es intratable, tu podrías resolver el problema para ciertas instancias del mismo, o bien, resolver un problema menos ambicioso. Por ejemplo, podrá conformarte con obtener un método para las especificaciones de solo cierto tipo de componentes para determinados conjuntos de entrada y no para "cualquier conjunto dado de especificaciones". Esto seguramente te llevará a tener mejor relaciones con tu compañía.

En general, nos interesa encontrar el algoritmo más eficiente para resolver un problema determinado. Desde ese punto de vista tendríamos que considerar todos los recursos utilizados de manera que el algoritmo más eficiente sería aquél que consumiera menos recursos. ¿Cuáles recursos? Si pensamos nuevamente que tu jefe te encarga que obtengas el algoritmo más eficiente, es decir el que consume menos recursos so pena de que si no lo

obtienes seas eliminado de la nomina de la compañía, estoy seguro, en que pensarías en todos los recursos posibles (horas-hombre para construir el algoritmo, tiempo de sintonización, tiempo requerido para obtener una solución, número de procesadores empleados, cantidad de memoria requerida, etc.). Para simplificar un poco la tarea, consideraremos que tu jefe te permite que la eficiencia la midas sobre un solo recurso: El tiempo de ejecución. Pensemos por un momento que solo se tiene una sola máquina de modo que el número de procesadores no requiere de un análisis. ¿Qué le pasó a tu jefe? ¿De pronto se volvió comprensivo contigo? En realidad los otros parámetros los ha tomado en cuenta como costo directo del programa en cuestión. Esto es en realidad lo que se hace en teoría de complejidad. Para el análisis de complejidad, se considera por lo general la eficiencia sobre el tiempo, para un solo procesador en cómputo secuencial; para cómputo paralelo se considera también el número de procesadores.

Los algorimos han sido divididos como buenos o malos algoritmos. La comunidad computacional acepta que un buen algoritmo es aquél para el cual existe un algoritmo polinomial determinístico que lo resuelva. También se acepta que un mal algoritmo es aquel para el cual dicho algoritmo simplemente no existe. Un problema se dice intratable, si es muy difícil que un algoritmo de tiempo no polinomial lo resuelva. No obstante, esta clasificación de algoritmos en buenos y malos puede resultar a veces engañosa, ya que se podría pensar que los algoritmos exponenciales no son de utilidad práctica y que habrá que utilizar solamente algoritmos polinomiales. Sin embargo se tiene el caso de los *métodos simplex y Branch and Bound*, los cuales son muy eficientes para muchos problemas prácticos. Desafortunadamente ejemplos como estos dos son raros, de modo que es preferible seguir empleando como regla de clasificación en buenos y malos algoritmos, la que lo hace dependiendo de si son o no polinomiales; todo esto con la prudencia necesaria.

La teoría de la *NP-Completés* fué presentada inicialmente por *Cook* desde 1971. *Cook* probó que un problema particular que se estudia en Lógica (no necesitamos ahora estudiarlo) y que se llama "El **Problema de Satisfactibilidad**", tenía la propiedad de que cualquier problema que perteneciera a la **clase *NP***, podía ser reducido a él a través de una transformación de tipo polinomial. Esto significaba que si el problema de satisfactibilidad podía ser resuelto en tiempo polinomial, todos los problemas No Polinomiales también podrían resolverse en tiempo polinomial, lo que significaría que la clase ***NP*** dejaría de existir!!.

De esta forma, si cualquier problema en ***NP*** es intratable, entonces satisfactibilidad es un problema intratable. *Cook* también sugirió que el problema de satisfactibilidad y otros problemas ***NP*** tenían la característica de ser los problemas más duros. Estos problemas tienen dos versiones: de decisión y de optimización. El conjunto de estos problemas de optimización se denominan ***NP Completos***, mientras que a sus correspondientes problemas de optimización, reciben el nombre de ***NP Hard***.

9.5 Problemas de Decisión

Aquí hemos hablado de problemas de optimización como aquél donde buscamos el máximo o el mínimo de una función donde existen o no un conjunto de restricciones. Sin embargo, un problema de optimización combinatoria puede también ser formulado de manera más relajada como sigue:

Dado un problema de optimización, podemos encontrar el costo de la solución óptima.

Se sabe que dado un problema de optimización, se puede definir un problema de decisión asociado a él, esto es, una pregunta que puede ser contestada por si o no. Por otro lado, varios problemas computacionales bien conocidos son problemas de decisión. Entre los problemas de decisión se pueden mencionar por ejemplo:

- El problema de paro (***Halting Problem***): dado un algoritmo y su entrada, ¿Parará éste alguna vez?

- El problema de satisfactibilidad: dada una fórmula *booleana*, ¿Es ésta satisfactible?

- El problema del circuito *Hamiltoniano*: dado un grafo *G*, ¿Hay un circuito en *G* que visite todos los nodos exactamente una vez?.

La definición de problema de decisión a partir del problema de optimización permite estudiar ambos tipos de problemas de una manera uniforme. Además, como se ha puntualizado que un problema de decisión no es más difícil que el problema de optimización original, cualquier resultado negativo probado sobre la complejidad del problema de decisión será aplicable al problema de optimización también.

Se está interesado en clasificar los problemas de decisión de acuerdo a su complejidad. Se denota por ***P*** a la clase de problemas de decisión que son polinomialmente acotados, esto es, la clase de problemas de decisión que pueden ser solucionados en tiempo polinomial. La **clase *P*** puede ser definida muy precisamente en términos de cualquier formalismo matemático para algoritmos, como por ejemplo la **Máquina de *Turing***.

Se puede decir que ***P*** es la clase de problemas de decisión relativamente fáciles, para los cuales existe un algoritmo que los soluciona eficientemente. Para una entrada dada, una "solución" es un objeto (por ejemplo, un grafo coloreado) que satisface el criterio en el problema y justifica una respuesta afirmativa de si. Una "solución propuesta" es simplemente un objeto del tipo apropiado, este puede o no satisfacer el criterio. Informalmente se puede decir que ***NP*** es la clase de problemas de decisión para los cuales una solución propuesta dada para una entrada dada, puede ser chequeada

rápidamente (en tiempo polinomial) para ver si ésta es realmente una solución, es decir, si ésta satisface todos los requerimientos del problema.

Una solución propuesta puede ser descrita por una cadena de símbolos a partir de algún conjunto finito. Simplemente se necesita alguna convención para describir grafos, conjuntos, funciones, etc. usando estos símbolos. El tamaño de la cadena es el número de símbolos en ella. Chequear una solución propuesta incluye chequear que la cadena tenga sentido (esto es, que tenga una sintáxis correcta) como una descripción del tipo de objeto requerido, también como chequear que ésta satisface el criterio del problema.

9.6 Algoritmos No Determinísticos

Un algoritmo no determinístico tiene dos fases:

Fase no Determinística: alguna cadena de caracteres, ***s***, completamente arbitraria es escrita a partir de algún lugar de memoria designado. Cada vez que el algoritmo corre, la cadena escrita puede diferir (Esta cadena puede ser vista como una adivinación de la solución para el problema, por lo que a esta fase se le da el nombre de fase de adivinación, pero ***s*** también podría ser ininteligible o sin sentido).

Fase Determinística: Un algoritmo determinístico (es decir ordinario) siendo ejecutado. Además de la entrada del problema de decisión, el algoritmo puede leer ***s***, o puede ignorarla. Eventualmente éste para con una salida de si o no, o puede entrar en un ciclo infinito y nunca parar (véase ésta como la fase de chequear, el algoritmo determinístico verifica ***s*** para ver si ésta es una solución para la entrada del problema de decisión).

El número de pasos llevados a cabo durante la ejecución de un algoritmo no determinístico es definido como la suma de los pasos en ambas fases; esto es, el número de pasos tomados para escribir ***s*** (simplemente el número de caracteres en ***s***) más el número de pasos ejecutados por la segunda fase determinística.

Normalmente cada vez que se corre un algoritmo con la misma entrada se obtiene la misma salida. Esto no ocurre con algoritmos no determinísticos; para una entrada particular x, la salida a partir de una corrida puede diferir de la salida de otra debido a que ésta depende de ***s***. aunque los algoritmos no determinísticos no son realistas (algoritmos útiles en la práctica), ellos son útiles para clasificar problemas.

Se dice que un algoritmo no determinístico es polinomialmente acotado si hay un polinomio p tal que para cada entrada de tamaño n para la cual la respuesta es si, hay alguna ejecución del algoritmo que produce una salida si en cuando mucho $p(n)$ pasos.

De esta forma se puede decir que: *NP es la clase de problemas de decisión para los cuales hay un algoritmo no determinístico acotado polinomialmente (el nombre de NP viene de no determinístico polinomialmente acotado).*

Un algoritmo ordinario (determinístico) para un problema de decisión es un caso especial de un algoritmo no determinístico. En otras palabras, si *A* es un algoritmo determinístico para un problema de decisión, entonces *A* es la segunda fase de un algoritmo no determinístico. *A* simplemente ignora lo que fue escrito por la primera fase y procede con su cálculo usual. Un algoritmo no determinístico puede hacer cero pasos en la primera fase (escribiendo la cadena nula). Así, si *A* corre en tiempo polinomial, el algoritmo no determinístico con *A* como su segunda fase corre también en tiempo polinomial. De lo mencionado se deduce que *P* es un subconjunto propio de *NP*.

La gran pregunta es ¿ ***P = NP*** o es ***P*** un subconjunto propio de ***NP***? Se cree que ***NP*** es un conjunto mucho mayor que ***P***, pero no hay un solo problema en ***NP*** para el cual éste haya sido probado que el problema no está en ***P***. No hay un algoritmo polinomialmente acotado conocido para muchos problemas en ***NP***, pero ningún límite inferior mayor que un polinomio ha sido probado para estos problemas.

NP-completo es el término usado para describir los problemas de decisión que son los más difíciles en ***NP*** en el sentido que, si hubiera un algoritmo polinomialmente acotado para un problema ***NP*-completo**, entonces habría un algoritmo polinomialmente acotado para cada problema en ***NP***.

Bibliografía

1. AHO, Alfred H., Hopcroft, John E. y Ullman, Jeffrey D. *Estructuras de Datos y Algoritmos*, Addison-Wesley Iberoamericana, 1988.

2. BAEZA-YATES, Ricardo. *Algoritmia*, Depto. Ciencias de la Computación, Universidad de Chile, 1991.

3. BAASE, S.; Van Gelder, A. *Computer Algorithms. Introduction to Design and Analysis*. Addison-Wesley, 2000.

4. BALCAZAR, José L. *Apuntes sobre el Cálculo de la Eficiencia de los Algoritmos*, Universidad Politécnica de Cataluña, España.

5. BRASSARD, G., Bratley, P., *Fundamentals of Algorithmics*. Prentice Hall, 1995.

6. CORMEN Thomas, Leiserson Charles and Rivest Ronald. *Introduction to Algorithms*. The MIT Press, 1990.

7. GIMÉNEZ Cánovas, Domingo. *Apuntes y Problemas de Algorítmica*. Facultad de Informática. Universidad de Murcia, 2001.

8. GONNET, G. H., Baeza-Yates, R. Handbook of Algorithms and Data Structures, Addison-Wesley, 1991.

9. HOROWITZ, E.; Sahni, S. *Fundamentals of Data Structures*. Computer Science Press, 1976.

10. KNUTH, Donald E. "*Fundamental algorithms*", volume 1 de "*The Art of Computer Programming*", segunda edición, Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

11. KNUTH, Donald E. "*Sorting and Searching*", volume 3 de "*The Art of Computer Programming*", segunda edición, Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

12. PARBERRY Ian. *Problems on Algorithms*. Prentice Hall, 1995.

13. PEÑA M., Ricardo. *Diseño de programas. Formalismo y abstracción*. Prentice-Hall, 1998.

14. WEISS, M. A., *Estructuras de datos y algoritmos*. Addison-Wesley Iberoamericana, 1995.

15. WIRTH, Niklaus. *Algoritmos y Estructuras de Datos*. Pentice Hall, 1987.