UAEM | Universidad Autónoma del Estado de México

CUTex
Centro Universitario UAEM Texcoco

INGENIERÍA EN COMPUTACIÓN

PROGRAMACIÓN AVANZADA

APUNTES

PERIODO 2015B.

Prof. Joel Ayala de la Vega

PROGRAMACIÓN AVANZADA.

Tabla de contenido

PRESENTACIÓN.

Este escrito fue hecho con la intención de apoyar a los estudiantes que cursan la Unidad de Aprendizaje (U. de A.) "**Programación Avanzada**" en el estudio de la modularidad, en la comprensión de funciones recursivas, de los algoritmos y sus diferentes paradigmas. Dentro del estudio de algoritmos, en cada capítulo del escrito muestra el pseudocódigo o el código en el lenguaje C y un ejemplo de la ejecución del algoritmo, permitiendo la comprensión del mismo.
El escrito se basa por completo en el temario de "Programación Avanzada" revisado en Mayo del 2011.

En la primer parte se explica la importancia de la programación modular, que consiste en dividir un programa muy grande en un determinado número de módulos en forma inteligente, y se da un enfoque desde el punto de vista de cohesión y acoplamiento. La segunda parte realiza una revisión de la programación recursiva.

Por el otro lado se tiene la complejidad algorítmica, esta complejidad no trata sobre el número de línea, más bien, trata sobre el tiempo que pueda un algoritmo en específico en ser ejecutado, en este sentido se puede tratar de una función de unas cuantas líneas, pero para poder llegar a un resultado se requieran de horas, días, meses, años e incluso siglos. (Puede ser un algoritmo de nueve líneas como el caso de las famosas torres de Hanoi o una representación matemática de una sola línea como lo muestra el algoritmo del agente viajero, ambos problemas intratables). Los algoritmos intratables tienen la característica de ser recursivos, por lo que se debe tener la idea de lo que es recursión tanto en la parte matemática como en el ambiente de la programación.

De esta forma, la tercera y cuarta parte del escrito permite explicar la "Introducción a la Algorítmica y Complejidad". En este punto se da una reseña histórica del nacimiento de los algoritmos y clasifica la forma de medir la complejidad del algoritmo.

La quinta parte da los primeros ejemplos del análisis mediante algoritmos de búsqueda y ordenamiento.

De la sexta a la décima parte del escrito responde a "Estrategias de diseño de algoritmos" bajo los paradigmas "Divide y vencerás", "insaciables", "Programación Dinámica", "Retorno sobre la misma ruta" y "Ramificación y acotamiento".
En el paradigma de programación dinámica se analiza un algoritmo recursivo "El agente Viajero", permitiendo hacer un "Análisis de algoritmo recursivo".
La onceava parte del escrito trata, en forma histórica, los conceptos de algoritmos NP. Cómo Hilbert planteó la matemática a inicios del siglo XX, los resultados obtenidos con Gödel, la definición de algoritmo por Turing y otros matemáticos llegando a comentar los algoritmos no determinísticos e intratabilidad (el Entscheidungsproblem).

Uno espera, con este escrito, dar un criterio al alumno de una clasificación de la complejidad de software y de los algoritmos. Además, observar la diferencia de lo que es un algoritmo tratable, un algoritmo intratable y los problemas no computables. Para estos últimos se explica "the halting problem", siendo éste un problema clásico de "Teoría de la Computación".

I PROGRAMACIÓN MODULAR

(Booch, 1991)
(Carlo Ghezzi, 1991)
(Deheza, 2005)

1.1 Programas y Sentencias

Un programa de computadora es un sistema con componentes e interrelaciones. Intentaremos determinar cuáles son dichos componentes y como se relacionan.

En primer lugar definiremos los conceptos de programa y programa de computadora.

Un *programa* puede ser definido como "una secuencia precisa y ordenada de instrucciones y grupos de instrucciones, las cuales, en su total, definen, describen, o caracterizan la realización de alguna tarea".

Un *programa de computadora* es simplemente un programa el cual, posiblemente a través de una transformación, indica a la computadora como realizar una tarea.

En el nivel más elemental, observamos que un programa de computadora está compuesto de sentencias o instrucciones. Estas instrucciones están ordenadas en una secuencia. Podemos por lo tanto identificar a las instrucciones como componentes y a la secuencia como una relación. Esta es la visión clásica o "algorítmica" de un programa.

De esta visión tenemos como consecuencia, que el esfuerzo en el desarrollo de un programa se enfatiza en encontrar un método de solución y su transcripción sentencia a sentencia.

Para los propósitos de nuestro estudio, consideraremos que una sentencia es una línea de código que el programador escribe.

1.2 La Lingüística de la Modularidad

Previo a la definición de módulo de programa haremos algunas observaciones. Supongamos un conjunto de sentencias como las que se representan a en la figura 1.1:

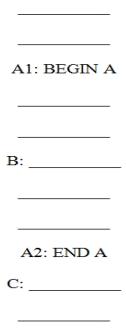


Fig. 1.1

Diremos que A1 y A2 son los límites del conjunto o agregado de sentencias llamado A. La sentencia B se encuentra dentro de A, y C se encuentra fuera de A.

Las sentencias se encuentran en el orden en que ingresaran a un compilador. Este orden es conocido como *orden lexicográfico* de un programa. Para nuestro estudio el término lexicográfico siempre significará "como está escrito" o el orden en que aparecen las sentencias de un programa en el listado de un compilador. Volviendo al ejemplo, diremos que la sentencia C está lexicográficamente después que la A2.

Es importante distinguir que el orden lexicográfico casi siempre no se corresponde con el orden de ejecución de las sentencias.

Uno de los propósitos de los elementos de límite (A1 y A2 en el ejemplo) es el de controlar el alcance en el que identificadores son definidos y asociados a objetos (variables).

Estamos ahora en condiciones de definir el término módulo de programa o simplemente módulo:

> *Un módulo es una secuencia lexicográficamente contigua de sentencias, encerrada entre elementos de frontera, y que poseen un identificador del conjunto de dichas sentencias.*

Dicho de otra manera, un módulo es un grupo de sentencias contiguas que poseen un identificador simple por el cual son referenciadas.

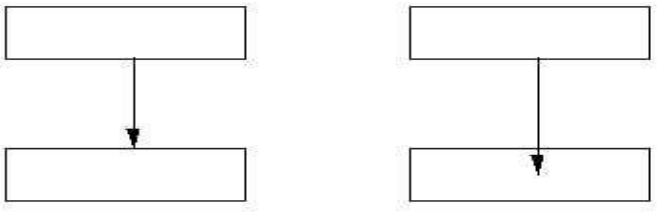
Esta definición es general y dentro de la misma podemos encontrar implementaciones particulares de lenguajes específicos como ser: "párrafos", "secciones", y "subprogramas" de COBOL, "funciones" de C, "procedimientos" de Pascal, etc.

Un lenguaje de programación incluye un determinado tipo de módulo, solo si implementa construcciones lingüísticas específicas que realizan las características de definición y activación de dichos módulos.

1.3 Conexiones Normales y Patológicas

Diremos que entre dos módulos existe una *conexión normal* cuando la conexión se produce al nivel del identificador del módulo invocado.

En oposición si la conexión intermódular se realiza a un identificador de un elemento interno del módulo invocado, diremos que es una *conexión patológica*.



1.4 Como alcanzar sistemas de mínimo costo

Cuando se trata con un problema de diseño, por ejemplo, un sistema que pueda ser desarrollado en un par de semanas, no se tienen mayores problemas, y el desarrollador puede tener todos los elementos del problema "en mente" a la vez. Sin embargo cuando se trabaja en proyectos de gran escala, es difícil que una sola persona sea capaz de llevar todas las tareas y tener en mente todos los elementos a la vez.

El diseño exitoso se basa en un viejo principio conocido desde los días de Julio Cesar: *Divide y conquistarás.*

Específicamente, diremos que el costo de *implementación* de un sistema de computadora podrá minimizarse cuando pueda separarse en partes

- *Manejablemente pequeñas*
- *Solucionables separadamente*.

Por supuesto, la interpretación de manejablemente pequeña varía de persona en persona. Por otro lado, muchos intentos de particionar sistemas en pequeñas partes arribaron incrementos en los tiempos de implementación. Esto se debe fundamentalmente al segundo punto: solucionables separadamente En muchos sistemas para implementar la parte A, debemos conocer algo sobre la B, y para implementar algo de B, debemos conocer algo de C.

De manera similar, podemos decir que el costo de *mantenimiento* puede ser minimizado cuando las partes de un sistema son:

- Fácilmente relacionables con la aplicación
- Manejablemente pequeñas
- Corregibles separadamente

Muchas veces la persona que realiza la modificación no es quien diseñó el sistema.

Es importante que las partes de un sistema sean manejablemente pequeñas en orden de simplificar el mantenimiento. Un trabajo de encontrar y corregir un error en una "pieza" de código de 1.000 líneas es muy superior a hacerlo con piezas de 20 líneas. No solo disminuye el tiempo de localizar la falla sino que si la modificación es muy engorrosa, puede reescribirse la pieza completamente. Este concepto de "módulos descartables" ha sido utilizado con éxito muchas veces.

Por otra parte, para minimizar los costos de mantenimiento debemos lograr que cada pieza sea independiente de otra. En otras palabras debemos ser capaces de realizar modificaciones al módulo A sin introducir efectos indeseados en el módulo B.

Finalmente, diremos que el costo de *modificación* de un sistema puede minimizarse si sus partes son

- Fácilmente relacionables con la aplicación
- Modificables separadamente

En resumen, podemos afirmar lo siguiente: los costos de implementación, mantenimiento, y modificación, generalmente serán minimizados cuando:

> *Cada pieza del sistema corresponda a exactamente una pequeña, bien definida pieza del dominio del problema, y cada relación entre las piezas del sistema corresponde a relaciones entre piezas del dominio del problema.*

1.5 Como se logra el costo mínimo con Diseño Estructurado

Un buen diseño estructurado es un ejercicio de *participación* y *organización* de los componentes de un sistema.

Entenderemos por particionar, la subdivisión de un problema en subproblemas más pequeños, de tal forma que cada subproblema corresponda a una pieza del sistema.

La cuestión es: ¿Dónde y cómo debe dividirse el problema? ¿Qué aspectos del problema deben pertenecer a la misma pieza del sistema, y cuales a distintas piezas? El diseño estructurado responde estas preguntas con dos principios básicos:

- Partes del problema altamente interrelacionadas deberán pertenecer a la misma pieza del sistema.
- Partes sin relación entre ellas, deben pertenecer a diferentes piezas del sistema sin relación directa.

Otro aspecto importante del diseño estructurado es la organización del sistema. Debemos decidir cómo se interrelacionan las partes, y que parte está en relación con cual. El objetivo es organizar el sistema de tal forma que no existan piezas más grandes de lo estrictamente necesario para resolver los aspectos del problema que ella abarca. Igualmente impórtate, es el evitar la introducción de relaciones en el sistema, que no existe en el dominio del problema.

1.6 El concepto de Cajas Negras

Una caja negra es un sistema (o un componente) con entradas conocidas, salidas conocidas, y generalmente transformaciones conocidas, pero del cual no se conoce el contenido en su interior.

En la vida diaria existe innumerable cantidad de ejemplos de uso cotidiano: una radio, un televisor, un automóvil, son cajas negras que usamos a diario sin conocer (en general) como funciona en su interior. Solo conocemos como controlarlos (entradas) y las respuestas que podemos obtener de los artefactos (salidas).

El concepto de caja negra utiliza el principio de *abstracción.*

Este concepto es de suma utilidad e importancia en la ingeniería en general, y por ende en el desarrollo de software. Lamentablemente muchas veces para poder hacer un uso efectivo de determinado módulo, el diseñador debe revisar su contenido ante posibles contingencias como ser comportamientos no deseados ante determinados valores. Por ejemplo, es posible que una rutina haya sido desarrollada para aceptar un determinado rango de valores y falla si se la utiliza con valores fuera de dicho rango, o produce resultados inesperados. Una buena

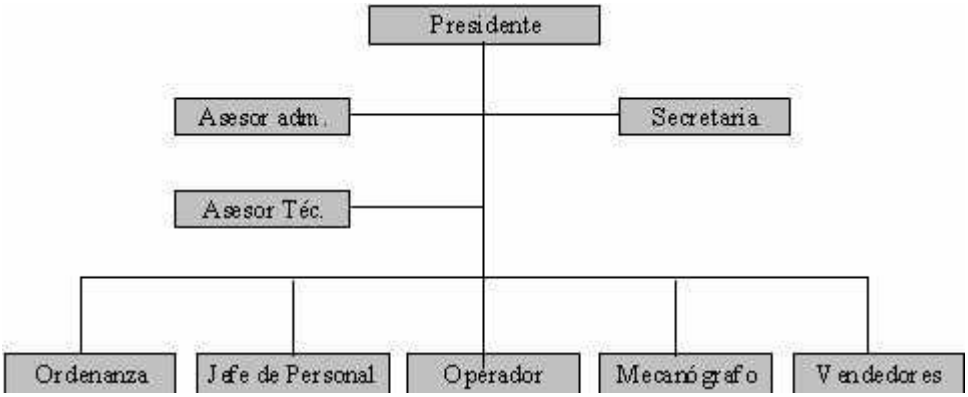
documentación en tales casos, es de utilidad pero no transforma al módulo en una verdadera caja negra. Podríamos hablar en todo caso de "cajas blancas".

Los módulos de programas de computadoras pueden variar en un amplio rango de aproximación al ideal de caja negra. En la mayoría de los casos podemos hablar de "cajas grises".

1.7 Comparación entre las estructuras administrativas y el diseño estructurado

Uno de los aspectos más interesantes del diseño de programas es la relación que puede establecerse con las estructuras de organización humana, particularmente la jerarquía de administración encontrada en la mayoría de las grandes organizaciones.

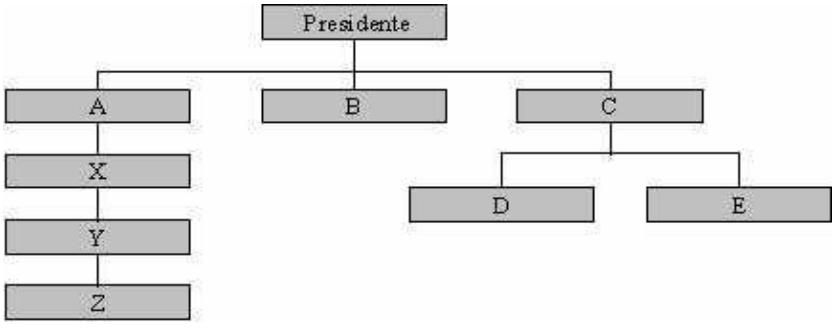
Observemos por ejemplo el siguiente diagrama de organización de una empresa



A simple vista podemos apreciar que el presidente de la empresa tiene demasiados subordinados, y consecuentemente su trabajo involucrará el manejo de demasiados datos y la toma de demasiadas decisiones, demasiada complejidad, que lo llevará a cometer posibles errores.

Podemos establecer una analogía con la estructura de programas y es razonable pensar que un módulo que tenga demasiados módulos subordinados a quienes controlar, sea sumamente complejo, y susceptible a fallas.

Veamos otro ejemplo



Podemos apreciar a simple vista que la tarea de los jefes A, X, Y, es relativamente trivial y podría ser comprimida en una sola jefatura. Estableciendo una comparación con la estructura de programas, si tenemos un módulo cuya única función es llamar a otro, y este a su vez a otro, el cual llama a uno que finalmente realizará la tarea, los módulos intermedios podrán comprimirse un único módulo de control.

Podemos decir que en una organización perfecta, los administradores no realizan ninguna tarea operativa. Su labor consiste en coordinar información entre los subordinados y tomar decisiones. Los niveles inferiores son los que realizan el trabajo operativo. Análogamente, podemos argumentar que los módulos de nivel alto en un programa o sistema solamente coordinan y controlan la ejecución de los módulos de menor nivel, quienes son los que realizan los cómputos y procesos que el sistema requiere.

Finalmente observaremos que los administradores suministran a sus subordinados únicamente la información que estrictamente necesitan. Análogamente los módulos inferiores solo deben tener acceso a la información que necesitan, y no a otras.

El establecimiento de un paralelo entre las estructuras organizativas humanas y los programas de computadora nos será muy útil en el estudio del diseño estructurado.

1.8 Manejo de la complejidad

En principio diremos que escribir un programa "grande" generalmente lleva más tiempo que escribir uno "pequeño". Esto es valedero si medimos "grande" y "pequeño" en unidades apropiadas. Claramente el número de instrucciones de un programa no es una medida de complejidad ya que existen instrucciones más complejas que otras, y algoritmos más complejos que otros. En realidad lo que diremos es que *es más difícil resolver un problema difícil!* , e intentaremos realizar un análisis sobre como disminuir la complejidad de un determinado problema.

Si asumimos que podemos medir por algún método la complejidad de un problema P (no importa aquí que método), diremos que la complejidad del mismo será M(P), y que el costo de realizar un programa que resuelva el problema P será C(P). Los enunciados previos responderán a la siguiente regla:

dados dos problemas P y Q observaremos lo siguiente

Si M(P) > M(Q) entonces C(P) > C(Q)

es decir el costo de resolver un determinado problema es directamente proporcional al tamaño del mismo.

Intentaremos tomar dos problemas separados y en lugar de escribir dos programas, crear un programa combinado. Si juntamos los dos problemas, obtendremos uno mayor que si tomamos los dos por separado. La razón fundamental para no combinar los problemas, es que los humanos tenemos inconvenientes para tratar adecuadamente grandes complejidades, y en la medida que esta se incrementa somos susceptibles a cometer un mayor número de errores. En virtud de esto podemos afirmar que

M(P+Q) > M(P) + M(Q)

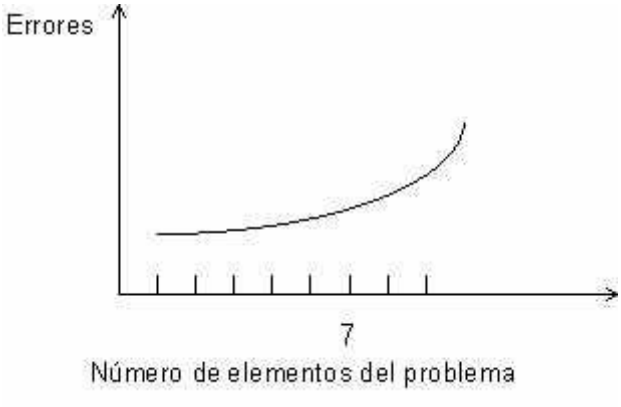
y consecuentemente

$C(P+Q) > C(P) + C(Q)$

Siempre será preferible crear dos piezas pequeñas que una sola más grande, si ambas solucionan el mismo problema.

Este fenómeno no es solo válido para el campo de la computación. Es verdadero en cualquier campo de la solución de problemas (matemática, física, etc.).

El psicólogo-matemático George Miller, realizó una serie de investigaciones sobre las limitaciones en el procesamiento de información humano. Estos estudios determinaron que la mente humana puede mantener y tratar simultáneamente hasta con siete objetos, o conceptos. En efecto, la memoria necesaria para la resolución de problemas con múltiples elementos, tiene una capacidad de 7 ± 2 entidades. Por sobre este número la cantidad de errores se incrementa desproporcionadamente en forma no lineal.



Esta es una propiedad de la capacidad de procesamiento de información del cerebro humano bien establecida sobre la que se asienta la descomposición de problemas en subproblemas.

Esto nos lleva a que dado un problema P no trivial, es conveniente descomponerlo en problemas más pequeños ya que se observa que

$C(P) > C(½\,P) + C(½\,P)$

Ahora bien, cuando se descompone una tarea en dos, si las subtareas no son realmente independientes, al solucionar una de las partes debe simultáneamente tratarse aspectos de la otra.

Supongamos que descomponemos P en dos partes iguales P' = ½ P y P" = ½ P, si las partes no son independientes, el costo de resolver el problema entero será:

$C(P' + I_1 \times P') + C(P'' + I_2 \times P'')$

donde I_1 es una fracción que representa la interacción de P' con P", e I_2 es una fracción que representa la interacción de P" con P'. Siempre que I_1 e I_2 sean mayores a cero, es obvio que será

$C(P' + I_1 \times P') + C(P'' + I_2 \times P'') > C(\frac{1}{2} P) + C(\frac{1}{2} P)$

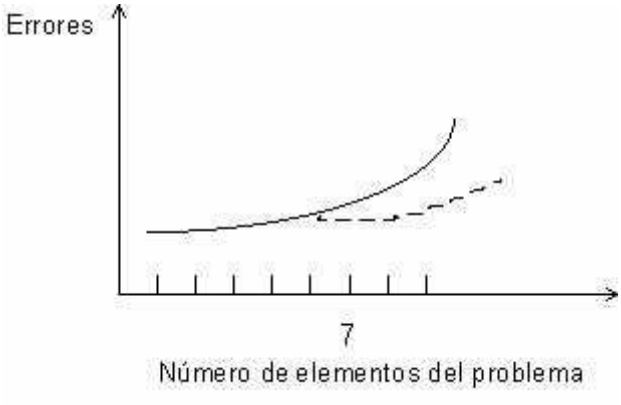
Si I_1 e I_2 son muy pequeños podremos decir que:

$C(P) > C(P' + I_1 \times P') + C(P'' + I_2 \times P'')$

Ahora, la subdivisión de un problema en otros menores tiene un límite. Es obvio que no podemos esperar dividir el sistema en un infinito número de "nadas", y en el caso límite, el desarrollo de un sistema como un número muy grande de piezas independientes es equivalente a desarrollarlo en una sola pieza.

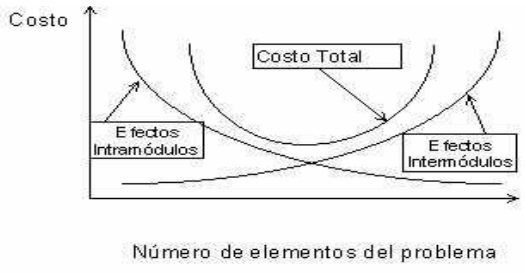
Más adelante analizaremos como determinar el tamaño conveniente de cada parte.

El proceso de factorización de un problema en partes, puede introducir algunos errores, pero en general si se realiza correctamente tiende a aplastar la curva de errores



Como puede sugerirse, la factorización de un sistema en mil módulos de una línea, es equivalente al costo de un módulo de 1000 líneas (y posiblemente mayor). Claramente estos son los extremos de un espectro de alternativas.

A medida que los módulos sean más pequeños, podemos esperar que su complejidad (y costo) disminuyan, pero además, a mayor cantidad de módulos, tendremos mayor posibilidad problemas debido a errores en las conexiones intermódulos. Estas son dos curvas en contraposición



En este punto no estamos preparados para predecir el tamaño "óptimamente pequeño" de un módulo. En realidad es muy dudoso que pueda establecerse con precisión, pero veremos una serie de principios que nos conducirán a aproximarnos.

1.9 Complejidad en términos humanos

En el punto anterior realizamos un análisis sobre la incidencia de la complejidad en los costos, y cómo manejarla a través de la subdivisión de un problema en problemas menores. Vimos que muchos de nuestros problemas en diseño y programación se deben a la limitada capacidad de la mente humana para lidiar con la complejidad.

La cuestión ahora es:

¿Qué es lo complejo para los humanos?

En otras palabras:

¿Qué aspectos del diseño de sistemas y programas son considerados complejos por el diseñador?

Y por extensión

¿Qué podemos hacer para realizar sistemas menos complejos?

En primer término podemos decir que el *tamaño* de un módulo es una medida simple de la complejidad. Generalmente un módulo de 1000 sentencias, será más complejo que uno de 10. Obviamente el problema es mayor ya que existen sentencias más complejas que otras.

Por ejemplo las *sentencias de decisión* son uno de los primeros factores que contribuyen a la complejidad de un módulo. Otro factor de importancia es el "espacio" de vida o *alcance de los elementos de dato*, es decir el número de sentencias durante las cuales el estado y valor de un elemento de datos debe ser recordado por el programador en orden de comprender que es lo que hace el módulo.

Otro aspecto relacionado con la complejidad es el *alcance o amplitud del flujo de control*, esto es el número de sentencias lexicográficamente contiguas que deben examinarse antes de encontrar una sección de código caja-negra con un punto de entrada y un punto de salida. Es interesante notar que la teoría detrás de la programación estructurada provee el medio de reducir este alcance a una mínima longitud organizando la lógica en combinaciones de operaciones de "secuencia", "decisión", e "iteración".

Todas estas medidas reconocen que la complejidad de los programas percibida por humanos, se ve altamente influenciada por el *tamaño* del módulo.

Tres factores, implícitos en el enfoque previo, han sido identificados como afectando la complejidad de las sentencias:

- La *cantidad* de información que debe ser comprendida correctamente.
- La *accesibilidad* de la información.
- La *estructura* de la información.

Estos factores determinan la probabilidad de error humano en el procesamiento de información de todo tipo.

Mientras que la complejidad de todo tipo de sentencias de programas puede evaluarse según estos criterios, enfocaremos la atención en aquellas que establecen relaciones intermodulares.

Cantidad

Por *cantidad* de información, entenderemos el número de bits de datos, en el sentido que le asigna la teoría de la información este término, que el programador debe manejar para comprender la interface.

En términos simples, esto se relaciona con el número de argumentos o parámetros que son pasados en la llamada al módulo. Por ejemplo, una llamada a una subrutina que involucra 100 parámetros, será más compleja que una que involucra solo 3.

Cuando un programador ve una referencia a un módulo en el medio de otro, él *debe* conocer cómo se resolverá la referencia, y que tipo de transformación se transmitirá.

Consideremos el siguiente ejemplo: una llamada al procedimiento SQRT

Si la llamada es: SQRT(X)

El programador inferirá que X funciona como parámetro de entrada y como valor de retorno del procedimiento. Y es muy probable que esto sea así.

Ahora bien, si la llamada es: SQRT(X,Y)

el programador inferirá que X funciona como parámetro de entrada y que el resultado es retornado en Y. Es muy probable que esto sea así, aunque podría ser al revés.

Ahora supongamos que tenemos: SQRT(X, Y, Z)

el programador podrá inferir que X funciona como parámetro de entrada, que el resultado es retornado del procedimiento, y que en Z se retorna algún código de error. Esto podría ser cierto, pero es alta la probabilidad de que el orden de los parámetros sea diferente.

Vemos que a medida que crece la cantidad de parámetros, la posibilidad de error es mayor. Puede argumentarse que esto se soluciona con una adecuada documentación, pero la realidad demuestra que en la mayoría de los casos los programas no están bien documentados.

Notaremos además que un módulo con demasiados parámetros, posiblemente esté realizando más de una función específica, y por lo tanto podría descomponerse en dos módulos más sencillos y funcionales con una menor cantidad de argumentos.

Accesibilidad

Quizá más importante que la cantidad de información es su *accesibilidad*. Cierta información acerca del uso de la interface debe ser comprendida por el programador para escribir o interpretar el código correctamente.

Consideraremos los siguientes puntos en esto:

- La interface es menos compleja si la información puede ser accedida (por el programador, no por la computadora) *directamente*; es más compleja si la información referencia *indirectamente* otros elementos de datos.

- La interface es menos compleja si la información es presentada *localmente* dentro de la misma sentencia de llamada. La interface es más compleja si la información necesaria es *remota* a la sentencia.
- La interface es menos compleja si la información es presentada en forma *estándar* que si se presenta de forma *imprevista*.
- La interface es menos compleja si su naturaleza es *obvia* es menos compleja que si su naturaleza es *obscura*.

Observaremos el siguiente ejemplo: supongamos que tenemos la función DIST que calcula la distancia existente entre dos puntos. La fórmula matemática para realizar dicho cálculo es:

$DIST = SQRT ((y1 - y0)^2 + (x1 - x0)^2)$

Consideraremos las siguientes interfaces:

Opción 1. CALL DIST (X0, Y0, X1, Y1, DISTANCIA)

Opción 2. CALL DIST (ORIGEN, FIN, DISTANCIA)

Opción 3. CALL DIST (XCOORDS, YCOORDS, DISTANCIA)

Opción 4. CALL DIST (LINEA, DISTANCIA)

Opción 5. CALL DIST (LINETABLA)

Opción 6. CALL DIST ( )

Trataremos de determinar cuál de las interfaces es la menos compleja.

A primera vista podemos pensar que la opción 1 es la más compleja ya que involucra el mayor número de parámetros. Sin embargo la opción 1 presenta los parámetros en forma *directa*.

En contraste, la opción 2 presenta la información de manera *indirecta.* En orden de comprender la interface, deberemos ir a otra parte del programa y verificar que ORIGEN se define en términos de subelementos X0 e Y0, FIN como X1 e Y1.

La opción 3, además de presentar la información en forma *indirecta* además la presenta en forma *no* estándar lo cual complica más la interface.

La opción 4 presenta la misma desventaja que 2 y 3, presentando los valores en forma remota.

La opción 5 es aún más compleja. El identificador LINETABLE es *obscuro*.

La opción 6 a diferencia de las anteriores no representa parámetros localmente sino en forma *remota*.

Lamentablemente, algunos lenguajes como COBOL no permiten la llamada a módulos con parámetros dentro de un mismo programa. Además, existe cierta aversión a utilizar llamadas parametrizadas por algunas personas fundamentadas principalmente en:

- Parametrizar una interface requiere más trabajo
- El proceso de parametrización mismo puede introducir errores
- En general la velocidad del programa es menor que cuando se usan variables globales

Estructura

Finalmente observaremos que la estructura de la información puede ser un punto clave en la complejidad.

La primera observación es que la información es menos compleja si se presenta en forma *lineal* y más compleja si se presenta en forma *anidada.*

La segunda observación es que la información es menos compleja si se presenta en modo *afirmativo* o *positivo*, y es más compleja si se presenta en modo *negativo*.

Ambos conceptos tienen aplicación primaria en la escritura de código de programas. Por ejemplo, ciertas construcciones de sentencias IF anidadas son más complejas de entender que un secuencia de varias sentencias IF simples.

Similarmente, las expresiones lógicas que involucran operadores de negación (NOT) son más difíciles de comprender que aquellas que no lo presentan.

Estas filosofías de pensamiento linear y positivo también son importantes en las referencias intermódulares. Supongamos la siguiente instrucción:

DISTANCIA = SQRT (sum(square (dif (Y1,Y0), square (dif (X1, X0))))

Normalmente esta expresión para el común de los programadores resultará complicada de leer. Si por el contrario descomponemos la expresión en otras menores tendremos una mayor cantidad de elementos lineares, y una reducción en el anidamiento. La secuencia de expresiones resultantes resulta más sencilla de leer:

A = dif (Y1, Y0)

B = dif (X1, X0)

A2 = square (A)

B2 = square (B)

DISTANCIA = SQRT (sum (A2, B2))

1.10 Acoplamiento

Muchos aspectos de la modularización pueden ser comprendidos solo si se examinan módulos en relación con otros. En principio veremos el concepto de ***independencia***. Diremos que dos módulos son totalmente independientes si ambos pueden funcionar completamente sin la

presencia del otro. Esto implica que no existen interconexiones entre los módulos, y que se tiene un valor cero en la escala de "dependencia".

En general veremos que a mayor número de interconexiones entre dos módulos, se tiene una menor independencia.

El concepto de independencia funcional es una derivación directa del de modularidad y de los conceptos de abstracción y ocultamiento de la información.

La cuestión aquí es: ¿cuánto debe conocerse acerca de un módulo para poder comprender otro módulo? Cuanto más debamos conocer acerca del módulo B para poder comprender el módulo A, menos independientes serán A de B.

La simple cantidad de conexiones entre módulos, no es una medida completa de la independencia funcional. La independencia funcional se mide con dos criterios cualitativos: *acoplamiento* y *cohesión*. Estudiaremos en principio el primero de ellos.

Módulos altamente "acoplados" estarán unidos por fuertes interconexiones, módulos débilmente acoplados tendrán pocas y débiles interconexiones, en tanto que los módulos "desacoplados" no tendrán interconexiones entre ellos y serán independientes.

El *acoplamiento* es un concepto abstracto que nos indica el grado de interdependencia entre módulos.

En la práctica podemos materializarlo como la probabilidad de que en la codificación, depuración, o modificación de un determinado módulo, el programador necesite tomar conocimiento acerca de partes de otro módulo. Si dos módulos están fuertemente acoplados, existe una alta probabilidad de que el programador necesite conocer uno de ellos en orden de intentar realizar modificaciones al otro.

Claramente, el costo total del sistema se verá fuertemente influenciado por el grado de acoplamiento entre los módulos.

1.10.1Factores que influencian el Acoplamiento

Los cuatro factores principales que influyen en el acoplamiento entre módulos son:

- Tipo de conexión entre módulos: los sistemas normalmente conectados, tienen menor acoplamiento que aquellos que tienen conexiones patológicas.
- Complejidad de la interface: Esto es aproximadamente igual al número de ítems diferentes pasados (no cantidad de datos). Más ítems, mayor acoplamiento.
- Tipo de flujo de información en la conexión: los sistemas con acoplamiento de datos tienen menor acoplamiento que los sistemas con acoplamiento de control, y estos a su vez menos que los que tienen acoplamiento híbrido.
- Momento en que se produce el ligado de la Conexión: Conexiones ligadas a referentes fijos en tiempo de ejecución, resultan con menor acoplamiento que cuando el ligado tiene lugar en tiempo de carga, el cual tiene a su ver menor acoplamiento que cuando el ligado se realiza en tiempo de linkage-edición, el cual tiene menos acoplamiento que el que se realiza en tiempo de compilación, todos los que a su vez tiene menos acoplamiento que cuando el ligado se realiza en tiempo de codificación.

1.10.2 Tipos de conexiones entre módulos

Una conexión en un programa, es una referencia de un elemento, por nombre, dirección, o identificador de otro elemento.

Una conexión intermódular ocurre cuando el elemento referenciado está en un módulo diferente al del elemento referenciante.

El elemento referenciado define una interface, un límite del módulo, a través del cual fluyen datos y control.

La interface puede considerarse como residente en el elemento referenciado. Puede pensarse como un enchufe (socket) donde la conexión del elemento referenciante se inserta.

Toda interface en un módulo representa cosas que deben ser conocidas, comprendidas, y apropiadamente conectadas por los otros módulos del sistema.

Se busca minimizar la complejidad del sistema/módulo, en parte, minimizando el número y complejidad de las interfaces por módulo.

Todo módulo además debe tener al menos una interface para ser definido y vinculado al resto del sistema.

Pero, ¿es una interface de identidad simple suficiente para implementar sistemas que funcionen adecuadamente? La cuestión aquí es: *¿A qué propósito sirven las interfaces?*

Solo flujos de *control* y *datos* pueden pasarse entre módulos en un sistema de programación. Una interface puede cumplir las siguientes cuatro únicas funciones:

- Transmitir datos a un módulo como parámetros de entrada
- Recibir datos desde un módulo como resultados de salida
- Ser un nombre por el cual se recibe el control
- Ser un nombre por el cual se transmite el control

Un módulo puede ser identificado y activado por medio de una interfaz de identidad simple. También podemos pasar datos a un módulo sin agregar otras interfaces, haciendo a la interfaz de entrada capaz de aceptar datos como control. Esto requiere que los elementos de datos sean pasados dinámicamente como argumentos (parámetros) como parte de la secuencia de activación, que da el control a un módulo; cualquier referencia estática a datos puede introducir nuevas interfaces.

Se necesita también que la interface de identidad de un módulo sirva para transferir el retorno del control al módulo llamador. Esto puede realizarse haciendo que la transferencia de control desde el llamador sea una transferencia *condicional.* Debe implementarse además un mecanismo para transmitir datos de retorno desde el módulo llamado hacia el llamador. Puede asociarse un valor a una activación particular del módulo llamado, la cual pueda ser usada contextualmente en el llamador. Tal es el caso de las funciones lógicas. Alternativamente pueden transmitirse parámetros para definir ubicaciones donde el módulo llamado retorna valores al llamador.

Si todas las conexiones de un sistema se restringen a ser completamente parametrizadas (con respecto a sus entradas y salidas), y la transferencia condicional de control a cada módulo se realiza a través de una identidad simple y única, diremos que el sistema está *mínimamente conectado*.

Diremos que un sistema está *normalmente conectado* cuando cumple con las condiciones de mínimamente conectado, excepto por alguna de las siguientes consideraciones:

- Existe más de un punto de entrada para un mismo módulo
- El módulo activador o llamador puede especificar como parte del proceso de activación un punto de retorno que no sea la próxima sentencia en el orden de ejecución.
- El control es transferido a un punto de entrada de un módulo por algún mecanismo distinto a una llamada explícita (ej. perform thru del COBOL).

El uso de múltiples puntos de entrada garantiza que existirán más que el número mínimo de interconexiones para el sistema. Por otra parte si cada punto de entrada determina funciones con mínima conexión a otros módulos, el comportamiento del sistema será similar a uno mínimamente interconectado.

De cualquier manera, la presencia de múltiples puntos de entrada a un mismo módulo, puede ser un indicativo de que el módulo está llevando a cabo más de una función específica. Además, es una excelente oportunidad para que el programador superpondrá parcialmente el código de las funciones comprendidas dentro del mismo módulo, quedando dichas funciones *acopladas por contenido*.

De manera similar, *los puntos de retorno alternativo* son frecuentemente útiles dentro del espíritu de los sistemas normalmente conectados. Esto se da cuando un módulo continuará su ejecución en un punto que depende del valor resultante de una decisión realizada por un módulo subordinado invocado previamente. En un caso de mínima conexión, el módulo subordinado retornará el valor como un parámetro, el cual deberá ser testeado nuevamente en el módulo superior. Sin embargo, el módulo superior puede indicar por algún medio directamente el punto donde debe continuarse la ejecución del programa, (un valor relativo + o - direcciones a partir de la instrucción llamadora, o un parámetro con una dirección explícita).

Si un sistema no está mínima o normalmente conectados, entonces algunos de sus módulos presentarán conexiones patológicas. Esto significa que al menos un módulo tendrá referencias explícitas a identificadores definidos dentro de los límites de otro módulo.

1.10.3 Complejidad de la interface

La segunda dimensión del acoplamiento es la *complejidad de interface*. Cuanto más compleja es una conexión, mayor acoplamiento se tiene. Un módulo con una interface de 100 parámetros generará mayor acoplamiento que un que solo necesite tres parámetros.

El significado de "complejidad" es el de complejidad en términos humanos, como lo visto anteriormente.

1.10.4 Flujo de Información

Otro aspecto importante del acoplamiento tiene que ver con el *tipo* de información que se transmite entre el módulo superior y subordinado. Distinguiremos tres tipos de flujo de información:

- datos
- control
- híbrido

Los datos son información sobre la cual una pieza de programa opera, manipula, o modifica.

La información de control (aun cuando está representada por variables de dato) es aquella que gobierna como se realizarán las operaciones o manipulaciones sobre los datos.

Diremos que una conexión presenta *acoplamiento por datos* si la salida de datos del módulo superior es usada como entrada de datos del subordinado. Este tipo de acoplamiento también es conocido como de entrada-salida.

Diremos que una conexión presenta *acoplamiento de control* si el módulo superior comunica al subordinado información que controlará la ejecución del mismo. Esta información puede pasarse como datos utilizados como señales o "banderas" (flags) o bien como direcciones de memoria para instrucciones de salto condicional (branch-adress). Estos son elementos de control "disfrazados" como datos.

El acoplamiento de datos es mínimo, y ningún sistema puede funcionar sin él.

La comunicación de datos es *necesaria* para el funcionamiento del sistema, sin embargo, la comunicación de control es una característica no deseable y *prescindible*, que sin embargo aparece muy frecuentemente en los programas.

Se puede minimizar el acoplamiento si solo se transmiten datos a través de las interfaces del sistema.

El acoplamiento de control abarca todas las formas de conexión que comuniquen elementos de control. Esto no solo involucra transferencia de control (direcciones o banderas), si no que puede involucrar el pasaje de datos que cambia, regula, o sincroniza la ejecución de otro módulo.

Esta forma de acoplamiento de control indirecto o secundario se conoce como *coordinación*. La coordinación involucra a un módulo en el contexto procedural de otro. Esto puede comprenderse con el siguiente ejemplo: supongamos que el módulo A llama al módulo B suministrándole elementos de datos discretos. La función del módulo B es la de agrupar estos elemento de datos en un ítem compuesto y retornárselo al módulo A (superior). El módulo B enviará al módulo A, señales o banderas indicando que necesita que se le suministre otro ítem elemental, o para indicarle que le está devolviendo el ítem compuesto. Estas banderas serán utilizadas dentro del módulo A para coordinar su funcionamiento y suministrar a B lo requerido.

Cuando un módulo modifica el contenido procedural de otro módulo, decimos que existe *acoplamiento híbrido*. El acoplamiento híbrido es una modificación de sentencias intermódular. En este caso, para el módulo destino o modificado, el acoplamiento es visto como de control en tanto que para el módulo llamador o modificador es considerado como de datos.

El grado de interdependencia entre dos módulos vinculados con acoplamiento híbrido es muy fuerte. Afortunadamente es una práctica en decadencia y reservada casi con exclusividad a los programadores en ensamblador.

1.10.5 Tiempo de ligado de conexiones intermódulares

"Ligado" o "Binding" es un término comúnmente usado en el campo del procesamiento de datos para referirse a un proceso que resuelve o fija los valores de identificadores dentro de un sistema.

El ligado de variables a valores, o más genéricamente, de identificadores a referentes específicos, puede tener lugar en diferentes estadios o períodos en la evolución del sistema. La historia de tiempo de un sistema puede pensarse como una línea extendiéndose desde el momento de la escritura del código fuente hasta el momento de su ejecución. Dicha línea puede subdividirse en diferentes niveles de refinamiento según distintas combinaciones de computador/lenguaje/compilador/sistema operativo.

De esta forma, el ligado puede tener lugar cuando el programador escribe una sentencia en el editor de código fuente, cuando un módulo es compilado o ensamblado, cuando el código objeto (compilado o ensamblado) es procesado por el "link-editor" o el "link-loader" (generalmente este proceso es el conocido como ligado en la mayoría de los sistemas), cuando el código "imagen-de-memoria" es cargado en la memoria principal, y finalmente cuando el sistema es ejecutado.

La importancia del tiempo de ligado radica en que *cuando el valor de variables dentro de una pieza de código es fijado más tarde, el sistema es más fácilmente modificable y adaptable al cambio de requerimientos.*

Veamos un ejemplo: supongamos que se nos encomienda la escritura de una serie de programas listadores siendo la impresora a utilizar en principio una del tipo matricial de 80 columnas que funciona con papel continúo de 12" de largo de página.

Alternativas:

1. Escribimos el literal "72" en todas las rutinas de impresión de todos los programas. (ligado en tiempo de escritura)
2. Reemplazamos el literal por la constante manifiesta LONG_PAG a la que asignamos el valor "72" en todos los programas (ligado en tiempo de compilación)
3. Ponemos la constante LONG_PAG en un archivo de inclusión externo a los programas (ligado en tiempo de compilación)
4. Nuestro lenguaje no permite la declaración de constantes por lo cual definimos una variable global LONG_PAG a la que le asignamos el valor de inicialización "72" (ligado en tiempo de link-edición)

5. Definimos un archivo de parámetros del sistema con un campo LONG_PAG al cual se le asigna el valor "72". Este valor es leído junto con otros parámetros cuando el sistema se inicia. (ligado en tiempo de ejecución)
6. Definimos en el archivo de parámetros un registro para cada terminal del sistema y personalizamos el valor del campo LONG_PAG según la impresora que tenga vinculada cada terminal. De esta forma las terminales que tienen impresoras de 12" imprimen 72 líneas por página, y las que tienen una impresora de inyección de tinta que usan papel oficio, imprimen 80. (ligado en tiempo de ejecución)

Examinaremos ahora la relación existente entre el tiempo de ligado y las conexiones intermódulares, y como el mismo afecta el grado de acoplamiento entre módulos.

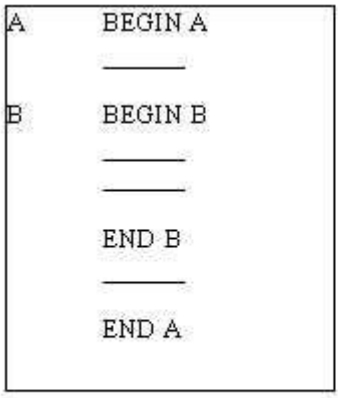
Nuevamente, una referencia intermódular fijada a un referente u objeto específico en tiempo de definición, tendrá un acoplamiento mayor a una referencia fijada en tiempo de traslación o posterior aún.

La posibilidad de compilación independiente de un módulo de otros facilitará el mantenimiento y modificación del sistema, que si debiera compilarse todos los módulos juntos. Igualmente, si la link-edición de los módulos es diferida hasta el instante previo a su ejecución, la implementación de cambios se verá simplificada.

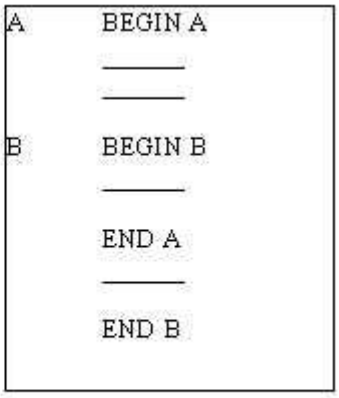
Existe un caso particular de acoplamiento de módulos derivado de la *estructura lexicográfica* del programa. Hablamos en este caso de *acoplamiento por contenido.*

Dos formas de acoplamiento por contenido pueden distinguirse:

- Inclusión lexicográfica: se da cuando un módulo está incluido lexicográficamente en otro, y es una forma menor de acoplamiento. Los módulos por lo general no pueden ejecutarse separadamente. Este es el caso en el que el módulo subordinado es activado en línea dentro del contexto del módulo superior.
- Solapamiento parcial: es un caso extremo de acoplamiento por contenido. Parte del código de un módulo está en intersección con el otro. Afortunadamente la mayoría de los lenguajes modernos de alto nivel no permiten este tipo de estructuras.



Inclusión Lexicográfica



Solapamiento parcial

En términos de uso, mantenimiento, y modificación, las consecuencias del acoplamiento por contenido son peores que las del acoplamiento de control. El acoplamiento por contenido hace

que los módulos no puedan funcionar uno sin el otro. No ocurre lo mismo en el acoplamiento de control, en el cual un módulo, aunque reciba información de control, puede ser invocado desde diferentes puntos del sistema.

1.10.6 Acoplamiento de Entorno Común (common-environment coupling)

Siempre que dos o más módulos interactúan con un entorno de datos común, se dice que dichos módulos están en *acoplamiento por entorno común.*

Ejemplos de entorno común pueden ser áreas de datos globales como la DATA división del COBOL o un archivo en disco.

El acoplamiento de entorno común es una forma de acoplamiento de segundo orden, distinto de los tratados anteriormente. La severidad del acoplamiento dependerá de la cantidad de módulos que acceden simultáneamente al entorno común. En el caso extremo de solo dos módulos donde uno utiliza como entrada los datos generados por el otro hablaremos de un acoplamiento de *entrada-salida*.

El punto es que el acoplamiento por entorno común no es necesariamente malo y deba ser evitado a toda costa. Por el contrario existen ciertas circunstancias en que es una opción válida.

1.10.7 Desacoplamiento

El concepto de acoplamiento invita a un concepto recíproco: *desacoplamiento.* Desacoplamiento es cualquier método sistemático o técnica para hacer más independientes a los módulos de un programa.

Cada tipo de acoplamiento generalmente sugiere un método de desacoplamiento. Por ejemplo, el acoplamiento causado por ligado, puede desacoplarse cambiando los parámetros apropiados tal lo visto en el ejemplo del contador de líneas de los programas impresores.

El desacoplamiento, desde el punto de vista funcional, rara vez puede realizarse, excepto en los comienzos de la fase del diseño.

Como regla general, una disciplina de diseño que favorezca el acoplamiento de entrada-salida y el acoplamiento de control por sobre el acoplamiento por contenido y el acoplamiento híbrido, y que busque limitar el alcance del acoplamiento por entorno común es el enfoque más efectivo.

Otras técnicas para reducir el acoplamiento son:

- Convertir las referencias implícitas en explícitas. Lo que puede verse con mayor facilidad es más fácil de comprender.
- Estandarización de las conexiones.
- Uso de "buffers" para los elementos comunicados en una conexión. Si un módulo puede ser diseñado desde el comienzo asumiendo que un buffer mediará cada corriente de comunicación, las cuestiones temporización, velocidad, frecuencia, etc., dentro de un módulo no afectarán el diseño de otros.
- Localización. Utilizado para reducir el acoplamiento por entorno común. Consiste en

dividir el área común en regiones para que los módulos solo tengan acceso a aquellos datos que les son de su estricta incumbencia.

Según Meiler Page-Jones (15), el acoplamiento se clasifica en:

- Acoplamiento normal por datos
- Acoplamiento normal por estampado
- Acoplamiento normal por empaquetado
- Acoplamiento normal de control
- Acoplamiento hibrido
- Acoplamiento común
- Acoplamiento patológico (por contenido)

Favorable

Desfavorable

Acoplamiento normal por datos. Toda conexión se realiza explícitamente por el mínimo número de parámetros, siendo los datos del tipo primitivo o elemental.

Acoplamiento normal por estampado. Se presenta cuando las piezas de datos que se intercambian so compuestas como estructuras o arreglos. Cuando se desea establecer el nivel de acoplamiento de un módulo, cada uno de los datos compuestos es contabilizado como uno, y no el número de elementos que componen la estructura.

Acoplamiento normal por empaquetado. Se agrupan en una estructura elementos que no están relacionados, con el único propósito de reducir el número de parámetros. Al momento de calificar el nivel de acoplamiento normal, cada uno de los elementos de los datos compuestos es contabilizado en uno.

Acoplamiento normal de control. Un módulo intercambia con otro modulo información que desea alterar la lógica interna del otro modulo. La información indicará expresamente la acción que debe realizar el otro módulo.

Acoplamiento hibrido. Los módulos intercambian información cuyo es diferente para el modulo llamador y el modulo llamado. Para el modulo llamado, el parámetro es visto de control y para el modulo llamador el parámetro es visto como un dato.

Acoplamiento común. Dos módulos intercambian información mediante variables globales, ya que una variable global no está protegida en ningún modulo, cualquier porción de código puede modificar el valor de esa variable haciendo que el comportamiento del módulo sea impredecible

Acoplamiento patológico. Dos módulos tienen la posibilidad de afectar datos de otro a través de errores en la programación. Esta situación puede darse, por ejemplo, cuando un módulo escribe los datos de otro modulo, por ejemplo en el mal uso de apuntadores o el uso explícito de memoria para modificar variables.

1.11 Cohesión

1.11.1 Relación Funcional

Hemos visto que la determinación de módulos en un sistema no es arbitraria. La manera en la cual dividimos físicamente un sistema en piezas (particularmente en relación con la estructura del problema) puede afectar significativamente la complejidad estructural del sistema resultante, así como el número total de referencias intermódulares.

Adaptar el diseño del sistema a la estructura del problema (o estructura de la aplicación, o dominio del problema) es una filosofía de diseño sumamente importante. A menudo encontramos que elementos de procesamiento del dominio de problemas altamente relacionados, son trasladados en código altamente interconectado. Las estructuras que agrupan elementos del problema altamente interrelacionados, tienden a ser modularmente efectivas.

Imaginemos que tengamos una magnitud para medir el grado de relación funcional existente entre pares de módulos. En términos de tal medida, diremos que el sistema más modularmente efectivo será aquel cuya suma de relación funcional entre pares de elementos que pertenezcan a diferentes módulos sea mínima. Entre otras cosas, esto tiende a minimizar el número de conexiones intermódulares requeridas y el acoplamiento intermódular.

Esta relación funcional intramódular se conoce como *cohesión*.

> La cohesión es la medida cualitativa de cuan estrechamente relacionados están los elementos internos de un módulo.

Otros términos utilizados frecuentemente son "fuerza modular", "ligazón", y "funcionalidad".

En la práctica un elemento de procesamiento simple aislado, puede estar funcionalmente relacionado en diferentes grados a otros elementos. Como consecuencia, diferentes diseñadores, con diferentes "visiones" o interpretaciones de un mismo problema, pueden obtener diferentes estructuras modulares con diferentes niveles de cohesión y acoplamiento. A esto se suma el inconveniente de que muchas veces es difícil evaluar el grado de relación funcional de un elemento respecto de otro.

La cohesión modular puede verse como el cemento que amalgama juntos a los elementos de procesamiento dentro de un mismo módulo. Es el factor más crucial en el diseño estructurado, y el de mayor importancia en un diseño modular efectivo.

Este concepto representa la técnica principal que posee un diseñador para mantener su diseño lo más semánticamente próximo al problema real, o dominio de problema.

Claramente los conceptos de cohesión y acoplamiento están íntimamente relacionados. Un mayor grado de cohesión implica un menor de acoplamiento. Maximizar el nivel de cohesión intramódular en todo el sistema resulta en una minimización del acoplamiento intermódular.

Matemáticamente el cálculo de la relación funcional intramódular (cohesión), involucra menos pares de elementos a los cuales debe aplicarse la medida, en comparación con el cálculo de la relación funcional intermódular (acoplamiento).

Ambas medidas son excelentes herramientas para el diseño modular efectivo, pero de las dos la más importante y extensiva es la cohesión.

Una cuestión importante a determinar es *como* reconocer la relación funcional.

El principio de cohesión puede ponerse en práctica con la introducción de la idea de un *principio asociativo*

En la decisión de poner ciertos elementos de procesamiento en un mismo módulo, el diseñador, utiliza el principio de que ciertas *propiedades* o *características* relacionan a los elementos que las poseen. Esto es, el diseñador pondrá el objeto Z en el mismo módulo que X e Y, porque X, Y, y Z poseen un misma propiedad. De esta manera, el principio asociativo es *relacional*, y es usualmente verificable en tales términos (p. e. "es correcto poner Z junto a X e Y, porque tiene la misma propiedad que ellos") o en términos de miembro de un conjunto (p. e."Es correcto poner Z junto a X e Y, pues todos pertenecen al mismo conjunto").

Debe tenerse en mente que la cohesión se aplica sobre todo el módulo, es decir sobre todos los pares de elementos. Así, si Z está relacionado a X e Y, pero no a A, B, y C, los cuales pertenecen al mismo módulo, la inclusión de Z en el módulo, redundará en baja cohesión del mismo.

Intencionalmente se ha usado el término "elemento de procesamiento" en esta discusión, en lugar de términos más comunes como instrucción o sentencia. Porqué:

Primero, un elemento de procesamiento puede ser algo que debe ser realizado en un módulo pero que aún no ha sido reducido a código. En orden de diseñar sistemas altamente modulares, debemos poder determinar la cohesión de módulos que todavía no existen.

Segundo, *elementos de procesamiento* incluyen *todas* las sentencias que aparecen en un módulo, no solo el procesamiento realizado por las instrucciones ejecutadas dentro de dicho módulo, sino también las que resultan de la invocación de subrutinas.

Por ejemplo, las sentencias individuales encontradas en el módulo B, el cual es invocado desde el módulo A, *no* figuran dentro de la cohesión del módulo A. Sin embargo el procesamiento global (función) realizado por la llamada al módulo B, es claramente un elemento de procesamiento en el módulo llamador A, y por lo tanto participa en la cohesión del módulo A.

1.11.2 Niveles de Cohesión

Diferentes principios asociativos fueron desenvolviéndose a través de los años por medio de la experimentación, argumentos teóricos, y la experiencia práctica de muchos diseñadores.

Existen siete niveles de cohesión distinguibles por siete principios asociativos. Estos se listan a continuación en orden creciente del grado de cohesión, de menor a mayor relación funcional:

- Cohesión Casual (la peor)
- Cohesión Lógica (sigue a la peor)
- Cohesión Temporal (de moderada a pobre)
- Cohesión de Procedimiento (moderada)
- Cohesión de Comunicación (moderada a buena)
- Cohesión Secuencial

- Cohesión Funcional (la mejor)

Podemos visualizar el grado de cohesión como un espectro que va desde un máximo a un mínimo.

1.11.2.1 Cohesión Casual (la peor)

La *cohesión casual* ocurre cuando existe poca o ninguna relación entre los elementos de un módulo.

La cohesión casual establece un punto cero en la escala de cohesión.

Es muy difícil encontrar módulos puramente casuales. Puede aparecer como resultado de la modularización de un programa ya escrito, en el cual el programador encuentra un determinada secuencia de instrucciones que se repiten de forma aleatoria, y decide por lo tanto agruparlas en una rutina.

Otro factor que influenció muchas veces la confección de módulos casualmente cohesivos, fue la mala práctica de la programación estructurada, cuando los programadores mal entendían que modularizar consistía en cambiar las sentencias GOTO por llamadas a subrutinas

Finalmente diremos que si bien en la práctica es difícil encontrar módulos casualmente cohesivos en su totalidad, es común que tengan elementos casualmente cohesivos. Tal es el caso de operaciones de inicialización y terminación que son puestas juntas en un módulo superior.

Debemos notar que si bien la cohesión casual no es necesariamente perjudicial (de hecho es preferible un programa casualmente cohesivo a uno lineal), dificulta las modificaciones y mantenimiento del código.

1.11.2.2 Cohesión Lógica (sigue a la peor)

Los elementos de un módulo están *lógicamente* asociados si puede pensarse en ellos como pertenecientes a la misma clase lógica de funciones, es decir aquellas que pueden pensarse como juntas lógicamente.

Por ejemplo, se puede combinar en un módulo simple todos los elementos de procesamiento que caen en la clase de "entradas", que abarca todas las operaciones de entrada.

Podemos tener un módulo que lea desde consola una tarjeta con parámetros de control, registros con transacciones erróneas de un archivo en cinta, registros con transacciones válidas de otro archivo en cinta, y los registros maestros anteriores de un archivo en disco. Este módulo que podría llamarse "Lecturas", y que agrupa todas las operaciones de entrada, es lógicamente cohesivo.

La cohesión lógica es más fuerte que la casual, debido a que representa un mínimo de asociación entre el problema y los elementos del módulo. Sin embargo podemos ver que un módulo lógicamente cohesivo no realiza una función específica, sino que abarca una serie de funciones.

1.11.2.3 Cohesión Temporal (de moderada a pobre)

Cohesión temporal significa que todos los elementos de procesamiento de una colección ocurren en el mismo período de tiempo durante la ejecución del sistema. Debido a que dicho procesamiento debe o puede realizarse en el mismo período de tiempo, los elementos asociados temporalmente pueden combinarse en un único módulo que los ejecute a la misma vez.

Existe una relación entre cohesión lógica y la temporal, sin embargo, la primera no implica una relación de tiempo entre los elementos de procesamiento. La cohesión temporal es más fuerte que la cohesión lógica, ya que implica un nivel de relación más: el factor tiempo. Sin embargo la cohesión temporal aún es pobre en nivel de cohesión y acarrea inconvenientes en el mantenimiento y modificación del sistema.

Un ejemplo común de cohesión temporal son las rutinas de inicialización (start-up) comúnmente encontradas en la mayoría de los programas, donde se leen parámetros de control, se abren archivos, se inicializan variables contadores y acumuladores, etc.

1.11.2.4 Cohesión de Procedimiento (moderada)

Elementos de procesamiento relacionados *proceduralmente* son elementos de una unidad procedural común. Estos se combinan en un módulo de cohesión procedural. Una unidad procedural común puede ser un proceso de iteración (loop) y de decisión, o una secuencia linear de pasos. En este último caso la cohesión es baja y es similar a la cohesión temporal, con la diferencia que la cohesión temporal no implica una determinada secuencia de ejecución de los pasos.

Al igual que en los casos anteriores, para decir que un módulo tiene *solo* cohesión procedural, los elementos de procesamiento deben ser elementos de alguna iteración, decisión, o secuencia, pero no deben estar vinculados con ningún principio asociativo de orden superior.

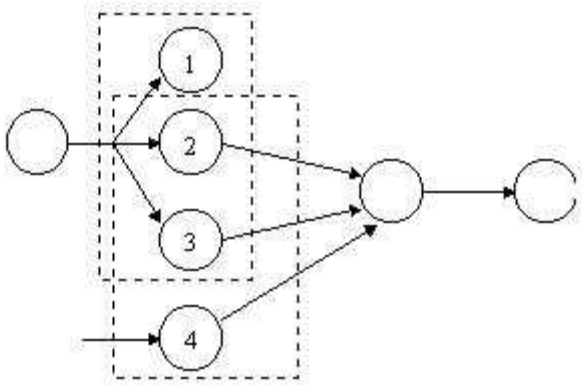
La cohesión procedural asocia elementos de procesamiento sobre la base de sus relaciones algorítmicas o procedurales.

Este nivel de cohesión comúnmente se tiene como resultado de derivar una estructura modular a partir de modelos de procedimiento como ser diagramas de flujo, o diagramas Nassi-Shneiderman.

1.11.2.5 Cohesión de Comunicación (moderada a buena)

Ninguno de los niveles de cohesión discutidos previamente está fuertemente vinculado a una estructura de problema en particular. *Cohesión de Comunicación* es el menor nivel en el cual encontramos una relación entre los elementos de procesamiento que es intrínsecamente *dependiente del problema*.

Decir que un conjunto de elementos de procesamiento están vinculados por comunicación significa que *todos los elementos operan sobre el mismo conjunto de datos* de entrada o de salida.



En el diagrama de la figura podemos observar que los elementos de procesamiento 1, 2, y 3, están asociados por comunicación sobre la corriente de datos de entrada, en tanto que 2, 3, y 4 se vinculan por los datos de salida.

Los diagramas de flujo de datos (DFD) son un medio objetivo para determinar si los elementos en un módulo están asociados por comunicación.

Las relaciones por comunicación presentan un grado de cohesión aceptable.

La cohesión por comunicación es común en aplicaciones comerciales. Ejemplos típicos pueden ser

- Un módulo que imprima o grabe un archivo de transacciones
- Un módulo que reciba datos de diferentes fuentes, y los transforme y ensamble en una línea de impresión.

1.11.2.6 Cohesión Secuencial

El siguiente nivel de cohesión en la escala es la asociación *secuencial*. En ella, los datos de salida (resultados) de un elemento de procesamiento sirven como datos de entrada al siguiente elemento de procesamiento.

En términos de un diagrama de flujo de datos de un problema, la cohesión secuencial combina una cadena lineal de transformaciones sucesivas de datos.

Este es claramente un principio asociativo relacionado con el dominio del problema.

1.11.2.7 Cohesión Funcional (la mejor)

En el límite superior del espectro de relación funcional encontramos la *cohesión funcional*. En un módulo completamente funcional, cada elemento de procesamiento, es parte integral de, y esencial para, la realización de una función simple.

En términos prácticos podemos decir que cohesión funcional es aquella que no es secuencial, por comunicación, por procedimiento, temporal, lógica, o casual.

Los ejemplos más claros y comprensibles provienen del campo de las matemáticas. Un módulo para realizar el cálculo de *raíz cuadrada* ciertamente será altamente cohesivo, y probablemente, completamente funcional. Es improbable que haya elementos superfluos más

allá de los absolutamente esenciales para realizar la función matemática, y es improbable que elementos de procesamiento puedan ser agregados sin alterar el cálculo de alguna forma.

En contraste un módulo que calcule raíz cuadrada y coseno, es improbable que sea enteramente funcional (deben realizarse dos funciones ambiguas).

En adición a estos ejemplos matemáticos obvios, usualmente podemos reconocer módulos funcionales que son elementales en naturaleza. Un módulo llamado LEER-REGISTRO-MAESTRO, o TRATAR-TRANS-TIPO3, presumiblemente serán funcionalmente cohesivos, en cambio TRATAR-TODAS-TRANS presumiblemente realizará más de una función y será lógicamente cohesivo.

1.11.3 Criterios para establecer el grado de cohesión

Una técnica útil para determinar si un módulo está acotado funcionalmente es escribir una frase que describa la función (propósito) del módulo y luego examinar dicha frase. Puede hacerse la siguiente prueba:

1. Si la frase resulta ser una sentencia compuesta, contiene una coma, o contiene más de un verbo, probablemente el módulo realiza más de una función; por tanto, probablemente tiene vinculación secuencial o de comunicación.
2. Si la frase contiene palabras relativas al tiempo, tales como "primero", "a continuación", "entonces", "después", "cuando", "al comienzo", etc., entonces probablemente el módulo tiene una vinculación secuencial o temporal.
3. Si el predicado de la frase no contiene un objeto específico sencillo a continuación del verbo, probablemente el módulo esté acotado lógicamente. Por ejemplo *editar todos los datos* tiene una vinculación lógica; *editar sentencia fuente* puede tener vinculación funcional.
4. Palabras tales como "inicializar", "limpiar", etc., implican vinculación temporal.

Los módulos acotados funcionalmente siempre se pueden describir en función de sus elementos usando una sentencia compuesta. Pero si no se puede evitar el lenguaje anterior, siendo aún una descripción completa de la función del módulo, entonces probablemente el módulo no esté acotado funcionalmente.

Es importante notar que no es necesario determinar el nivel preciso de cohesión. En su lugar, lo importante es intentar conseguir una cohesión alta y saber reconocer la cohesión baja, de forma que se pueda modificar el diseño del software para que disponga de una mayor independencia funcional.

1.11.4 Medición de Cohesión

Cualquier módulo, rara vez verifica un solo principio asociativo. Sus elementos pueden estar relacionados por una mezcla de los siete niveles de cohesión. Esto lleva a tener una escala continua en el grado de cohesión más que una escala con siete puntos discretos.

Donde existe más de una relación entre un par de elementos de procesamiento, se aplica el máximo nivel que alcanzan. Por esto, si un módulo presenta cohesión lógica entre *todos* sus pares de elementos de procesamiento, y a su vez presenta cohesión de comunicación también entre *todos* dichos pares, entonces dicho módulo es considerado como de cohesión por comunicación.

Ahora, ¿cuál sería la cohesión de dicho módulo si también contiene algún par de elementos completamente no relacionados? En teoría, debería tener algún tipo de promedio entre la cohesión de comunicación y la casual. Para propósitos de depuración, mantenimiento, y modificación, un módulo se comporta como si fuera "**solo tan fuerte como sus vínculos más débiles**".

El efecto sobre los costos de programación es próximo al menor nivel de cohesión aplicable dentro del módulo en vez del mayor nivel de cohesión.

> *La cohesión de un módulo es aproximada al nivel más alto de cohesión que es aplicable a todos los elementos de procesamiento dentro del módulo.*

Un módulo puede consistir de varias funciones *completas* relacionadas lógicamente. Esto es definitivamente más cohesivo que un módulo que liga lógicamente fragmentos de varias funciones.

La decisión de que nivel de cohesión es aplicable a un módulo dado requiere de cierto juicio humano. Algunos criterios establecidos son:

- La cohesión secuencial es más próxima al óptimo funcional que a su antecesor de comunicación.

- Similarmente existe un salto mayor entre la cohesión lógica y la temporal que entre casual y lógica.

Podemos asignar la siguiente escala de valores para ayudar al diseñador en la calificación de niveles:

- 0: casual
- 1: lógica
- 3: temporal
- 5: procedural
- 7: de comunicación
- 9: secuencial
- 10: funcional

De cualquier modo, no es una regla fija, sino una conclusión.

La obligación del diseñador es conocer los efectos producidos por la variación en la cohesión, especialmente en términos de modularidad, en orden de realizar soluciones de *compromiso* beneficiando un aspecto en contra de otro.

En conclusión se puede decir:

UNA BUENA MODULARIDAD SE OBTIENE SI EXISTE UNA ALTA COHESIÓN Y UN BAJO ACOPLAMIENTO

Ejercicios.

1. La palabra función se emplea en algunos lenguajes como lo es C. Investigar la diferencia de esta palabra en el ambiente de programación y en el ambiente matemático.
2. ¿Cuál será el tamaño óptimo de una función.
3. Explique la importancia de la cohesión para la modularidad.
4. Explique la importancia del acoplamiento para la modularidad.
5. ¿La modularidad tiene alguna relación con el encapsulamiento?
6. ¿La modularidad tiene alguna relación con la abstracción?
7. ¿La modularidad tiene alguna relación con los tipos de datos abstractos?
8. Platique de los sistemas operativos con estructura monolítica, ¿Cuáles eran sus principales desventajas?

II PROGRAMACIÓN RECURSIVA

(Cairó/Gardati, 2000)
(Loomis, 2013)
(Levin, 2004)
(Serie de Fibonacci)

http://www.ugr.es/~eaznar/fibo.htm

El área de la programación es muy amplia y con muchos detalles. En el lenguaje de programación C, así como en otros lenguajes de programación se puede aplicar una técnica que se le dio el nombre de recursividad por su funcionalidad. La asignación de memoria puede ser estática o dinámica y en un momento dado se puede emplear la combinación de estas dos.
La recursividad es una técnica con la que un problema se resuelve sustituyéndolo por otro problema de la misma forma pero más simple.
Por ejemplo, la definición del factorial para n>=0.

O!=1
N!=N*(N-1)! Si n>1

Ciertos problemas se adaptan de manera natural a soluciones recursivas.

2.1 Clasificación de funciones recursivas.

Las funciones recursivas se clasifican según se haga la llamada recursiva en:

- Recursividad directa: La función se llama a sí misma
- Recursividad indirecta: La función A llama a la función B, y la función B llama a A.

Según el número de llamadas recursivas generadas en tiempo de ejecución:

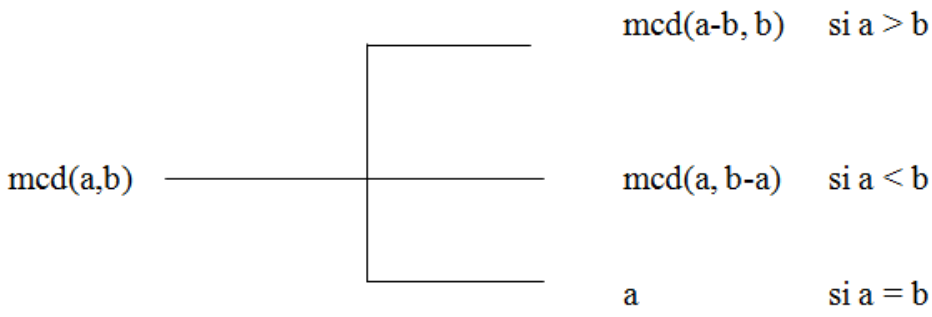
- Función recursiva lineal o simple: Se genera una única llamada interna.
- Función recursiva no lineal o múltiple: Se generan dos o más llamadas internas.

Según el punto donde se realice la llamada recursiva, la función recursiva puede ser:

- Final: (Tail recursión): La llamada recursiva es la última instrucción que se produce dentro de la función.
- No final: (Nontail recursive Function): Se realiza alguna operación al volver de la llamada recursiva.

Las funciones recursivas finales suelen ser más eficientes (en la constante multiplicativa en cuanto al tiempo, y sobre todo en cuanto al espacio de memoria) que las no finales. (Algunos compiladores pueden optimizar automáticamente estas funciones pasándolas a iterativas).

Un ejemplo de recursividad final es el algoritmo de Euclides para calcular el máximo común divisor de dos números enteros positivos:



2.2 Diseño de funciones recursivas

- El problema original se puede transformar en otro problema similar más simple
- Tenemos alguna manera directa de solucionar "problemas triviales"

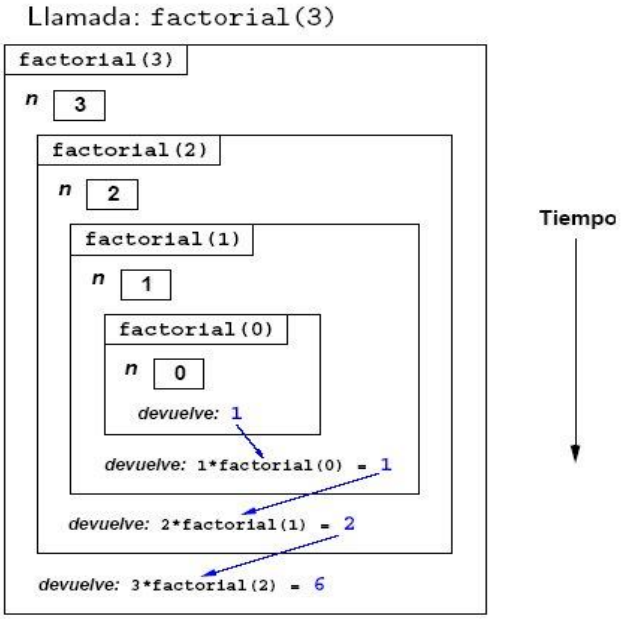
Para que el módulo recursivo sea correcto se debe realizar:

- Un análisis por casos del problema: Existe al menos una condición de terminación en la cual no es necesario una llamada recursiva. Son los casos triviales que se solucionan en forma directa.
  Si n=0 o n=1, el factorial es 1

- Convergencia de la llamada recursiva: Cada llamada recursiva se realiza con un dato más pequeño, de forma que se llegue a la condición de terminación.
  Factorial (n)=n*Factorial (n-1)

- Si las llamadas recursivas funcionan bien, el módulo completo funciona bien: principio de inducción.
  Factorial(0)=1
  Factorial(1)=1
  Para n>1, si suponemos correcto el cálculo del factorial de (n-1),
  Factorial (n)=n*Factorial (n-1)

Gráficamente, la función factorial quedaría:



Un requisito importante para que sea correcto un algoritmo recursivo es que no genere una secuencia infinita de llamadas así mismo.

1.1 Ventajas e inconvenientes de la recursividad.

Una función recursiva debe ser una manera natural, sencilla, comprensible y elegante del problema. Por ejemplo, dado un número entero no negativo, escribir su codificación en binario.

```
void binario(int n){
    if(n<2)
        printf("%d", n);
    else{
        binario(n/2);
        printf ("%d",n%2);
    }
}
```

Otro elemento a tomar en cuenta es la facilidad para comprobar y verificar que la solución es correcta (inducción matemática).

En general, las soluciones recursivas son más ineficientes en tiempo y espacio que las versiones iterativas, debido a las llamadas a subprogramas, la creación de variables dinámicas en la pila recursiva y la duplicación de variables. Otra desventaja es que en algunas soluciones recursivas repiten cálculos en forma innecesaria. Por ejemplo, el cálculo del n-ésimo término de la sucesión de Fibonacci.

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

En general, cualquier función recursiva se puede transformar en una función iterativa.

- Ventaja de la función iterativa: Más eficiente en tiempo y espacio.
- Desventaja de la función iterativa: en algunos casos, muy complicada; además, suelen necesitarse estructuras de datos auxiliares.

Si la eficiencia es un parámetro crítico, y la función se va a ejecutar frecuentemente, conviene escribir una solución iterativa. La recursión se puede simular con el uso de pilas para transformar un programa recursivo en iterativo. Las pilas se usan para almacenar los valores de los parámetros del subprograma, los valores de las variables locales, y los resultados de la función.

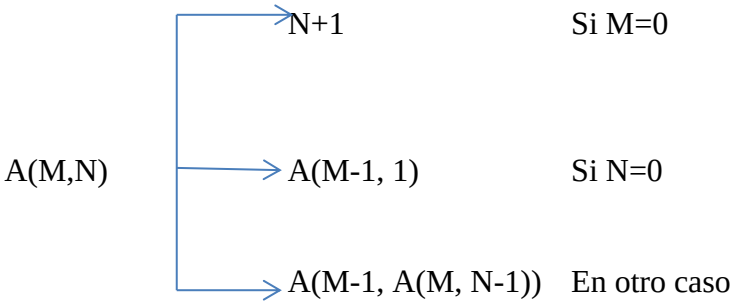
Ejercicios.

1. Realice un árbol recursivo con Fibonacci(5)
2. Realice un árbol recursivo con la función de Hanoi(4,o,d,a)

```
Hanoi(N,Origen, Destino, Auxiliar)
        Si N=1
                Imprime “Mover disco de” Origen “a” Destino
        Sino
                Hanoi(N-1, Origen, Auxiliar, Destino)
```

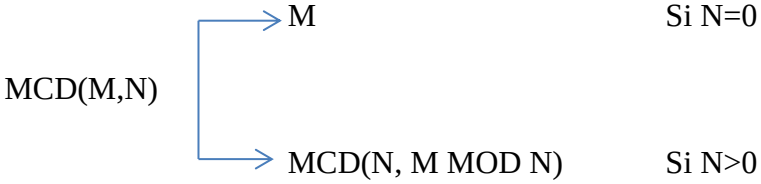
Imprime “Mover disco de” Origen “a” Destino
Hanoi(N-1, Auxiliar, Destino, Origen)

3. La función de Ackermann se define como:



Realizar un árbol recursivo con A(2,2)

4. El algoritmo de Euclides para el cálculo del máximo común divisor se define como:



Realizar un árbol recursivo para MCD(15,4) y MCD(15,3)

5. Realizar un programa que imprima una palabra al revés (no importando su dimensión) sin el uso de arreglos o memoria dinámica.

III INTRODUCCIÓN A LA ALGORITMICA

(Ellis horowitz, 1978)
(Dasgupta, Papadimitriou, & Vazirani, 2008)
(Dewdney, 1989)

Dos ideas cambiaron al mundo. En 1448 en la ciudad de Mainz, un orfebre llamado Johann Gutenberg descubrió el camino para imprimir libros colocando juntas dos piezas metálicas móviles. En este momento se inició la disipación de la edad obscura y el intelecto humano se liberaba, la ciencia y la tecnología habían triunfado, con ello inició la fermentación de la semilla que culminó con la revolución industrial. Varios historiadores indican que nosotros se lo debemos a la tipografía. Imagine a un mundo en el cual sólo una elite podría leer estas líneas. Pero otros insisten que la llave del desarrollo no fue la tipografía, sino que fue la algorítmica.

Fig. 3.1. Johann Gutenberg (1398-1468)

En nuestros días, estamos acostumbrados a escribir números en notación decimal, es fácil de olvidad que Gutenberg, para escribir 1448 utilizaba la notación romana MCDXL + VIII. ¿Cómo se pueden sumar dos números romanos? Lo máximo que podría hacer Gutenberg era sumar números pequeños con los dedos de sus manos; para algo más complicado se tendría que consultar el ábaco.

El sistema decimal, inventado en la India alrededor del año 600 A. C. produjo una revolución en razonamiento cualitativo: usando sólo 10 símbolos, aún números de gran tamaño podrían escribirse sin ninguna complicación. Cualquier operación aritmética se puede realizar sin ninguna complicación. No obstante, estas ideas tardaron bastante tiempo en expandirse, debido a la ignorancia, tradiciones, distancia y la barrera del lenguaje. El medio de mayor influencia para permitir la transmisión de éstos conocimientos se realizó por medio de un libro, escrito por un árabe del siglo noveno que vivió en Bagdad. Al Khwarizmi. Él dio los métodos básicos para la suma, resta, multiplicación, división e inclusive para la raíz cuadrada y el cálculo de $\prod$. Los procedimientos fueron precisos, no ambiguos, mecánicos, eficientes y correctos, fueron algoritmos, un término acuñado en honor al hombre sabio. El sistema decimal fue adoptado en Europa varios siglos después.

Desde la aparición de la notación decimal en Europa se permitió que la tecnología, la ciencia, el comercio, la industria, y posteriormente el computo, se desarrollaran plenamente. Científicos de todo el mundo desarrollaron cada vez algoritmos más complejos para todo tipo de problemas e investigaron novedosas aplicaciones que han cambiado al mundo en forma radical.

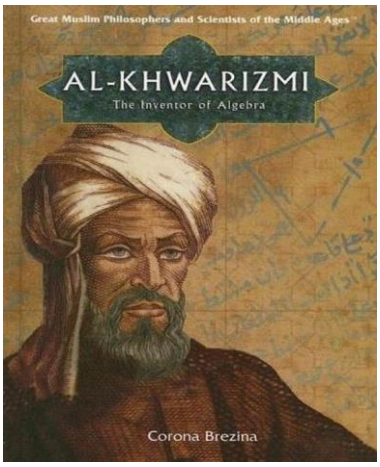


Figura 3.2 Al Khwarizmi
(Vivió entre el año 780 y 850 D. C)

El trabajo de Al Khwarizmi se pudo establecer en Europa por el esfuerzo de un matemático italiano del siglo XIII, conocido como Leonardo Fibonacci, quien descubrió el potencial del sistema posicional y trabajó bastante para el desarrollo y futura propagación.
Pero, hoy en día, Fibonacci es más conocido por la famosa secuencia de números:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

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

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

Fig. 3.3. Leonardo de Pisa (1170-1250)

No existe otra secuencia de números que haya sido estudiada en forma tan extensa, o aplicada a más campos del conocimiento: biología, demografía, arte, arquitectura, música, para nombrar algunos de los campos. Y junto con la potencia de dos, es en ciencias de la computación una de las secuencias favoritas.

En realidad, la secuencia de Fibonacci crece tan rápido como la potencia de dos: por ejemplo, F_{30} rebasa el millón, y la secuencia F_{100} tiene aproximadamente 21 dígitos de largo. En general, $F_n \approx 2^{0.694n}$.

Peor, ¿Cuál es el valor exacto de F_{100} o de F_{200} ? Fibonacci nunca supo la respuesta. Para saberlo, necesitamos un algoritmo para computar el n-ésimo número de Fibonacci.
Una idea es utilizar la definición recursiva de F_n. El pseudocódigo se muestra enseguida:

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

Todas las veces que se tiene un algoritmo, existen preguntas que se deben de responder:

1. ¿Es correcto?
2. ¿Cuánto tiempo tarda en dar el resultado, cómo una función de n?
3. ¿Se puede mejorar?

La primera pregunta no se discute aquí ya que el algoritmo es precisamente la definición de Fibonacci para F_n. Pero la segunda definición demanda una respuesta. Sea T(n) el número de pasos computacionales necesarios para computar fib(n); ¿qué podemos decir sobre ésta función? Si n < 2, el procedimiento termina casi en forma inmediata, justo después de un par de pasos. Por lo tanto:

$T(n) \leq 2$ para $n \leq 1$

Para números mayores existen invocaciones recursivas de fib( ), así tenemos:

T(0)=1, la revisión del primer condicional
T(1)=2, la revisión de dos condicionales
T(2)= La revisión del primer y segundo condicional y la invocación a función.
T(2)= 3 + T(1) + T(0)
T(2)=T(1)+T(0)+3=1+2+3=5
T(3)= La revisión de los dos condicionales mas la invocación recursiva de T(2)+T(1)
T(3)=T(2) + T(1)+3=5+2+3=10
T(4)=T(3)+T(2)+3=10+5+3=18
…
T(n) = T(n-1) + T(n-2) + 3 para n > 1.

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

Una demostración de la complejidad del algoritmo se puede observar de la siguiente forma:
Veamos una aproximación ingenua al cálculo de la complejidad de la función recursiva de Fibonacci. Si llamamos S(n) al número de sumas necesarias para hallar F(n). Para los primeros valores se tiene:
S(1)=0=S(2), S(3)=1, S(4)=2, S(5)=4, S(6)=7,…

Y en general por inducción, el número de sumas para calcular F(n) es igual a
$S(n)=S(n-1)+S(n-2)+1$

Por inducción se obtiene
$F(n-2)<S(n)$
Pero ¿Qué tan rápido crece la función de Fibonacci?
Podemos hacer una analogía
$F(2)=F(1)+F(0)$
$X^2=X+1$
O sea, las raíces de $X^2-X-1=0$
Siendo esta una ecuación característica y una de sus raíces se conoce como el número de oro (Golden Ratio), y su valor exacto es $c=(1+\sqrt{5})/2$.
La progresión geométrica $\{c^n\}$ satisface la misma ecuación en recurrencia que la función de Fibonacci, esto es $c^n=c^{n-1}+c^{n-2}$ tiene una equivalencia con $F(n)=F(n-1)+F(n-2)$. Ahora, por inducción
Como $c^0=1=F(2)$, $c=c^1<2=F(3)$, obtenemos que $c^{n-2}<F(n)$
En conclusión, la función de Fibonacci crece, como mínimo, exponencialmente.
Enlazando las dos desigualdades, para toda n se tiene:

$C^{n-4} < F(n-2)<S(n)$
Y también la función S(n) crece, como mínimo, exponencialmente.

Por ejemplo, para calcular F_{200} , la función fib( ) se ejecuta $T(200) \geq F_{200} \geq 2^{138}$ pasos elementares de cómputo. ¿Cuánto tiempo se requiere? Bueno, eso depende de la computadora usada. En este momento, la computadora más veloz en el mundo es la NEC Earth Simulator, con 400 trillones de pasos por segundo. Aún para ésta máquina, fib(200) tardará al menos 2^{92} segundos. Esto significa que, si nosotros iniciamos el cómputo hoy, estaría trabajando después de que el sol se torne una estrella roja gigante.

Pero la tecnología se ha mejorado de tal forma que los pasos de computo se han estado en forma aproximada duplicando cada 18 meses, a tal fenómeno se le conoce la ley de Moore. Con éste extraordinario crecimiento, posiblemente la función fib se ejecutará en forma mucho más rápida para el próximo año. Chequemos éste dato, El tiempo de ejecución de $fib(n) \approx 2^{0.694n} \approx (1.6)^n$, por lo que toma 1.6 veces más tiempo para computar F_{n+1} que F_n . Y bajo la ley de Moore, el poder de computo crece aproximadamente 1.6 veces cada año. Si nosotros podemos computar en forma razonable F_{100} con el crecimiento de la tecnología, el siguiente año se podrá calcular F_{101} . y el siguiente año, F_{102} y así sucesivamente: Solo un número de Fibonacci más cada año. Así es el comportamiento de un tiempo exponencial.
En corto, nuestro algoritmo recursivo es correcto pero sumamente ineficiente. ¿Podemos hacerlo mejor?

El algoritmo se vuelve ineficiente por la razón de que una llamada a fib(n) se tiene una cascada de llamadas recursivas en las cuales la mayoría de ellas son repetitivas.

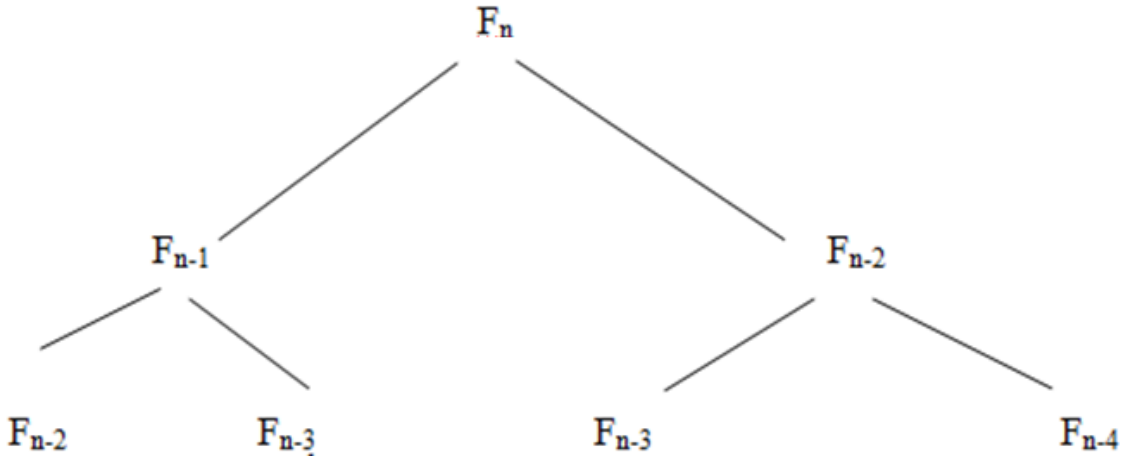


Fig. 1.4. Árbol recursivo con la función de Fibonacci.

Se muestra en lenguaje C un algoritmo mucho más sencillo:

```
#include <stdio.h>
main(){
      int i, n=5, fibn_2,fibn_1, fibn;

      fibn_2=fibn_1=1;
      for ( i =2;i<=n; i++){
         fibn=fibn_2+fibn_1;
         printf ("fibo (%d) =%d\n", i , fibn);
         fibn_2=fibn_1;
         fibn_1=fibn;
      }
      getchar ();
}
```

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

IV COMPLEJIDAD Y ORDEN

(Ellis horowitz, 1978)
(Dasgupta, Papadimitriou, & Vazirani, 2008)
(Diccionario de la Real Academia Española)
(Complejidad Algorítmica)

¿Qué es un algoritmo?
El Diccionario de la Real Academia Española lo define como:

> *Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.*

En cómputo se define como:

Un método preciso usado por una computadora para la solución de problemas.

Un algoritmo está compuesto de un conjunto finito de pasos, cada paso puede requerir una o más operaciones. Cada operación debe estar definida. Cada paso debe ser tal que deba, al menos, ser hecha por una persona usando lápiz y papel en un tiempo finito.
Existe una gran diferencia entre receta y algoritmo:

Receta: Colocar sal al gusto.
Algoritmo: Debe de indicarse cada paso con exactitud.

Otra palabra que obedece a casi todo lo indicado es "proceso computacional". Un ejemplo es el Sistema Operativo. Este proceso está diseñado para controlar la ejecución de trabajos, en teoría, el proceso nunca termina ya que se queda en estado de espera hasta que la solicitud de otro trabajo llegue.

El estudio de algoritmos incluye varias y activas áreas de la investigación:

a. Cómo elaborar algoritmos.
b. Cómo expresarlos.
c. Cómo validarlos.
d. Cómo analizarlos.
e. Cómo probarlos. El debugging sólo nos indica la presencia de errores, no la ausencia de ellos. También se debe de evaluar a un algoritmo en forma temporal como en forma espacial.

Al analizar un algoritmo, lo primero es verificar cuales operaciones son empleadas y su costo.

Estas operaciones se pueden acotar en un tiempo constante. Ejemplo, la comparación entre caracteres se puede hacer en un tiempo fijo.
La comparación de las cadenas depende del tamaño de las cadenas. (No se puede acotar el tiempo).
Para ver los tiempos de un algoritmo se requiere un análisis a priori y no a posteriori. En un análisis a priori se obtiene una función que acota el tiempo del algoritmo. En un análisis a posteriori se colecta una estadística sobre el desarrollo del algoritmo en tiempo y espacio al momento de la ejecución de tal algoritmo.

Un ejemplo de un análisis a priori es:

x ← x+y	para i ← 1 a n x ← x+y	para i ← 1 a n para j ←1 a n x ← x+y
1	**n**	**n^2**

El análisis a priori ignora qué tipo de máquina es, el lenguaje, y sólo se concentra en determinar el orden de magnitud de la frecuencia de ejecución.

Notación $\mathcal{O}$

F(n) = $\mathcal{O}$(g(n)) iff existen 2 constantes "c" y " n_o" tal que
$|f(n)| \leq c|g(n)|$ para toda $n > n_o$.

Cuando se dice que un algoritmo tiene un tiempo computacional $\mathcal{O}$(g(n)), indica que el algoritmo se corre en una computadora x con el mismo tipo de datos pero con n mayor, el tiempo será menor que algún tiempo constante |g(n)|.

Si $A(n) = a_m n^m + \ldots + a_1 n + a_o$ entonces
$A(n) \approx \mathcal{O}(n^m)$

Siendo A(n) el número de pasos de un algoritmo determinado.
Si un algoritmo tiene x pasos cuyo orden de magnitud es c_1n^{m1}, c_2n^{m2}, c_3n^{m3}, . . ., c_kn^{mk},
Entonces el orden del algoritmo es:

$\mathcal{O}(n^m)$ donde $m = \max\{m_i\}$, $1 \leq i \leq k$.

Como ejemplo para comprender el orden de magnitud suponga que se tienen dos algoritmos para la misma tarea en el cual requieren uno del $\mathcal{O}(n^2)$ y el otro del
$\mathcal{O}$(n log n). Si n = 1024 se requiere para el primer algoritmo 1048576 operaciones y para el segundo algoritmo son 10241 operaciones. Si la computadora toma un μseg para realizar cada operación, el algoritmo uno requiere aproximadamente 1.05 segundos y el segundo algoritmo requiere aproximadamente 0.0102 segundos para la misma entrada.
Los tiempos más comunes son:
$\mathcal{O}(1) < \mathcal{O}(\log n) < \mathcal{O}(n) < \mathcal{O}(n \log n) < \mathcal{O}(n^2) < \mathcal{O}(n^3) < \mathcal{O}(2^n)$.

Nota: La base del logaritmo es dos. ($Log_b N = Log_a N / Log_b b$)
$\mathcal{O}(1)$ El número de operaciones básicas es fijo por lo que el tiempo se acota por una constante.
$\mathcal{O}(n)$, $\mathcal{O}(n^2)$ y $\mathcal{O}(n^3)$ son del tipo polinomial.
$\mathcal{O}(2^n)$ es de tipo exponencial.

Los algoritmos con complejidad mayor a $\mathcal{O}(n \log n)$ a veces son imprácticos.
Un algoritmo exponencial es práctico sólo con un valor pequeño de n.

Ejemplo:

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

Tabla 4.1 Complejidad algorítmica.

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

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

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

Tabla 4.2 Tiempo en segundos que tarda en realizar f(n) operaciones.

En la figura 4.1 se puede observar gráficamente el crecimiento de las principales funciones de complejidad temporal.

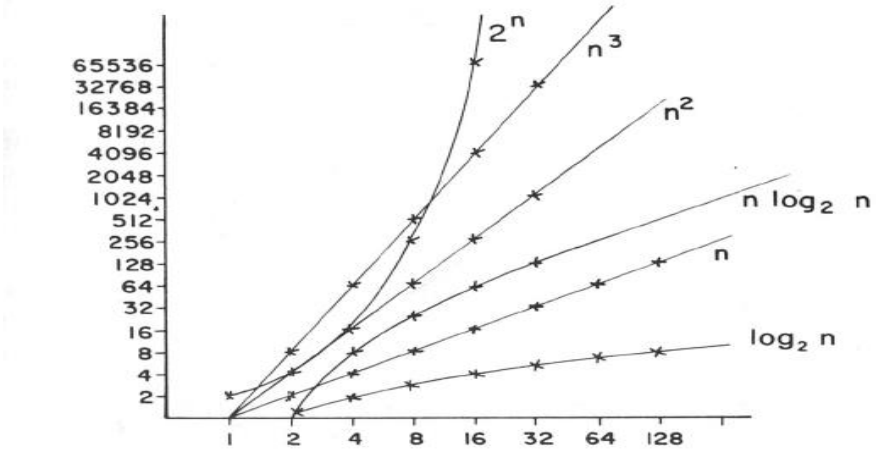


Figura 4.1

La notación $\mathcal{O}$ (o gran $\mathcal{O}$) sirve para indicar una cota superior. También se puede definir una cota inferior.

$F(n) = \Omega(g(n))$ iff existe una constante "c" y " n_o" tal que

$| f(n)| \geq c|g(n)|$ para toda $n > n_o$

Si $F(n) = \Omega(g(n))$ y $f(n) = \mathcal{O}(g(n))$ entonces:

$F(n) = \Theta(g(n))$ iff existen constantes positivas c y n_o tal que para toda $n > n_o$,

$C_1 |g(n)| \leq |f(n)| \leq C_2|g(n)|$

Esto indica que el peor y el mejor caso marcan la misma cantidad de tiempo.

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

$\Theta(n)$.

Buscar un arreglo un valor

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

Ejercicios

1. Dado el algoritmo de las Torres de Hanoi mostrado en el capítulo de recursividad, determinar su orden. Suponiendo que se tienen 63 anillos.
2. Investigue la complejidad del algoritmo de Ackermann y compárelo con el algoritmo de las Torres de Hanoi.

V ORDENACIÓN Y BUSQUEDA

(Cairó/Gardati, 2000)
(Loomis, 2013)
(Ellis horowitz, 1978)
(Método de la Burbuja)

5.1 Método de la burbuja.

Es el algoritmo más sencillo. Ideal para empezar. Consiste en ciclar repetidamente a través de una lista, comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian.
El algoritmo en pseudocódigo en C es:

```
for (i=1; i<TAM; i++)
          for j=0 ; j<TAM - 1; j++)
               if (lista[j] > lista[j+1])
                    temp = lista[j];
                    lista[j] = lista[j+1];
                    lista[j+1] = temp;
```

Dónde:
lista: Es cualquier lista a ordenar
TAM: Es una constante que determina el tamaño de la lista.
i, j: Contadores
temp: Permite realizar los intercambios de la lista

Vamos a ver un ejemplo. Esta es nuestra lista:

4 - 3 - 5 - 2 - 1

Tenemos 5 elementos. Es decir, TAM toma el valor 5. Comenzamos comparando el primero con el segundo elemento. 4 es mayor que 3, así que intercambiamos. Ahora tenemos:

3 - **4** - 5 - 2 - 1

Ahora comparamos el segundo con el tercero: 4 es menor que 5, así que no hacemos nada. Continuamos con el tercero y el cuarto: 5 es mayor que 2. Intercambiamos y obtenemos:

3 - 4 - **2** - **5** - 1

Comparamos el cuarto y el quinto: 5 es mayor que 1. Intercambiamos nuevamente:

3 - 4 - 2 - **1** - **5**

Repitiendo este proceso vamos obteniendo los siguientes resultados:

3 - 2 - 1 - 4 - 5

2 - 1 - 3 - 4 - 5

1 - 2 - 3 - 4 – 5

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

- Estabilidad: Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto es *estable*.
- Requerimientos de Memoria: Este algoritmo sólo requiere de una variable adicional para realizar los intercambios.
- Tiempo de Ejecución: El ciclo interno se ejecuta **n** veces para una lista de n elementos. El ciclo externo también se ejecuta **n** veces. Es decir, la complejidad es $\mathcal{O}(n^2)$. El comportamiento del caso promedio depende del orden de entrada de los datos, pero es sólo un poco mejor que el del peor caso, y sigue siendo $\mathcal{O}(n^2)$.

Ventajas:

- Fácil implementación.
- No requiere memoria adicional.

Desventajas:

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

Cálculo de la complejidad:

CODIGO	COSTO	VECES A EJECUTAR
`for (i=1; i<n; i++)`	C_1	N
`for j=0 ; j<n - 1; j++)`	C_2	N^2
`if (lista[j] > lista[j+1])`	C_3	N^2
`temp = lista[j];`	C_4	N^2
`lista[j] = lista[j+1];`	C_5	N^2
`lista[j+1] = temp:`	C6	N^2

Por lo que:

$T(n)=N + 5*N^2$ por lo que la complejidad es $O(n^2)$.

5.2 Ordenación por selección directa

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

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

El algoritmo en código C es:

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

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

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

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

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

$\underline{S_n = n+ (n-1)+(n-2)+(N-3)+\ldots+ 4+ 3+ 2+ 1}$

$2S_n=(n+1)+(n+1)+(n+1)+(n+1)+\ldots+(n+1)+(n+1)+(n+1)+(n+1)$

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

Utilizando la misma idea se tiene:

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

$\underline{S_n = (n-1)+(n-2)+(N-3)+(n-4)+\ldots + 2+ 1}$

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

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

De esta forma se tiene:

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

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

5.3 Método de inserción binaria

El método de ordenación por inserción binaria realiza una búsqueda binara el lugar de una búsqueda secuencial, para insertar un elemento en la parte izquierda del arreglo, que ya se encuentra ordenado. El proceso se repite desde el segundo hasta el n-ésimo elemento.

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

Al analizar el método de ordenación por inserción binaria se advierte la presencia de un caso antinatural. El método efectúa el menor número de comparaciones cuando el arreglo está totalmente desordenado y el máximo cuando se encuentra ordenado.
Es posible suponer que mientras en una búsqueda secuencial se necesitan K comparaciones para insertar un elemento, en una binaria se necesita la mitad de las K comparaciones. Por lo tanto, el número de comparaciones promedio en el método de ordenación por inserción binaria se puede calcular como:

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

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

5.4 Método de ordenación rápida (quicksort)

El método de ordenación quicksort es actualmente el más eficiente y veloz de los métodos de ordenación interna. Este método es una mejora sustancial por la velocidad con que ordena los

elementos del arreglo. Su autor, C. A. Hoare, lo llamó así. La idea central de este algoritmo consiste en lo siguiente:

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

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

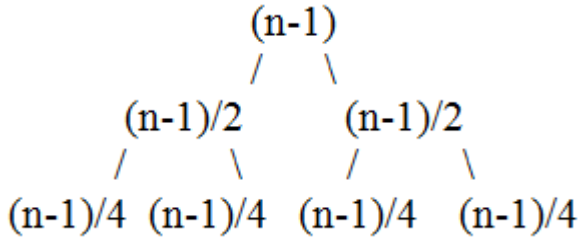
```
#include <stdio.h>
#define N       10
void quicksort(int [], int, int);
main(){
int a[]={10,8,7,2,1,3,5,4,6,9},i;
quicksort(a,0,N-1);
for (i=0;i<10;i++)
        printf("%d..",a[i]);
putchar('\n');
getchar();
}

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

```
                izq++;
            if (pos!=izq){
                band=1;
                aux=a[pos];
                a[pos]=a[izq];
                a[izq]=aux;
                pos=izq;
            }
        }
}
if (pos-1>ini)
        quicksort(a, ini, pos-1);
if (fin>pos+1)
        quicksort(a,pos+1,fin);
}
```

El método quicksort es el más rápido de ordenación interna que existe en la actualidad. Esto es sorprendente, porque el método tiene su origen en el método de intercambio directo, el peor de todos los métodos directos. Diversos estudios realizados sobre su comportamiento demuestran que si se escoge en cada pasada el elemento que ocupa la posición central del conjunto de datos a analizar, el número de comparaciones, si el tamaño del arreglo es **una potencia de** 2, en la primera pasada realiza (n-1) comparaciones, en la segunda (n-1)/2 comparaciones, pero en dos conjuntos diferentes, en la tercera realizará (n-1)/4 comparaciones, pero en cuatro conjuntos diferentes y así sucesivamente, esto produce un árbol binario recursivo. Por lo tanto:

<u>Árbol de recursividad</u>



<u>K. indica el nivel del árbol</u>
K=0

K=1. La suma es: 2*(n-1)/2

K=2. La suma es: 4*(n-1)/4

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

Lo cual es lo mismo que:
C=(n-1)+(n-1)+…+(n-1)

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

C=(n-1)*k

Considerando que el número de términos de la sumatoria (k) es el número de niveles del árbol binario, el número de elementos del arreglo se puede definir como $2^k=n$, por lo que $\log_2 2^k=\log_2 n$, $k \log_2 2 = \log_2 n$ (recordando que $\log_m m=1$), $k =\log_2 n$, por lo que la expresión anterior queda como:

C=(n-*1)* log *n*

Sin embargo, encontrar el elemento que ocupe la posición central del conjunto de datos que se van a analizar es una tarea difícil, ya que existen 1/n posibilades de lograrlo. Además, el rendimiento medio del método es aproximadamente (2*ln 2) inferior al caso óptimo, por lo que Hoare, el autor del método, propone como solución que el elemento X se seleccione arbitrariamente o bien entre una muestra relativamente pequeña de elemento del arreglo.
El peor caso ocurre cuando los elementos del arreglo ya se encuentran ordenados, o bien cuando se encuentran ordenados en forma inversa. Supongamos que se debe ordenar el siguiente arreglo unidimensional que ya se encuentra ordenado:
A: 08 12 15 16 27 35 44 67
Si se escoge arbitrariamente el primer elemento (08), entonces se particionará el arreglo en dos mitades, una de 0 y la otra de (n-1) elementos.
Si se continua con el proceso de ordenamiento y se escoge de nuevo el primer elemento (12) del conjunto de datos que se analizará, entonces de dividirá el arreglo en dos nuevos subconjuntos, nuevamente uno de 0 y otro de (n-2) elementos. Por lo tanto, el número de comparaciones que se realizará será:

$$C_{máx}=n+(n-1)+(n-2)+\ldots+2=n*(n-1)/2-1$$

Que es igual a:

$$C_{máx}=(n^2+n)/2-1$$

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

5.5 Método de mezcla (merge sort)

Éste algoritmo ordena elementos y tiene la propiedad de que el peor caso en complejidad será: $\mathcal{O}(n \log n)$. Los elementos van a ser ordenados en forma creciente. Dados n elementos, éstos se dividirán en 2 subconjuntos. Cada subconjunto será ordenado y el resultado será unido para producir una secuencia de elementos ordenados. El código en C es el siguiente:

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

void mergesort(int a[],int low, int high){
    int mid;
    if (low<high){
```

```
            mid=(low+high)/2;
            mergesort(a,low,mid);
            mergesort(a,mid+1,high);
            merge(a,low,mid,high);
    }
}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

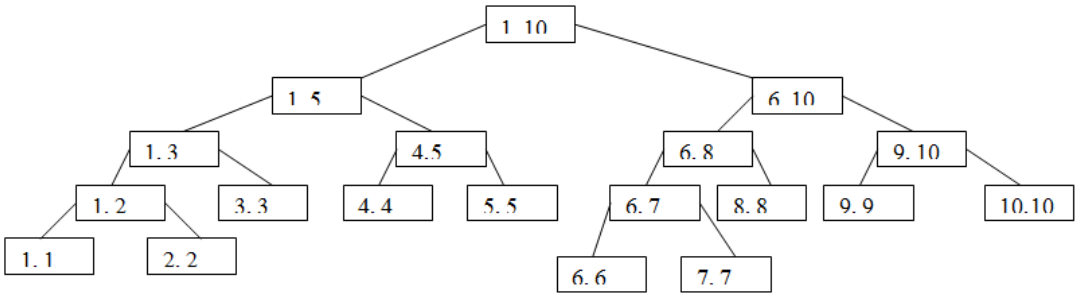


Fig. 5.1 Árbol recursivo para el algoritmo MergeSort

La figura 5.1 muestra la secuencia de recursividades producidas por MERGESORT con los diez elementos. Note que la división continúa hasta contener un simple elemento.

El tiempo de cómputo para mergesort se describe a continuación:

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

2T(n/2) se refiere a las dos llamadas recursivas realizadas por MERGESORT.

C_n se refiere a la función MERGE en la cual sólo tiene ciclos no anidados cuya complejidad es lineal.

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

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

$T(n)= 8T(n/8) + C_n +2 C_n = 8T(n/8) + 3 C_n \ldots$

$T(n)= 2^kT(1) + K C_n = an +K C_n$ //Sabemos que $n = 2^k$ y $T(1) = a$

$T(n)=an + C_n \log n$ // Sabemos que $n = 2^k$ por lo tanto $k=\log_2 n$.

Si $2^k < n \leq n = 2^{k+1}$, entonces $T(n) \leq T(2^{k+1})$. Por lo tanto:

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

Este algoritmo es un ejemplo clásico del paradigna “divide y vencerás” que se explica posteriormente.

5.6 Búsqueda Secuencial

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

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

Si hubiera dos o más ocurrencias del mismo valor, se encuentra la primera de ellas. Sin embargo, es posible modificar el algoritmo para obtener todas las ocurrencias de datos buscados.
A continuación se presenta una variante de este algoritmo, pero utilizando recursividad, en lugar de interactividad.

```
#include <stdio.h>
#define N 10
void secuencial(int [], int,int,int);
main(){
    int x,i=0,a[N]={9,7,10,8,2,4,6,5,1,3};
    scanf("%d",&x);
    secuencial(a,N,x,0);
    getchar();
    getchar();
    }
void secuencial(int a[],int n, int x, int i){
        if (i>n-1)
            printf("Dato no localizado\n");
```

```
        else if (a[i]==x)
                printf("dato localizado en la posicion %d\n",i);
        else
                secuencial(a,n,x,i+1);
}
```

El número de comparaciones es uno de los factores más importantes para determinar la complejidad de los métodos de búsqueda secuencial, se deben establecer los casos más favorables o desfavorables que se presenten.
Al buscar un elemento en el arreglo unidimensional desordenado de N componentes, puede suceder que ese valor no se encuentre; por lo tanto, se harán N comparaciones al recorrer el arreglo. Por otra parte, si el elemento se encuentra en el arreglo, éste puede estar en la primer posición o en la última o en alguna intermedia. Por lo que.

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

Por lo que el algoritmo tiene un $\mathcal{O}(n)$.

5.7 Búsqueda binaria (Binary Search).

La búsqueda binaria consiste en dividir el intervalo de búsqueda en dos partes, comparando el elemento buscado con el que ocupa la posición central en el arreglo. Para el caso de que no fueran iguales se redefinen los extremos del intervalo, según el elemento central sea mayor o menor que el elemento buscado, disminuyendo de esta forma el espacio de búsqueda. El proceso concluye cuando el elemento es encontrado, o cuando el intervalo de búsqueda se anula, es vacío.
El método de búsqueda binaria funciona exclusivamente con arreglos ordenados. No se puede utilizar con listas simplemente ligadas ni con arreglos desordenados. Con cada iteración del método el espacio de búsqueda se reduce a la mitad; por lo tanto, el número de comparaciones a realizarse disminuye notablemente. Esta disminución resulta significativa cuanto más grande sea el tamaño del arreglo. El algoritmo es el siguiente:

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

```
        printf("Elemento localizado en el lugar %d.\n",j);
    getchar();
    getchar();
}
```

¿Es Busqueda Binaria un algoritmo?

Se debe verificar si las comparaciones entre x y A(mid) están bien definidas.

¿Binsrch termina?

¿Qué tipo de variables se manejan?

Por ejemplo: Se tiene el siguiente arreglo con los siguientes números:

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

En el peor de los casos se requieren 4 comparaciones.

El promedio de las comparaciones es 2.77

El óptimo es: una comparación.

Si no se encuentra un elemento:

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

El promedio es 3.4.

Éste análisis trabaja para 9 elementos, pero ¿y para cualquier n?

Por ejemplo, para n=14 se tiene:

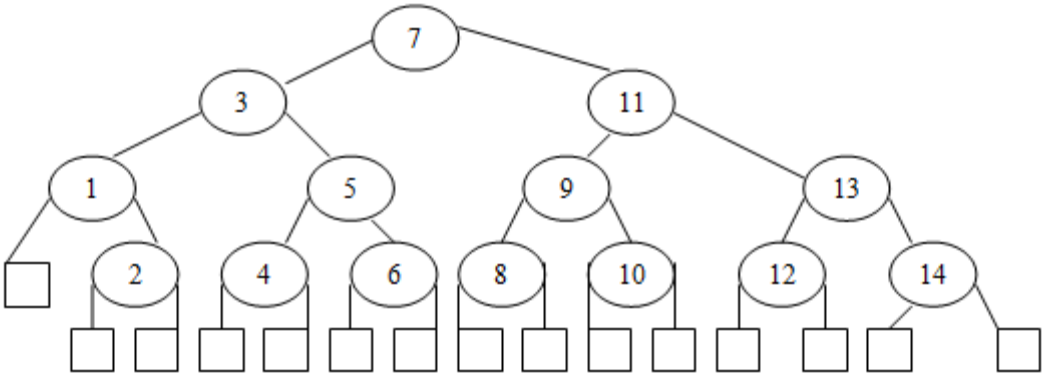


Fig. 5.2 Algoritmo Busqueda Binaria con 14 elementos.

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

PARADIGMAS.

VI DIVIDE Y CONQUISTARAS.

(Ellis horowitz, 1978)
(Dasgupta, Papadimitriou, & Vazirani, 2008)
(Abellanas & Lodares, 1990)
(Goldshlager & Lister, 1986)

Dada una función para computar n entradas, la estrategia divide y conquistaras sugiere dividir la entrada en k subconjuntos, $1< k \leq n$ produciendo k subproblemas. Estos subproblemas deben ser resueltos y entonces un método debe ser encontrado para combinar las subsoluciones dentro de una solución como un todo. Si el subproblema es aún demasiado grande, entonces puede ser reaplicada la estrategia. Frecuentemente los subproblemas resultantes de un diseño divide y conquistaras son del mismo tipo del problema original. El principio divide y conquistaras se expresa en forma natural por un procedimiento recursivo. Por lo que se obtienen subproblemas de menor dimensión de la misma clase, eventualmente se producen subproblemas que son lo suficientemente pequeños que son resueltos sin la necesidad de dividirlos.
Una forma general de ver el modelo es:

```
void DANDC(int a[],p,q)
//a[] es el arreglo a trabajar
//p y q son los subespacios a dividir
int m,p,q;
if small(p,q)
  return(G((p,q))
else{
    m=divide(p,q);
    return(conbinacion(DANDC(a,p,m), DANDC(a,m+1,q))
}
}
```

En este caso, la función small determina si se cumple una condición especifica para que el algoritmo se detenga, si no es así se crean dos subespacios en los cuales se puede subdivide el espacio en dos más pequeños.

DANDC se puede describir por la relación recurrente:

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

f(n) es el tiempo que se dedica para calcular las funciones DIVIDE y COMBINE.

Tres algoritmos vistos con anterioridad pertenecen a este paradigma:

- Busqueda binaria (Binary Search)
- Mezcla (Merge Sort)
- Ordenación rápida (Quick Sort)

Otro ejemplo es el siguiente algoritmo:

6.1 Buscando el Máximo y Mínimo (Finding The Maximun and Minimun).

El problema es encontrar el máximo y el mínimo de un conjunto de n elementos en desorden.
Un algoritmo directo sería:

```
#include <stdio.h>
main(){
    int max,min,i,j,a[]={4,2,1,5,7,9,8,6,3,10};
    max=min=a[0];
    for(i=1;i<10;i++){
                if (a[i]>max)
                  max=a[i];
                if (a[i]<min)
                   min=a[i];
    }

    printf("El maximo valor es %d y el minimo valor es %d\n",max,min);
    getchar();
}
```

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

```
                if (a[i]>max)
                  max=a[i];
                else if (a[i]<min)
                   min=a[i];
```

Ahora el mejor caso es cuando los elementos están en forma creciente ya que en el mejor de los casos se requier n-1 comparaciones en el mejor de los casos y en el peor de los casos se requieren 2(n-1) comparaciones. El promedio será.

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

A continuación se muestra un algoritmo recursivo que encuentra el máximo y el mínimo de un conjunto de elementos y maneja la estrategia de divide y conquistaras. Este algoritmo envía cuatro parámetros, los dos primeros se manejan como paso de parámetros por valor y los dos últimos se manejan como pase de parámetros por referencia. En este caso, el segundo y tercer parámetro indican el subconjunto a analizar y los dos últimos parámetros sirven para retornar el mínimo y máximo de un subconjunto determinado. Al término de la recursión se obtienen el mínimo y máximo del conjunto dado. Se muestra el algoritmo en C:

```
#include <stdio.h>
void MaxMin(int [],int, int, int *, int *);
int max(int, int);
int min(int, int);
main(){
      int fmax,fmin,a[]={4,2,10,5,-7,9,80,6,3,1};
      MaxMin(a,0,9,&fmax,&fmin);
      printf("El maximo valor es %d y el minimo valor es %d\n",fmax,fmin);
      getchar();
}

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

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

}
```

Ejemplo:

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

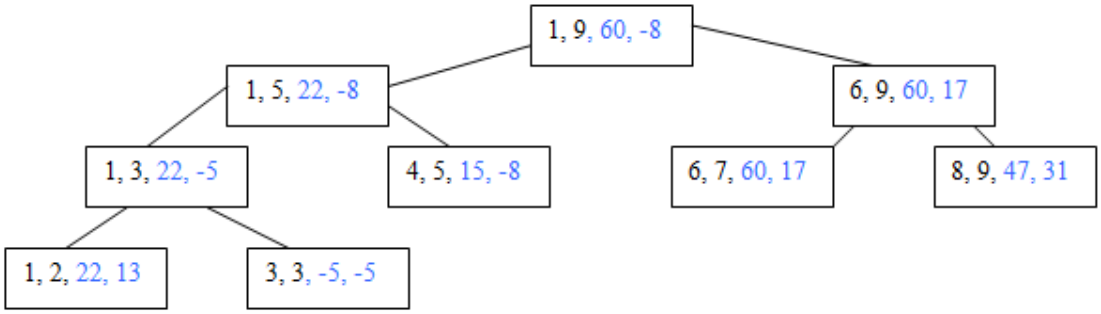


Fig. 6.1 Árbol recursivo con el algoritmo MaxMin.

¿Cuantas comparaciones se requieren?

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

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

Por lo tanto

$T(n) = 2T(n/2) +2 = 2(2T(n/4)+2)+2 = 4T(n/4) + 4 +2 = 4(2T(n/8)+2) + 4 + 2$
$T(n) = 8T(n/8)+8 + 4 + 2 = 8(2T(n/16)+2) + 8 + 4 + 2 = 16T(n/16)+16+8+4+2 \ldots$
$T(n) = 2^{k-1} T(2) + \sum 2^i = 2^{k-1} + 2^k - 2 = 3n/2 - 2$

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

Si n es igual a 16 se tiene:
$T(16)=2T(8) + 2 = 2(2T(4)+2) + 2 = 4T(4) + 4 + 2 = 4(2T(2)+2) +4 +2$
$T(16) = 8T(2) + 8 + 4 + 2$
Donde $n=2^4$, K = 4 y T(2)=1; por lo tanto:

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

Observe que 3n/2 – 2 es el mejor promedio y peor caso si n es poder de 2. Comparado con 2n-2 comparaciones existe un ahorro del 25%. Por lo que es mejor que el secuencial. Pero:
¿Esto indica que sea mejor en la práctica? No necesariamente. En términos de almacenamiento es peor ya que requiere una pila para guardar a i , j, fmax, fmin. Dados n elementos se requieren log n +1 niveles de recursión. Se requieren guardar 5 valores y el direccionamiento de retorno. Por lo MaxMin es más ineficiente ya que se maneja una pila y recursión.

VII MÉTODO CODICIOSO. (Greedy)

(Ellis horowitz, 1978)
(Dasgupta, Papadimitriou, & Vazirani, 2008)

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

El método greedy sugiere que uno puede hacer un algoritmo que trabaje por pasos, considerando una entrada a la vez. En cada paso se realiza una decisión. Si la inclusión de la siguiente entrada da una solución no factible, la entrada no se adiciona a la solución parcial. Enseguida se muestra en pseudocódigo la estrategia greedy:

```
Procedure Greedy.
//A(1:n) contiene n entradas.
solución ← φ
for i ← to n do
        x ← select (A)
        if feasible(solución, x) then
                Solución ← UNION(Solución, x)
        endif
repeat
return (solución)
end Greedy.
```

7.1. Almacenamiento Óptimo en Cintas. (Optimal Storage On Tapes)

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

Ejemplo:
N=3 (I_1, I_2, I_3) =(5, 10, 3). Existen
N!=6 posibles ordenaciones.

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

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

7.2 El Problema de la Mochila. (Knapsack Problem)

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

El problema es:

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

S. A.

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

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

Una posible solución es cualquier conjunto $(X_1, X_2, \ldots, X_n)$ que satisfaga (2) y (3). Una solución óptima es una solución factible en el cual maximice la ganancia (1).
Ejemplo: n=3, M=20, $(P_1, P_2, P_3) = (25, 24, 15)$
$(W_1, W_2, W_3) = (18, 15, 10)$

Cuatro posibles soluciones son:

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

De las cuatro soluciones, la cuarta produce un máximo
Toda solución óptima llena la mochila al máximo.
Se pueden tener tres estrategias:

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

La estrategia (c) produce un óptimo y requiere sólo $\mathcal{O}(n)$.
El siguiente algoritmo localiza el óptimo siempre y cuando $P(i+1)/W(i+1) \leq P(i)/W(i)$:

```
#include <stdio.h>
#define  N        3
main(){
     float Cu,M=20, P[]={24,15,25},W[]={15,10,18},X[]={0,0,0},ganancia=0;
     float R[N];
```

```
    int i;
    Cu=M;
    for (i=0;i<N;i++){
       if(W[i]>Cu)
              break;
       X[i]=1;
       Cu=Cu-W[i];
    }
    if (i<N)
      X[i]=Cu/W[i];
    for (i=0;i<N;i++){
       printf("X[%d]=%f..",i,X[i]);
       ganancia=ganancia+X[i]*P[i];
    }
    printf("\nGanancia=%f\n",ganancia);
    getchar();
}
```

Mientras que las dos primeras estrategias no garantizan la solución óptima para el problema de la mochila, el siguiente teorema muestra que la tercera estrategia siempre obtiene una solución óptima.
Teorema: Si $P_1/W_1 \geq P_2/W_2 \geq P_3/W_3 \geq \ldots \geq P_n/W_n$ entonces el algoritmo de la mochila genera un óptimo a la instancia dada por el problema.

7.3 Patrón óptimo de concatenación (Optimal Merge Pattern)

Dos archivos ordenados se pueden unir en un solo archivo en $\mathcal{O}(n + m)$. Cuando se tienen que unir más de dos archivos, esto se puede realizar uniendo los archivos en pares. Así, si se van a unir X_1, X_2, X_3, X_4, uno puede unir X_1 y X_2 para producir Y_1. Y_1 y X_3 para producir Y_2, Y_2 con X_4 para producir el archivo deseado. Alternativamente se pueden primero unir X_1 y X_2 obteniendo Y_1, unir X_3 y X_4 obteniendo Y_2 y finalmente Y_1 con Y_2 para producir el archivo deseado. Existen varias formas para unir varios archivos en uno solo. Cada estrategia requiere diferente cantidad de tiempo. El problema es determinar un tiempo óptimo para unir tales archivos. Ejempo: X =(30. 20, 10) donde X es un vector donde tiene el tamaño de los archivos ordenados por magnitud. Uniendo X_1 con X_2 se requieren 50 movimientos. Uniendo el resultado con X_3 se requieren otros 60 movimientos. El número total de movimientos para unir los 3 archivos es de 110. Si unimos primero X_2 y X_3 toma 30 movimientos y posteriormente se une X_1 (60 movimientos más) el total de movimientos será de 90. Por lo que la segunda estrategia es más rápida que la primera.
Un intento greedy para obtener un óptimo es fácilmente formulado. Unir dos archivos de n y m registros requiere n + m movimientos de registros, el criterio obvio es: En cada paso unir los dos archivos de menor tamaño. Por ejemplo, si tenemos F=(20, 30, 10, 5, 30) el método greedy generará los siguientes pasos: Unir F_3 y F_4 para obtener Z_1, ($|Z_1|$=15); unir Z_1 con F_1 para obtener Z_2 ($|Z_2|$=35); unir F_2 con F_5 para producir Z_3 ($|Z_3|$=60); unir Z_2 y Z_3 para obtener Z_4 ($|Z_4|$=95); El total de movimientos es (95 + 60 + 35 + 15 = 205, siendo éste el óptimo.
La unión se puede representar como un árbol binario. Los nodos hoja son los archivos a unir.

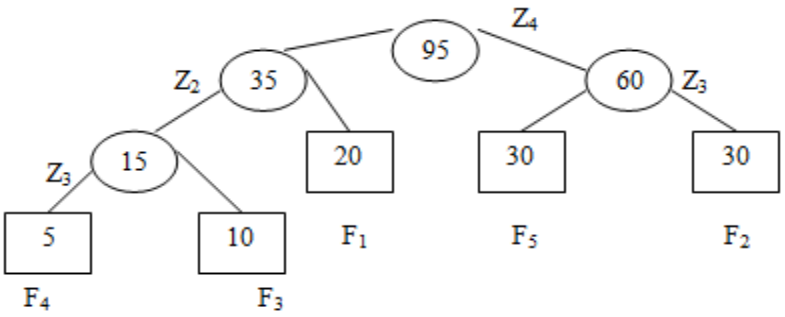


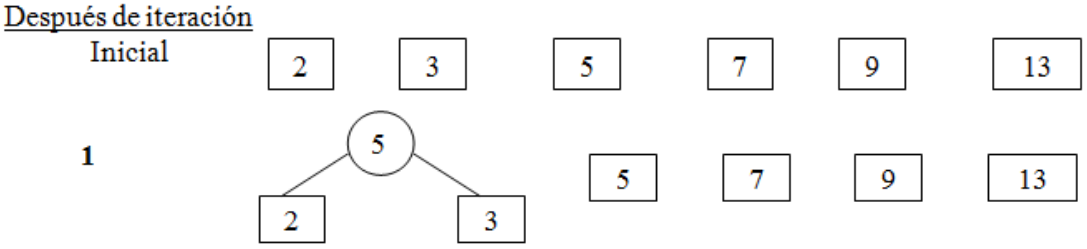
Fig. 7.1 Patrón óptimo de concatenación.

Si d_i es la distancia entre la raíz y el nodo extremo del archivo F_i y q_i, el tamaño de F_i, entonces el número total de movimientos de registros se define como $\sum_{i=1 \text{ hasta } n} d_i\, q_i$, esta suma se conoce como el peso específico de la trayectoria para la parte externa del árbol binario.
El algoritmo ÁRBOL contiene una lista de entrada L con n árboles. Cada árbol tiene tres campos, LCHILD, RCHILD y peso. Al inicio, cada árbol sólo tiene un nodo y todos los nodos son externos. El peso representa el número de registros del archivo. WEIGHT(T) es el tamaño de los archivos a unirse. El procedimiento TREE emplea tres subalgoritmos:
GETNODE(T) provee el nuevo nodo a usarse en el árbol a construirse. LEAST(L) encuentra un árbol en L cuya raíz tiene el menor peso. Este árbol es removido de L. INSERT(L, T) inserta el árbol con raíz T en la lista L.

El algoritmo se muestra en pseudocódigo:

```
PROCEDURE TREE (L, n)
for i ← 1 to n- 1 do
        Call GETNODE (T)
        LCHILD(T) ← LEAST(L)
        RCHILD(T) ← LEAST (L)
        WEIGHT (T) ← WEIGHT (LCHILD(T)) + WEIGHT(RCHILD(T))
        CALL INSERT(L, T)
repeat
return(LEAST(L))
end TREE
```

Ejemplo:
Sea L=(2, 3, 5, 7, 9, 13) el tamaño de seis archivos. En la figura 3.2 se mostrará el final de cada iteración del "for" (un árbol de unión binaria.)



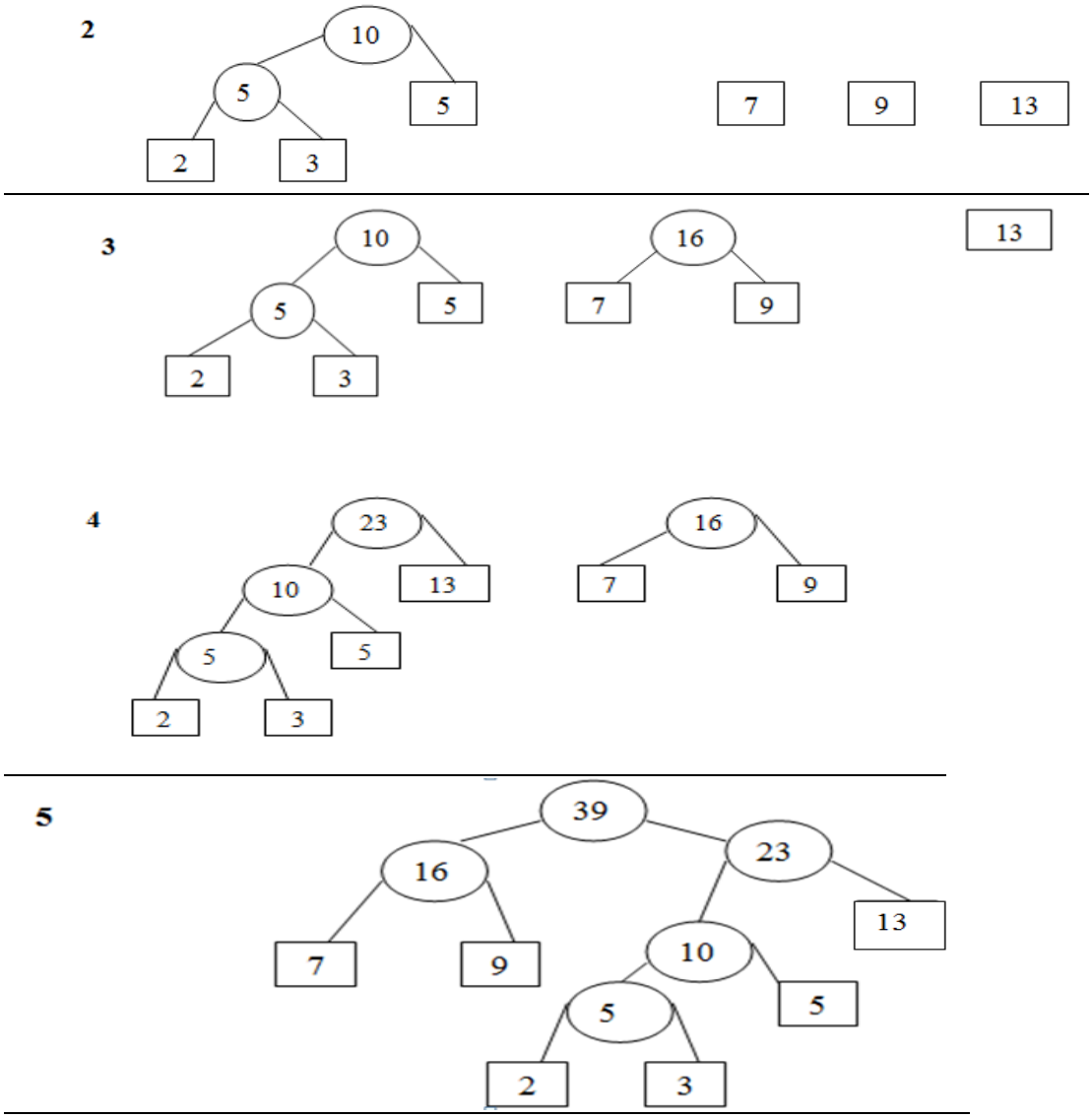


Fig. 7.2 Un ejemplo de árbol de unión binaria.

7.4 Árbol de expansión mínima (Minimun Spanning Tree)

Definición: Sea G(V, E) una gráfica no direccionada. Una subgráfica T(V, E`) es un árbol de extensión de G iff T es un árbol.

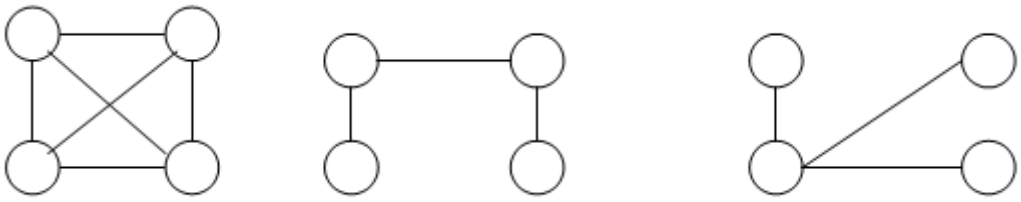


Fig. 7.3 Una subgráfica.

Si los nodos representan ciudades y el segmento que las une representa una posible unión entre ambas ciudades, entonces el mínimo número de segmentos para unir las n ciudades es n-1. El árbol de extensión representa todas las posibles combinaciones.
En una situación práctica, los segmentos tendrán pesos asignados. Estos pesos pueden representar costos de construcción, distancias, etc. Ahora, dado un peso, lo que se desea es conectarse a todos los nodos con el mínimo costo. En cualquier caso, los segmentos seleccionados formarán un árbol (suponiendo todos los costos positivos). El interés será localizar un árbol de extensión en G con el costo mínimo.
Un método Greedy para obtener un mínimo costo será ir edificando el árbol segmento por segmento. El siguiente segmento a escoger será aquel en que se minimice el incremento en costos.
Si A es el conjunto de segmentos seleccionados hasta el momento, entonces A forma un árbol. El siguiente segmento (u, v) a ser incluido en A es un segmento con costo mínimo no en A con la propiedad de que A U {u, v} también es un árbol. Este algoritmo se conoce como el algoritmo de Prim.

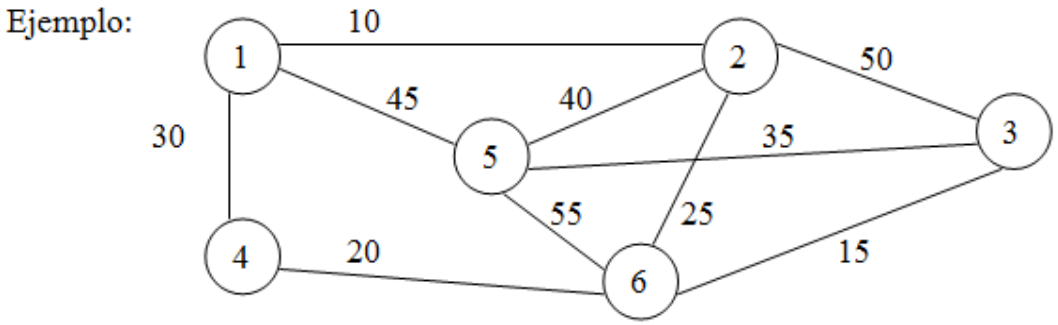


Fig. 7.4 Un ejemplo de grafo no dirigido.
En la siguiente figura se muestra la forma en que trabaja el método PRIM. El árbol de expansión tiene un costo de 105.

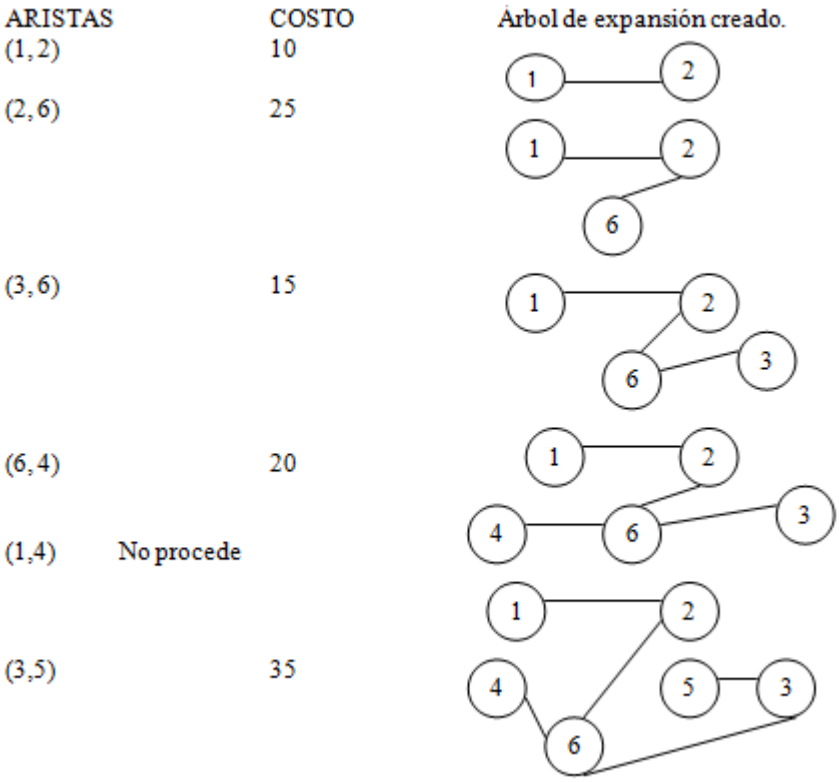


Fig. 7.5 Solución paso a paso del árbol de expansión mínima.

El algoritmo inicia con un árbol que incluye sólo un borde con costo mínimo de G. Entonces, las aristas serán adicionadas al árbol una por una. La siguiente arista (i, j) a ser adicionada es tal que el vértice i se encuentra incluido en el árbol y j es el vértice aún no incluido y el COSTO(i, j) es mínimo sobre todas la aristas (k, l) donde el vértice k es parte del árbol y el vértice l no se encuentra en el árbol. En orden de determinar este vértice (i, j) en forma eficiente, nosotros asociamos por cada vértice j aún no incluido en el árbol un valor NEAR(j). NEAR(j) es un vértice en el árbol tal que COSTO(j, NEAR(j)) es mínimo sobre todas las elecciones para NEAR(j). Se define NEAR(j)=0 para todos los vértices j que se encuentran en el árbol. La siguiente arista a incluir es definida por el vértice j tal que NEAR(j) ≠0 (j no se encuentra aún en el árbol) y COSTO(j, NEAR(j)) es mínimo.

```
#include <stdio.h>
#define TAM      6
main(){
      //costo(n,n) es la matriz de costos del trayecto del grafo
      //El costo(i,i) es infinito, el costo(i,j), donde i!=j, es positivo.
      //El trayecto se guarda en el arrego T(n,2)
      //El costo final se asigna a minicost.
      //Se requiere un vector que indique  el nodo  más cercano al nodo j, ese vector
       //NEAR se guarda en la variable ne.
      int
cost[TAM][TAM]={{999,10,999,30,45,999},{10,999,50,999,40,25},{999,50,999,999,35,15}
,{30,999,999,999,999,20},{45,40,35,999,999,55},{999,25,15,20,55,999}};
      int ne[6], n, i, j, k, l, t[6][2], min=999,mincost;
      for (i=0;i<TAM;i++)
            for (j=0;j<TAM;j++)
```

```
            if (min>cost[i][j]){
                    min=cost[i][j];
                    k=i;
                    l=j;
          }
mincost=cost[k][l];
t[0][0]=k;
t[0][1]=l;
for (i=0;i<TAM;i++)
          if (cost[i][l]<cost[i][k])
            ne[i]=l;
   else
             ne[i]=k;
ne[k]=ne[l]=-1;
for (i=1;i<TAM-1;i++){
          min=999;
          for (k=0;k<TAM;k++){
             if (ne[k]!=-1 && cost[k][ne[k]]<min){
                    min=cost[k][ne[k]];
                    j=k;
             }
    }
    t[i][0]=j;
          t[i][1]=ne[j];
          mincost=mincost+cost[j][ne[j]];
          ne[j]=-1;
          for (k=0;k<TAM;k++)
                  if(ne[k]!=-1 && cost[k][ne[k]]>cost[k][j])
                ne[k]=j;
}
printf("Costo del árbol de expansión mínima=%d\n",mincost);
for (i=0;i<TAM-1;i++)
             printf("Trayecto de %d a %d\n",t[i][0]+1,t[i][1]+1);
getchar();
getchar();
}
```

El tiempo para ejecutar el algoritmo PRIM es del $\Theta(n^2)$ donde n es el número de nodos.
La matriz COSTO del ejemplo anterior junto con una simulación de los valores históricos del vector NEAR queda de la siguiente forma:

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

Tabla 7.1

7.5 Single Source Shortest Paths (La ruta más corta a partir de un origen)

Los grafos pueden ser utilizados para representar carreteras, estructuras de un estado o país con vértices representando ciudades y segmentos que unen los vértices como la carretera. Los segmentos pueden tener asignados pesos que marcan una distancia entre dos ciudades conectadas.

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

Ejemplo:

Considere la gráfica dirigida de la siguiente figura. El número de trayectos también es el número de pesos. Si v_0 es el vértice de origen, entonces el trayecto más corto desde v_0 a v_1 es v_0 v_2 v_3 v_1. La distancia del trayecto es 10 + 15 + 20 = 45. En este caso, recorrer tres caminos es más económico que recorrer en forma directa v_0 v_1, el cual tiene un costo de 50. No existe trayecto alguno de v_0 a v_1.

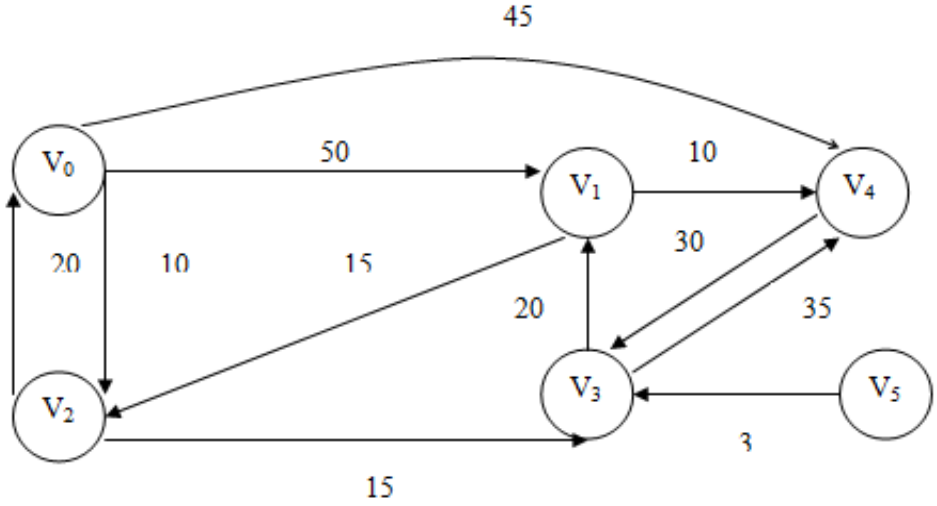


Fig. 7.6 Un ejemplo para el algoritmo del trayecto más corto

Para formular un algoritmo greedy para generar el trayecto más corto, debemos concebir una solución multi etapa. Una posibilidad es construir el trayecto más corto uno por uno. Como una medida de optimización se puede usar la suma de todos los trayectos hasta el momento generados. En orden de que la medida sea mínima, cada trayecto individual debe ser de tamaño mínimo. Si se han construido i trayectos mínimos, entonces el siguiente trayecto a ser construido debería ser el siguiente trayecto con mínima distancia. El camino greedy para generar los trayectos cortos desde V_0 a los vértices remanentes es generando los trayectos en orden creciente. Primero, el trayecto más corto al vértice más cercano es generado. Entonces el trayecto más corto al segundo vértice más cercano se genera y así sucesivamente. Para la gráfica del ejemplo, el trayecto más cercano para V_0 es V_2 (c(V_0, V_2)=10). Por lo que el trayecto V_0 V_2 será el primer trayecto generado. El segundo trayecto más cercano es V_0 V_3 con una distancia de 25. El trayecto V_0 V_2 V_3 será el siguiente trayecto generado. Para generar los siguientes trayectos se debe determinar i) el siguiente vértice que con el cual deba generar un camino más corto y ii) un camino más corto para éste vértice. Sea S el conjunto de vértices

(incluyendo V_0) en el cual el trayecto más corto ha sido generado. Para w no en S, sea DIST(w) la distancia del trayecto más corto desde V_0 yendo sólo a través de esto
s vértices que están en S y terminando en w. Se observa que:

I. Si el siguiente trayecto más corto es al vértice u, entonces el trayecto inicia en V_0, termina en u y va a través de los vértices localizados en S.
II. El destino del siguiente trayecto generado debe de ser aquel vértice u tal que la mínima distancia, DIST(u), sobre todos los vértices no en S.
III. Habiendo seleccionado un vértice u en II y generado el trayecto más corto de V_0 a u, el vértice u viene a ser miembro de S. En este punto la dimensión del trayecto más corto iniciando en V_0. irá en los vértices localizados en S y terminando en w no en S puede decrecer. Esto es, el valor de la distancia DIST(w) puede cambiar. Si cambia, entonces se debe a que existe un trayecto más corto iniciando en V_0 posteriormente va a u y entonces a w. Los vértices intermedios de V_0, a u y de w a w deben estar todos en S. Además, el trayecto V_0 a u debe ser el más corto, de otra forma DIST(w) no está definido en forma apropiada. También, el trayecto u a w puede no contener vértices intermedios.

Las observaciones arriba indicadas forman un algoritmo simple. (El algoritmo fue desarrollado por Dijkstra). De hecho solo determina la magnitud de la trayectoria del vértice V_0 a todos los vértices en G.
Se asume que los n vértices en G se numeran de 1 a n. El conjunto se mantiene S con un arreglo con S(i)=0 si el vértice i no se encuentra en S y S(i)=1 si pertenece a S. Se asume que la gráfica se representa por una matriz de costos.

En pseudocódigo el algoritmo es el siguiente:

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

El código en C con matrices declaradas dinámicamente es:

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

```
//La ruta más corta a partir de un origen
void generar(int **,int);
void path(int **,int *,int,int);

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

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

void path(int **cost,int *dist,int tam,int v){
```

```
int u,w,i,num,s[tam],min;
for(i=0;i<tam;i++){
s[i]=0;
dist[i]=cost[v][i];}
s[v]=1;
dist[v]=0;
for(num=1;num<tam-1;num++){
                min=9999;
                for(w=0;w<tam;w++)
                if(s[w]==0 && dist[w]<min){
                                min=dist[w];
                                u=w;
                                }
                                s[u]=1;
                                for(w=0;w<tam;w++)
                                if(s[w]==0){
                                        if(dist[w]<dist[u]+cost[u][w])
                                        dist[w]=dist[w];
                                        else
                                        dist[w]=dist[u]+cost[u][w];
                                        }
                }
}
```

El tiempo que tarda el algoritmo con n vértices es $\mathcal{O}(n^2)$. Esto se ve fácilmente ya que el algoritmo contiene dos for anidados.

Ejemplo.

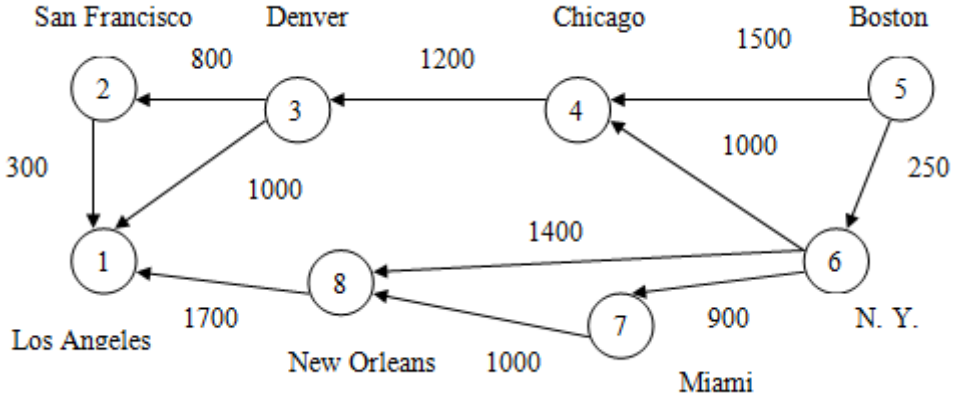


Fig. 7.7 Un ejemplo para el trayecto más corto.

La matriz de costos es:

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8								0

Tabla 7.2

Si v=5, nos indica que se busca el trayecto de mínimo costo a todos los nodos desde el nodo 5. Por lo tanto, la corrida es:

Iteración	S	Vértice	1	2	3	4	5	6	7	8
Inicial	-									
1	5	6	∞	∞	∞	1500	∞	250	∞	∞
2	5. 6	7	∞	∞	∞	1250	0	250	1150	1650
3	5, 6, 7	4	∞	∞	∞	1250	0	250	1150	1650
4	5, 6, 7, 4	8	∞	∞	2450	1250	0	250	1150	1650
5	5, 6, 7, 4, 8	3	3350	∞	2450	1250	0	250	1150	1650
6	5, 6, 7, 4, 8, 3	2	3350	3250	2450	1250	0	250	1150	1650
	5, 6, 7, 4, 8, 3, 2		3350	3250	2450	1250	0	250	1150	1650

Tabla 7.3

Se observará que este algoritmo tiene una complejidad de $\mathcal{O}(n^2)$ donde n es el número de nodos. Se tiene un for anidado de la siguiente forma:

$\mathcal{O}(n^2)$ [$\mathcal{O}(n)$] [$\mathcal{O}(n)$]

,

Ejercicios.

1. Dado el problema de la mochila para:
   n=4, M=50, P=(30, 15, 10, 40) W=(20, 30, 25, 19)
   Determinar los valores que optimicen la ganancia.
2. Dada la siguiente matriz de costos:

$$C=\begin{bmatrix} \infty & 10 & \infty & 40 & 30 & \infty \\ 10 & \infty & 50 & \infty & 40 & 45 \\ \infty & 50 & \infty & \infty & 40 & 10 \\ 30 & \infty & \infty & \infty & \infty & 40 \\ 50 & 40 & 50 & \infty & \infty & 10 \\ \infty & 30 & 20 & 40 & 50 & \infty \end{bmatrix}$$

   Obtener el árbol de expansión mínima.

3. Dada la siguiente matriz de costos:

	1	2	3	4	5	6	7	8
1	0							
2	400	0						
3	800	600	0					
4			1200	0				
5				1400	0	250		
6				1200		0	1000	1600
7							0	1000
8								0

Obtener el trayecto más corto de 5 a 1

VIII PROGRAMACIÓN DINÁMICA. (Dynamic Programming).

(Ellis horowitz, 1978)
(Dasgupta, Papadimitriou, & Vazirani, 2008)

Principio de optimalidad

Cuando hablamos de *optimizar* nos referimos a buscar alguna de las **mejores** soluciones de entre muchas alternativas posibles. Dicho proceso de optimización puede ser visto como una secuencia de decisiones que nos proporcionan la solución correcta. Si, dada una subsecuencia de decisiones, siempre se conoce cuál es la decisión que debe tomarse a continuación para obtener la secuencia óptima, el problema es elemental y se resuelve trivialmente tomando una decisión detrás de otra, lo que se conoce como estrategia voraz. En otros casos, aunque no sea posible aplicar la estrategia voraz, se cumple el **principio de optimalidad de Bellman** enunciado en 1957 y que dicta que «dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima». En este caso sigue siendo posible el ir tomando decisiones elementales, en la confianza de que la combinación de ellas seguirá siendo óptima, pero será entonces necesario explorar muchas secuencias de decisiones para dar con la correcta, siendo aquí donde interviene la programación dinámica. Aunque este principio parece evidente no siempre es aplicable y por tanto es necesario verificar que se cumple para el problema en cuestión.

Contemplar un problema como una secuencia de decisiones equivale a dividirlo en problemas más pequeños y por lo tanto más fáciles de resolver como hacemos en Divide y Vencerás, técnica similar a la de programación dinámica. La programación dinámica se aplica cuando la subdivisión de un problema conduce a:

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

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

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

En general, el diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:

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

8.1 Gráficas de Múltiples Etapas. (MultiStage Graphs)

Es una gráfica dirigida en el cual los vértices son particionados en K ≥2 conjuntos disjuntos V_i, $1 \leq i \leq k$. En adición, si <u, v> son una arista en E entonces u Є V_i y vЄV_{i+1} para algun i, $1 \leq i < k$. El conjuto V_1 y V_k son tales que $|V_1| = |V_k| = 1$. Sea s y t respectivamente el vértice en V_1 y V_k. s es la fuente y t es la meta a llegar. Sea c(i, j) el costo de la arista <i, j>. El costo del trayecto de s a t iniciando en el estado 1, va la etapa 2, posteriormente al la etapa 3, a la etapa 4 etc. Y eventualmente termina en la etapa k. En la siguiente figura muestra una gráfica de 5 etapas. El trayecto de mínimo costo de s a t se muestra en negrita.

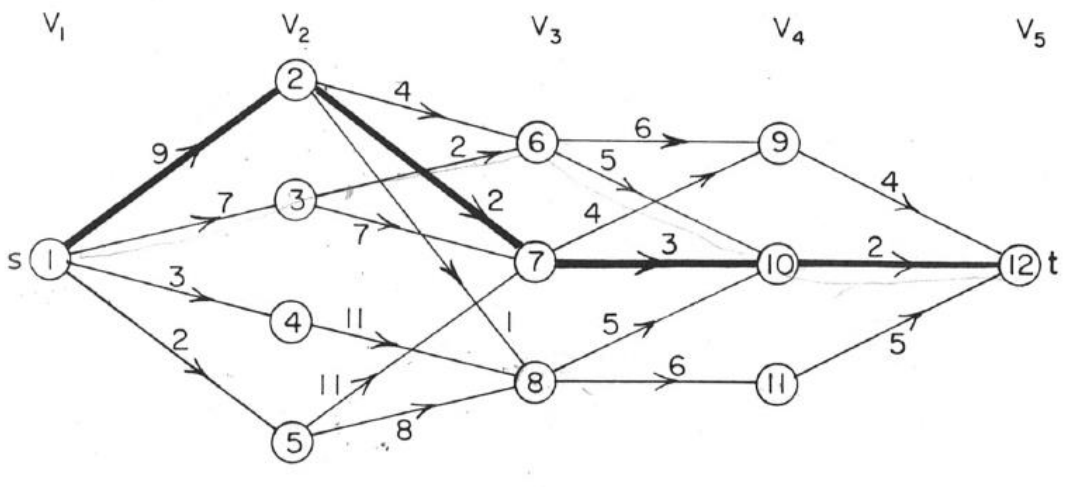


Fig. 8.1 Grafica de 5 etapas.

Varios problemas pueden ser formulados como problemas de múltiple etapa. Se dará sólo un ejemplo. Considere un problema de asignación de recursos en el cual n unidades de recursos van a ser asignados a r proyectos. Si j, 0<=j<=n unidades de recursos son asignados al proyecto i entonces el beneficio neto resultante es N(i, j). El problema es asignar el recurso al proyecto r de tal forma que maximice el beneficio neto. Este problema puede ser formulado como una grafica de r+1 etapas como sigue. Etapa i, 1<=i<=r representa el proyecto i. Hay n + 1 vertices V(i, j), 0<=j<=n asociados con la etapa i, 2<=i<=r. Etapas 1 y r + 1 cada uno tiene un vértice V(1,0)=s y V(r+1, n) = t respectivamente. Vértice V(i, j), 2<=i<=r representa el estado en el cual un total de j unidades de recursos han sido asignados a los proyectos 1,2, . . .i-1. Las aristas en G son de la forma <V(i, j, V(i+1, L)>, para toda j<=L y 1<=r. La arista <(v(i,j, Vi+1, L)>, j<=L se asigna con un peso o costo de N(i, L-j) y corresponde a la asignación de L-j unidades de recursos al proyecto i, 1<=i<r. En adición, G tiene aristas del tipo <V(r,j, V(r+1,n)>. A cada de estas aristas es asignado un peso de $\max_{0<=p<=n-j}\{N(r, p)\}$. La grafica resultante con un proyecto de tres problemas con n=4 se muestra en la siguiente figura. Debería ser fácil ver que una asignación óptima de recursos se define por un máximo costo en la trayectoria a t. Esto es fácilmente convertido a un problema de consto mínimo sólo cambiando el singo de toda arista.

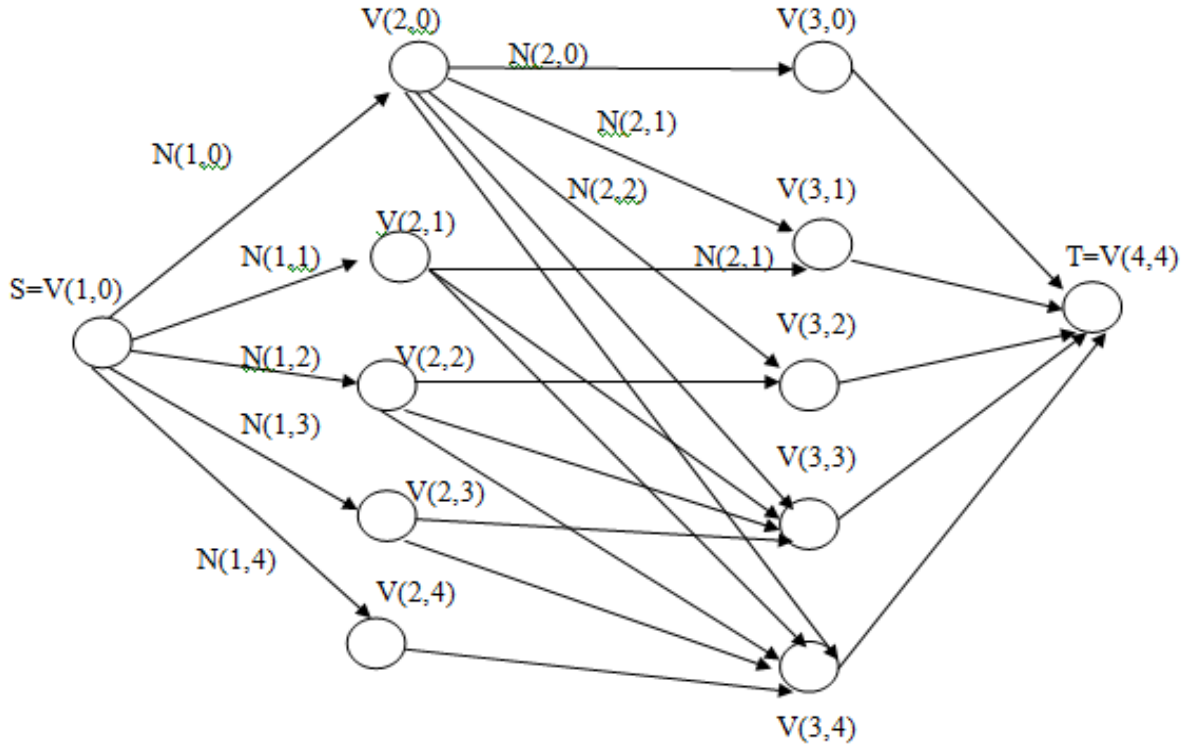


Fig. 8.2 Gráfica de etapas correspondiente a un proyecto de 3 problemas.

Una formulación de programación dinámica para un problema gráfico de k etapas se obtiene primero, al darse cuenta que todos los caminos de s a t es el resultado de una secuencia de k – 2 decisiones. La i-ésima decisión consiste en determinar cuál vértice en V_{i+1}, 1<=i<=k – 2, va a estar en el trayecto. Es fácil de observar que el principio de optimización se cumple. Sea P(i,j) el costo de esta trayecto. Entonces usando el enfoque hacia adelante, se obtiene:

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

Siendo E el conjunto de aristas del grafo. El costo COSTO(k – 1, j)=c(j, t) si (j, t) € E y COSTO(k – 1, j) = ∞ si (j, t) ¢ E. Resolviendo la gráfica de 5 etapas indicada en la figura 8.1, se obtienen los siguientes valores:

COSTO(3,6) = min{6 + COSTO(4,9), 5 + COSTO(4,10)}= min{6+4, 5+2} = 7
COSTO (3,7)= min{4 + COSTO(4,9), 3 + COSTO(4,10)}= min{4+4, 3+2} = 5
COSTO(3,8)= min{5 + COSTO(4,10),6 + COSTO(4,11)}= min{5+2, 6+5} = 7

COSTO(2,2)= min{4 + COSTO(3,6), 2+COSTO(3,7), 1+ COSTO(3,8)}= 7
COSTO(2,3)=9
COSTO(2,4)=18
COSTO(2,5)=15

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

Por lo que el mínimo el trayecto del mínimo costo de s a t es 16. Este trayecto puede ser determinado fácilmente si registramos la decisión realizada en cada etapa. Sea D(i,j) el valor de L que minimiza c(j,L) + COSTO(i + 1,L), para la gráfica de 5 etapas se tiene:
D(3,6)=10; D(3,7)=10; D(3,8)=10;
D(2,2)=7; D(2,3)=6; D(2,4)=8; D(2,5)=8;
D(1,1)=2

Por lo que el trayecto del mínimo costo puede ser s=1, v_2, v_3, v_4,. . . v_{k-1}, t. Es fácil ver que v_2= D(1,1)=2; v_3= D(2, D(1,1))=7 y v_4=D3, D(2,D(1,1)))=D(3,7)=10.
Antes de escribir un algoritmo para resolver una gráfica de k etapas, se impondrá un orden en los vértices en V. Éste orden será fácil de escribir el algoritmo. Se requerirá que los n vértices en V estén indexados de 1 hasta n. Los índices serán asignados en orden a los a las etapas. Primero s será asignado al índice 1, los vértices en V_2 son asignados en al índice, así sucesivamente. t tiene el índice n. Por lo que los índices en V_{i+1} son mayores que los asignados a los vértices en V_i. Como un resultado de este esquema de índices, COSTO y D pueden ser calculados en el orden n-1, n-2,. . ., 1. El primer subíndice en COSTO, P y D sólo identifica el número de etapa y es omitido en el algoritmo. El algoritmo es:

```
Procedure FGRAPH(E, k, n, P)
        //k Número de etapas en la gráfica.
        //E un conjunto de aristas
        //c(i,j) es el costo de <i, j>.
        //P(1:K) es la trayectoria de mínimo costo.
Real COSTO(n), integer (D(n-1), P(k), r, j, k, n
COSTO(n)=0;
For j=n-1 a 1 by -1 do          //calcular COSTO(j)
        Sea r un vértice tal que <j, r>  € E y c(j,r) + COSTO( r ) sea mínimo.
        COSTO(j)=c(j, r) + COSTO( r)
        D(j)=r
repeat
//busca el trayecto del mínimo costo.
P(1)=1; P(k)=n
For  j= 2 to k - 1 do
        P(j ) = D(P(j-1));
 repeat
end FGRAPH
```

Se observará que existen dos operadores de control for que no son anidados, por lo que el tiempo necesario será Θ(n).

8.2 El Problema del Agente Viajero (The Traveling Sales Person Problem, TSP)

Éste es un problema de permutaciones. Un problema de permutaciones usualmente será mucho más difícil de resolver que problemas en el que se escogen subconjuntos ya que existen n! permutaciones de n objetos mientras que sólo existen 2^n diferentes subconjuntos de n objetos (n! > O(2^n)).

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

Tabla 8.1

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

Una aplicación del problema es la siguiente: Suponga que se tiene una ruta de una camioneta postal que recoge correo de cajas de correo localizados en n diferentes sitios. Una gráfica de n+1 vértices puede ser usada para representar tal situación. Un vértice representa la oficina postal desde donde la camioneta postal inicia su recorrido y en el cual debe de retornar. La arista <i, j> tiene asignado un costo igual a la distancia desde el sitio i al sitio j. La ruta tomada por la camioneta postal es una gira y lo que se espera es minimizar el trayecto de la camioneta.

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

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

Generalizando, se obtiene (para i ¢ S):

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

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

El algoritmo programado en C es:

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

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

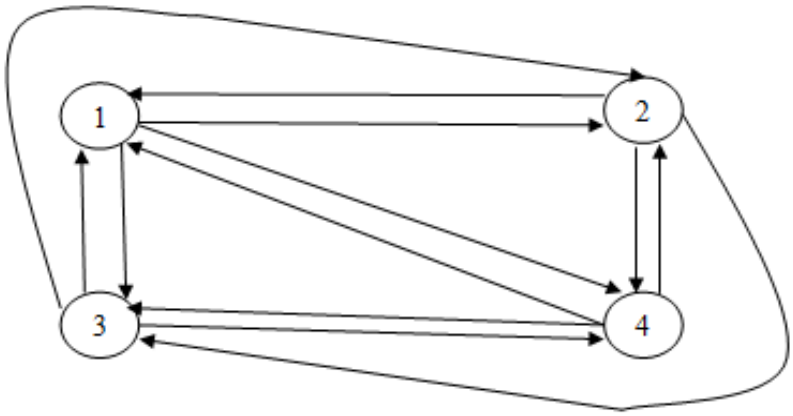
```
main(){
      int **cost,*m,d,o,v,i,j,r;
      printf("       ***************    TSP    ***************\n");
      printf("\nIntroduce el numero de vertices: ");
      scanf("%d",&v);
      cost=(int **)malloc(sizeof(int *)*v);
      for(i=0;i<v;i++)
                  cost[i]=(int *)malloc(sizeof(int)*v);
      m=(int *)malloc(sizeof(int)*v);
      for(i=0;i<v;i++)
          m[i]=0;
      for(i=0;i<v;i++)
         for(j=0;j<v;j++){
                  if(i==j)
                           cost[i][j]=0;
                  else
                           cost[i][j]=9999;
         }
      generar(cost,v);
      printf("\nIntroduce el origen:");
      scanf("%d",&r);
      printf("La matriz generada es:  \n");
      for(i=0;i<v;i++){
      for(j=0;j<v;j++){
      printf("%6d",cost[i][j]); } printf("\n"); }
      printf("\n\nCosto minimo a partir de %d: %d\n",r,tsp(cost,m,v,r-1,v,r-1));
      system("PAUSE");
      }

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

int tsp(int **cost,int *m,int d,int o,int v,int r){
    if(d==1)
```

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

Ejemplo. Considere la siguiente gráfica donde el tamaño de las aristas se dan en la matriz c:



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

Fig. 8.3 Gráfica dirigida cuya longitud de cada arista se localiza en la matriz C.

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

Utilizando a (2) se obtiene:

$g(2, \{3\}) = c_{2,3} + g(3,\phi) = 15$; $g(2, \{4\}) = c_{243} + g(4,\phi) = 18$;
$g(3, \{2\}) = 18$; $g(3, \{4\}) = 20$;
$g(4, \{2\}) = 13$; $g(4, \{3\}) = 15$;

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

$g(2, \{3,4\})=\min\{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = 25$
$g(3, \{2,4\})=\min\{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\} = 25$
$g(4, \{2,3\})=\min\{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\} = 23$

Finalmente, de (1) obtenemos:

$g(1, \{2,3,4\}) = \min\{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2,4\}), c_{14} + g(4, \{2,3\})\}$
$= \min\{35, 40, 43\}$
$= 35.$

El árbol recursivo se observa en la figura 8.4.

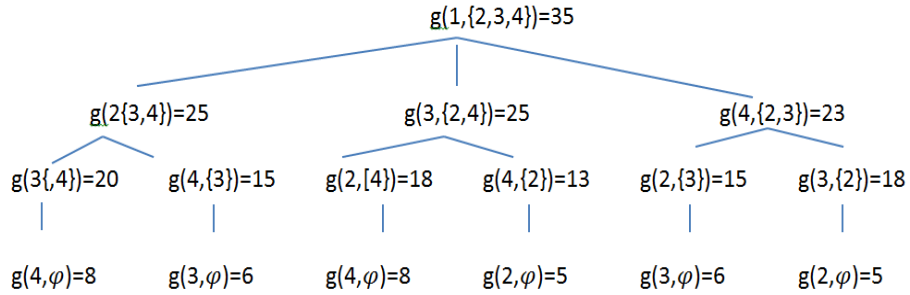


Figura 8.4 Árbol recursivo del Agente Viajero

Una gira óptima de la gráfica de la figura tiene una longitud de 35. Una gira de esta longitud puede ser construida si se retiene de cada g(i, S) el valor de j que minimiza el lado derecho de (2). Sea J(i, S) este valor. Entonces J(1,{2, 3, 4}) =2. De esta forma la gira inicia de 1 a 2. El siguiente punto a visitar se obtiene de g(2,{3,4}), J(2,{3, 4}=4, por lo que la siguiente arista es <2. 4>. Lo que falta de la gira es g(4,{3}), J(4,{3})=3. El recorrido óptimo es 1, 2, 4, 3, 1.
Sea N el número de g(i, S) que tiene que ser calculado antes que de que g(1, V – {1}) sea calculado. Para cada valor de |S| hay n – 1 opciones para i. El número de conjuntos distintos S de tamaño k que no incluyen a 1 y a ***i*** es $\left| \begin{matrix} n-2 \\ k \end{matrix} \right|$

De esta forma:

$$N = \sum_{K=0}^{n-2} (n-1) \left| \begin{matrix} n-2 \\ k \end{matrix} \right| = (n-1)2^{n-2}$$

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

Ejercicios.

1. Dada la siguiente matriz de costos:

$$C = \begin{matrix} 10 & 20 & 10 & 5 & \\ 5 & 0 & 10 & 15 & 6 \\ 7 & 6 & 0 & 10 & 7 \\ 8 & 6 & 5 & 0 & 6 \\ 3 & 4 & 2 & 3 & 0 \end{matrix}$$

Determinar el trayecto del agente viajero iniciando en el punto uno.

IX Retorno Sobre la Misma Ruta. (Backtracking)

(Ellis horowitz, 1978)
(Dasgupta, Papadimitriou, & Vazirani, 2008)

En la búsqueda de los principios fundamentales del diseño de algoritmos, backtracking representa una de las técnicas más generales. Varios problemas que tratan de la búsqueda de un conjunto de soluciones o buscan una solución óptima satisfaciendo algunas restricciones pueden ser resueltos usando backtracking.
Para utilizar el método backtracking, el problema debe de estar expresado en n-tuplas. (x_1, x_2, x_3, . . . , x_n) donde x_i se escoge desde un conjunto finito S_i. A veces el problema a resolver es maximizar o minimizar una función $P(x_1, x_2, x_3, \ldots x_n)$. A veces se busca todos los vectores tales que satisfacen a P. Por ejemplo, ordenar los enteros localizados en A(1:n) es un problema cuya solución se expresa por n tuplas donde x_i es el índice de A donde se localiza el i-ésimo elemento más pequeño. La función P es la desigualdad $A(x_i) <= A(x_{i+1})$ para $1<= i <n$. Ordenación de números no es en sí un ejemplo de backtracking, sólo es un ejemplo de un problema cuya solución se puede formular por medio de n tuplas. En esta sección se estudiarán algunos algoritmos cuya solución se realiza mejor por medio de backtracking.
Suponga que m_i es el tamaño del conjunto S_i. Entonces existen $m = m_1, m_2 \ldots m_n$ tuplas cuyos posibles candidatos pueden satisfacer la función P. el enfoque de la "Fuerza bruta" formaría todas las n-tuplas y evaluaría a cada una con P. Salvando a los que producen el óptimo. La virtud de backtracking es la habilidad para producir la misma respuesta con un menor número de pasos. Su idea básica es construir un vector y evaluarlo con la función $P(x_1, x_2, x_3, \ldots x_n)$ para probar si el vector recién formado tiene una oportunidad de ser el óptimo. La gran ventaja de éste método es: si se observa que el vector parcial (x_1, x_2, x_3, . . . , x_n) no tiene la posibilidad de conducir hacia una posible solución óptima, entonces $m_{1+1}, \ldots m_n$ puede ser ignorado completamente.
Varios de los problemas que se resuelven utilizando backtracking requieren que todas las soluciones satisfagan a un complejo conjunto de restricciones. Estas restricciones pueden ser divididas en dos categorías: explicitas e implícitas. Restricciones explicitas son reglas cuyas restricciones de cada x_i toman valores para un conjunto determinado. Un ejemplo de restricciones explicitas es:

$x_i >= 0$	o	S_i = {todos los números reales son no negativos}
$x_i = 0$ o 1	o	$S_i = \{0, 1\}$
$l_i <= x_i <= u_i$	o	$S_i = \{a: l_i <= a <= u_i\}$

Las restricciones explicitas pueden o no depender de una particular instancia I del problema a ser resuelto. Todas las tuplas que satisfacen a restricciones explicitas definen a un posible espacio de solución para I. Las restricciones implícitas determinan cuál de las tuplas en un espacio de solución de I en realidad satisfacen la función del criterio. Así, restricciones implícitas describen el camino en el cual las x_i deben de relacionarse una a otra.

9.1 El problema de las ocho reinas (8-Queens).

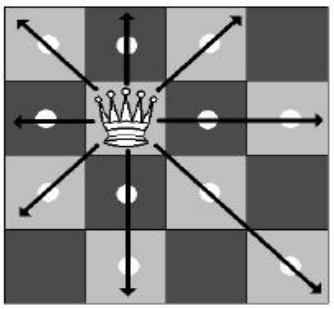


Fig. 9.1 Posiciones que puede atacar la reina.

Un problema combinatorio clásico es colocar las ocho reinas en un tablero de ajedrez de tal forma que no se puedan atacar entre ellas, esto es que no existan dos reinas en la misma hilera, columna o diagonal. Enumerando las hileras y columnas del tablero del 1 al 8. Las reinas pueden enumerarse también del 1 al 8. Ya que cada reina debe estar sobre una hilera diferente, se puede asumir que la reina *i* se colocará en la hilera *i*. Toda solución puede ser representad como 8 tuplas (x_1, x_2, x_3, . . . , x_8) donde x_i es la columna donde la reina *i* será colocada La restricción explicita usando esta formulación será S= {1, 2, 3, . . .,8}, $1 \leq i \leq n$. Por lo que espacio de soluciones consiste de 8^8 tuplas de 8. Una de las restricciones implícitas del problema es que dos x´s no deben de ser las mismas (esto es, toda reina debe de estar en diferente columna) y tampoco en la misma diagonal. La primer restricción indica que todas las soluciones son permutaciones de las 8 tuplas (1, 2, . . .,8). Esto reduce el espacio de la solución de 8^8 tuplas a 8! Tuplas.

Planteamiento del problema.
Como cada reina puede amenazar a todas las reinas que estén en la misma fila, cada una ha de situarse en una fila diferente. Podemos representar las 8 reinas mediante un vector[1-8], teniendo en cuenta que cada índice del vector representa una fila y el valor una columna. Así cada reina estaría en la posición (i, v[i]) para i = 1-8.

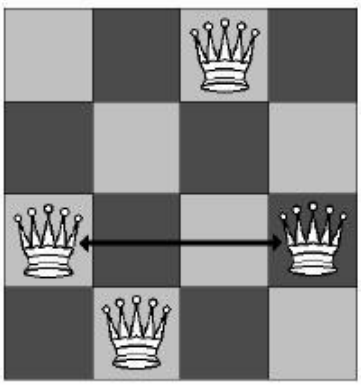


Fig. 9.2 Ejemplo de dos reinas amenazadas en el tablero de 4 por 4.

El vector (3,1,6,2,8,6,4,7) significa que la reina 1 está en la columna 3, fila1; la reina 2 en la columna 1, fila 2; la reina 3 en la columna 6, fila 3; la reina 4 en la columna 2, fila 4; etc. Como se puede apreciar esta solución es incorrecta ya que estarían la reina 3 y la 6 en la misma columna. Por tanto el vector correspondería a una permutación de los ocho primeros números enteros.

El problema de las filas y columnas lo tenemos cubierto, pero ¿qué ocurre con las diagonales? Para las posiciones sobre una misma diagonal descendente se cumple que tienen el mismo valor *fila − columna*, mientras que para las posiciones en la misma diagonal ascendente se

cumple que tienen el mismo valor *fila + columna*. Así, si tenemos dos reinas colocadas en posiciones (*i*,*j*) y (*k*,*l*) entonces están en la misma diagonal si y solo si cumple:

$i - j = k - l$ o $i + j = k + l$

$j - l = i - k$ o $j - l = k - i$

Teniendo todas las consideraciones en cuenta podemos aplicar el esquema backtracking para implementar las ocho reinas de una manera realmente eficiente. Para ello, reformulamos el problema como un problema de búsqueda en un árbol. Decimos que en un vector $V_{1...k}$ de enteros entre 1 y 8 es *k*-prometedor, para $0 \leq k \leq 8$ si ninguna de las *k* reinas colocadas en las posiciones (1, V_1), (2, V_2), . . ., (3, V_3) amenaza a ninguna de las otras. Las soluciones a nuestro problema se corresponden con aquellos vectores que son 8-prometedores.

i ϵ A

Establecimiento del Algoritmo:

Sea *N* el conjunto de vectores de *k*-prometedores, $0 \leq k \leq 8$, sea G = (N,A) el grafo dirigido tal que (U, V) ϵ A si y solo si existe un entero *k*, con $0 \leq k \leq 8$ tal que

- *U* es *k*-prometedor
- *V* es (*k* + 1)-prometedor
- $U_i = V_i$ para todo i ϵ{1, . . ., k}

Este grafo es un árbol. Su raíz es el vector vacío correspondiente a *k* = 0. sus hojas son o bien soluciones (*k* = 8), o posiciones sin salida (*k* < 8). Las soluciones del problema de las ocho reinas se pueden obtener explorando este árbol. Sin embargo no generamos explícitamente el árbol para explorarlo después. Los nodos se van generando y abandonando en el transcurso de la exploración mediante un recorrido en profundidad.

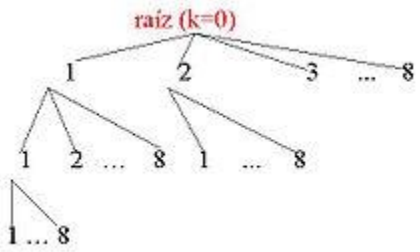


Fig. 9.3 Esquema reducido del árbol de soluciones.

Hay que decidir si un vector es *k*-prometedor, sabiendo que es una extensión de un vector (*k* − 1)-prometedor, únicamente necesitamos comprobar la última reina que haya que añadir. Este se puede acelerar si asociamos a cada nodo prometedor el conjunto de columnas, el de diagonales positivas (a 45 grados) y el de diagonales negativas (a 135 grados) controlados por las reinas que ya están puestas.

Descripción del Algoritmo:
A continuación se muestra el algoritmo que arroja la solución de nuestro problema, en el cual $S_{1...8}$ es un vector global. Para imprimir todas las soluciones, la llamada inicial es *reinas (0,0,0,0).*

Procedimiento reinas(k, col, diag45, diag135)
//$sol_{1\ldots k}$ es el k prometedor
//col = $\{sol_i \mid 1 \leq i \leq k\}$
//diag45 =$\{sol_i - i + 1 \mid 1 \leq i \leq k\}$
// diag135 =$\{sol_i + i - 1 \mid 1 \leq i \leq k\}$
si k = 8 entonces //un vector 8-prometedor es una solución
Escribir sol
Si no //explorar las extensiones (k+1) prometedoras de sol
Para j←1 hasta 8 hacer
Si j¢ col y j – k ¢ diag45 y j + k ¢ diag135 entonces
$Sol_{k+1} \leftarrow j//sol_{1,\ldots,k+1}$ es (k + 1) prometedor
Reinas(k+1, col U {j}, diag45 U {j – k}, diag135 U {j + k})

El algoritmo comprueba primero si *k* = 8, si esto es cierto resulta que tenemos ante nosotros un vector 8-prometedor, lo cual indica que cumple todas las restricciones originando una solución. Si *k* es distinto de 8, el algoritmo explora las extensiones (*k* + 1)-prometedoras, para ello realiza un bucle, el cual va de 1 a 8, debido al número de reinas. En este bucle se comprueba si entran en jaque las reinas colocadas en el tablero, si no entran en jaque, se realiza una recurrencia en la cual incrementamos *k* (buscamos (*k* + 1)-prometedor) y añadimos la nueva fila, columna y diagonales al conjunto de restricciones. Al realizar la recurrencia hemos añadido al vector sol una nueva reina la cual no entra en jaque con ninguna de las anteriores, además hemos incrementado el conjunto de restricciones añadiendo una nueva fila, columna y diagonales (una positiva y otra negativa) prohibidas.

Un ejemplo del árbol en profundidad puede verse fácilmente en la figura 9.4 con un ejemplo de las 4 reinas:

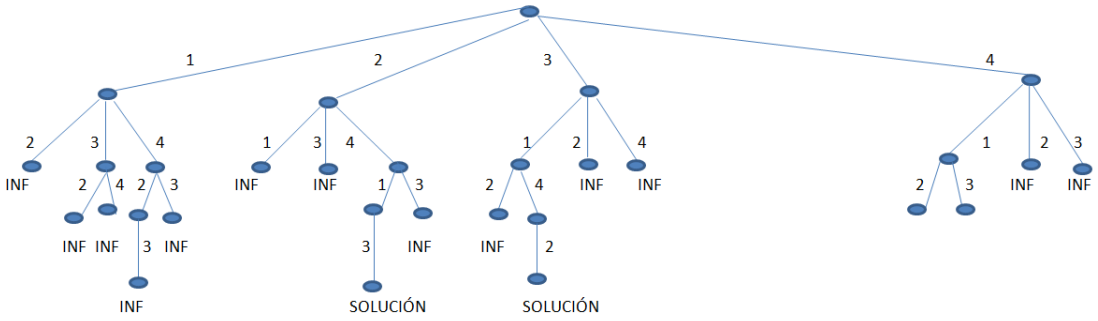


Fig. 9.4 Un árbol de decisión para 4 reinas

Se muestra enseguida el programa de las N reinas:

```
#include<stdio.h>
#include<stdlib.h>
void marcar(int **,int,int,int);
void vaciar(int **,int);
void solucion(int **,int);
void dam(int **,int **,int,int,int,int);
void regresar(int **,int **,int,int,int);
int cont=0;
```

```
main(){
      system("color 2f");
      int **matriz,**tablero,reinas,fila=0,columna=0;
      printf("Introduce el numero de reinas: ");
      scanf("%d",&reinas);
      matriz=(int **)malloc(sizeof(int *)*reinas);
      tablero=(int **)malloc(sizeof(int *)*reinas);
      for(int i=0;i<reinas;i++){
      matriz[i]=(int *)malloc(sizeof(int)*reinas);
      tablero[i]=(int *)malloc(sizeof(int)*reinas);}
      vaciar(matriz,reinas);
      vaciar(tablero,reinas);
      for(int i=0;i<reinas;i++)
      dam(matriz,tablero,i,0,1,reinas);
      if(cont==0)
      printf("No hay soluciones para el problema con %d reinas\n",reinas);
      system("PAUSE");
            }

void dam(int **matriz,int **tablero,int fila,int columna,int reinas,int R){
    matriz[fila][columna]=1;
    tablero[fila][columna]=1;
    marcar(matriz,R,fila,columna);
    if(reinas==R)
    solucion(tablero,reinas);
    else{
    for(int j=0;j<R;j++)
    if(matriz[j][columna+1]==0)
    dam(matriz,tablero,j,columna+1,reinas+1,R);}
    regresar(matriz,tablero,fila,columna,R);
    }

void regresar(int **matriz,int **tablero,int fila, int columna,int R){
       tablero[fila][columna] = 0;
       for(int i=0; i<R; i++){
          for(int j=0; j<R; j++){
             matriz[i][j] = 0;
          }
       }

       for(int i=0; i<R; i++){
          for(int j=0; j<R; j++){
             if(tablero[i][j]==1) marcar(matriz,R, i,j);
          }
       }
   }

void solucion(int **vect,int reinas){
    printf("Solucion %d \n",++cont);
```

```
    for(int i=0;i<reinas;i++){
                        for(int j=0;j<reinas;j++){
                                printf("%d ",vect[i][j]); }printf("\n");}
                                printf("\n\n");
    }

void vaciar(int **vect,int reinas){
    for(int i=0;i<reinas;i++)
    for(int j=0;j<reinas;j++)
    vect[i][j]=0;
    }

void marcar(int **matriz,int reinas,int falfil,int calfil){
    for(int fila=0;fila<reinas;fila++)
        for(int columna=0;columna<reinas;columna++)
            if((fila+columna==falfil+calfil)||(fila-columna==falfil-calfil))
                matriz[fila][columna]=1;
    for(int fila=0;fila<reinas;fila++){
                        matriz[falfil][fila]=1;
                        matriz[fila][calfil]=1;
                        }
    }
```

9.2 Hamiltonian Cycles (Camino Hamiltoniano)

En el campo matemático de la teoría de grafos, un **camino hamiltoniano** en un grafo es un camino, una sucesión de aristas adyacentes, que visita todos los vértices del grafo una sola vez. Si además el último vértice visitado es adyacente al primero, el camino es un **ciclo hamiltoniano**.

Los caminos y ciclos hamiltonianos fueron nombrados después que William Rowan Hamilton, inventor del *juego de Hamilton*, lanzara un juguete que involucraba encontrar un ciclo hamiltoniano en las aristas de un grafo de un dodecaedro. Hamilton resolvió este problema usando cuaterniones, pero esta solución no se generaliza a todos los grafos.

Definición

Un **camino hamiltoniano** es un camino que pasa por cada vértice exactamente una vez. Un grafo que contiene un camino hamiltoniano se denomina un **ciclo hamiltoniano** o **circuito hamiltoniano** si es un ciclo que pasa por cada vértice exactamente una vez (excepto el vértice del que parte y al cual llega). Un grafo que contiene un ciclo hamiltoniano se dice **grafo hamiltoniano.**

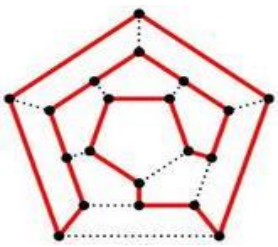


Fig. 9.4 Ejemplo de un ciclo Hamiltoniano.

El vector solución por backtraking (x_1, x_2, x_3, . . . , x_n) se define de tal forma que x_i representa el i-ésimo vértice visitado del ciclo propuesto. Ahora, todo lo se tiene que hacer es determinar como calcular el conjunto posible de vértices para x_k si x_1, . . . x_{k-1} han sido ya escogidos. Si k=1 entonces X(1) puede ser cualquiera de los n vértices. Para evitar la impresión del mismo ciclo n veces se requiere que X(1) =1. Si 1 < k < n entonces X(k) puede ser cualquier vértice v el cual es distinto de X(1), X(2), . . . , X(k-1) y v es conectado por una arista a X(k-1). X(n) puede ser sólo un vértice restante y debe ser conectado a ambos X(n-1) y X(1).

```
Procedure  NEXTVALUE(K)
//X(1),… X(k-1) es un trayecto de k-1 vertices. Si (X(k)=0 entonces no se ha asignado
//un vértice a X(K). Despues de la ejecución de X(k). Después de la ejecución de X(k) //es
asignado al siguiente vértice numerado mayor el cual (i) aun no aparece en X(1), . . // , X(k-
1). De otra manera X(k)=0. Si k=n entonces en adición X(k) se conecta a X(1).
Global integer n, X(1:n), Boolean GRAPH(1:n, 1:n)
Integer k, j
Loop
        X(k)← (X(k)+1) mod (n+1)  //el siguiente vértice.
        if  X(k) =0 then return  endif
        if GRAPH (X(k-1), X(k)) then //existe una arista?
                For j←1 to k-1 do //verificación de distinción
                        If (X(j)=X(k) then
                                Exit
                        Endif
                Repeat
                If j=k then //si es verdadero entonces el vértice es distinto.
                        If k<n or (k=n and GRAPH(X(n),1)) then return
                        Endif
                Endif
        Endif
Repeat
End NEXTVALUE
```

Usando el procedimiento NEXTVALUE se puede particularizar el esquema recursivo de backtracking para encontrar todos los ciclos Hamiltonianos.

```
Procedure  HAMILTONIAN (K)
//Este procedimiento usa una formulación recursiva de backtracking para encontrar
//todos los ciclos Hamiltonianos de la gráfica. La gráfica se guarda  como una matriz
//adyacente en GRAPH(1:n, 1:n). Todo ciclo inicia en el vértice 1.
global integer X(1:n)
local integer k, n
loop //genera valores para X(k)
        call NEXTVALUE(K)
        if X(k)=0 then return endif
        if k=n then
                print(X, ‘1’)
        else call HAMILTONIAN(k+1)
        endif
repeat
end HAMILTONIAN
```

.
Este procedimiento primero inicializa la matriz adyacente GRAPH(1:n, 1:n), a continuación establece X(2:n)←0, X(1)←1 y ejecuta la llamada a HAMILTONIAN(2).
El problema del agente viajero es un ciclo Hamiltoniano con la diferencia de que cada arista tiene un costo diferente.

Ejercicios

1. Explicar el problema de satisfabilidad
2. Revise el problema de colorear gráficas y su solución por backtracking
3. Comente sobre la solución para el camino Hamiltoniano.
4. Realice un árbol de decisión para 6 reinas.

X RAMIFICACIÓN Y ACOTAMIENTO. (Branch and Bound)

(Ellis horowitz, 1978)
(Dasgupta, Papadimitriou, & Vazirani, 2008)

El método de diseño de algoritmos **Ramificación y Acotamiento** (también llamado *Ramificación y Poda*) es una variante del Backtracking mejorado sustancialmente. El término (del inglés, *Branch and Bound*) se aplica mayoritariamente para resolver cuestiones o problemas de optimización.

La técnica de **Ramificación y Acotamiento** se suele interpretar como un árbol de soluciones, donde cada rama nos lleva a una posible solución posterior a la actual. La característica de esta técnica con respecto a otras anteriores (y a la que debe su nombre) es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas, para «podar» esa rama del árbol y no continuar malgastando recursos y procesos en casos que se alejan de la solución óptima.

10.1 Descripción General

Nuestra meta será encontrar el valor mínimo de una función f(x) (un ejemplo puede ser el coste de manufacturación de un determinado producto) donde fijamos x rangos sobre un determinado conjunto S de posibles soluciones. Un procedimiento de ramificación y poda requiere dos herramientas.

La primera es la de un procedimiento de expansión, que dado un conjunto fijo S de candidatos, devuelve dos o más conjuntos más pequeños S_1 , S_2 , … , S_n cuya unión cubre S. Nótese que el mínimo de f(x) sobre S es min{ V_1, V_2, … } donde cada v_i es el mínimo de f(x) sin S_i. Este paso es llamado ramificación; como su aplicación es recursiva, esta definirá una estructura de árbol cuyos nodos serán subconjuntos de S.

La idea clave del algoritmo de ramificación y poda es: si la menor rama para algún árbol nodo (conjunto de candidatos) A es mayor que la rama padre para otro nodo B, entonces A debe ser descartada con seguridad de la búsqueda. Este paso es llamado poda, y usualmente es implementado manteniendo una variable global *m* que graba el mínimo nodo padre visto entre todas las subregiones examinadas hasta entonces. Cualquier nodo cuyo nodo hijo es mayor que *m* puede ser descartado. La recursión se detiene cuando el conjunto candidato S es reducido a un solo elemento, o también cuando el nodo padre para el conjunto S coincide con el nodo hijo. De cualquier forma, cualquier elemento de S va a ser el mínimo de una función sin S.

El pseudocódigo del algoritmo de Ramificación y poda es el siguiente:

```
Funcion RyP {
 P = Hijos(x,k)
 mientras ( no vacio(P) )
    x(k) = extraer(P)
    si esFactible(x,k) y G(x,k) < optimo
       si esSolucion(x)
            Almacenar(x)
       sino
```

```
        RyP(x,k+1)
}
```

Donde:

- **G(x)** es la función de estimación del algoritmo.
- **P** es la pila de posibles soluciones.
- **esFactible** es la función que considera si la propuesta es válida.
- **esSolución** es la función que comprueba si se satisface el objetivo.
- **óptimo** es el valor de la función a optimizar evaluado sobre la mejor solución encontrada hasta el momento.
- **NOTA**: Usamos un menor que (<) para los problemas de **minimización** y un mayor que (>) para problemas de **maximización**.

Subdivisión Efectiva

La eficiencia de este método depende fundamentalmente del procedimiento de expansión de nodos, o de la estimación de los nodos padres e hijos. Es mejor elegir un método de expansión que provea que no se solapen los subconjuntos para ahorrarnos problemas de duplicación de ramas.

Idealmente, el procedimiento se detiene cuando todos los nodos del árbol de búsqueda están podados o resueltos. En ese punto, todas las subregiones no podadas, tendrán un nodo padre e hijo iguales a una función global mínima. En la práctica el procedimiento a menudo termina, cuando finaliza un tiempo dado, hasta el punto que el mínimo de nodos hijos y el máximo de nodos padres sobe todas las secciones no podadas, definen un rango de valores que contienen el mínimo global. Alternativamente, sin superar un tiempo restringido, el algoritmo debe terminar cuando un criterio de error, tal que (max-min)/(max+min), cae bajo un valor específico.

La eficiencia del método depende especialmente de la efectividad de los algoritmos de ramificación y poda usados. Una mala elección puede llevarnos a una repetida ramificación, sin poda, hasta que las subregiones se conviertan en muy pequeñas. En ese caso el método seria reducido a una exhaustiva enumeración del dominio, que es a menudo impracticablemente larga. No hay un algoritmo de poda universal que trabaje para todos los problemas, pero existe una pequeña esperanza de que alguna vez se encuentre alguno. Hasta entonces necesitaremos implementar cada uno por separado para cada aplicación informática, con el algoritmo de ramificación y poda especialmente diseñado para él.

Los métodos de ramificación y poda deben ser clasificados acorde a los métodos de poda, y a las maneras de creación/clasificación de los árboles de búsqueda.

La estrategia de diseño de ramificación y poda, es muy similar al de vuelta atrás (backtracking), donde el estado del árbol es usado para resolver un problema. Las diferencias son que el método de ramificación y poda no nos limitan a ninguna forma particular de obtener un árbol transverso, y es usado solamente para problemas de optimización.

10.2 Estrategias de Poda

Nuestro objetivo principal será eliminar aquellos nodos que no lleven a soluciones buenas. Podemos utilizar dos estrategias básicas. Supongamos un problema de maximización donde se han recorrido varios nodos i=1,…,n. estimando para cada uno la cota superior CS(xi) e inferior CI(x_i).

Vamos a trabajar sobre un problema donde se quiere maximizar el valor (si fuese un problema de minimización entonces se aplicaría la estrategia equivalente).

Estrategia 1

Si a partir de un nodo x_i se puede obtener una solución válida, entonces se podrá podar dicho nodo si la cota superior CS(x_i) es menor o igual que la cota inferior CI(x_j) para algún nodo *j* generado en el árbol.

Por ejemplo: Supongamos el problema de la mochila, donde utilizamos un árbol binario. Entonces:

Si a partir de x_i se puede encontrar un beneficio máximo de CS(x_i) = 4 y a partir de x_j, se tiene asegurado un beneficio mínimo de CI(x_j) = 5, esto nos llevará a la conclusión de que se puede podar el nodo x_i sin que perdamos ninguna posible solución óptima.

Estrategia 2

Si se obtiene una posible solución válida para el problema con un beneficio B_j, entonces se podrán podar aquellos nodos x_i cuya cota superior CS(x_i) sea menor o igual que el beneficio que se puede obtener B_j (este proceso sería similar para la cota inferior).

10.3 Estrategias de Ramificación

Como se comentó en la introducción de éste apartado, la expansión del árbol con las distintas estrategias está condicionada por la búsqueda de la solución óptima. Debido a esto, todos los nodos de un nivel deben ser expandidos antes de alcanzar un nuevo nivel, cosa que es lógica ya que para poder elegir la rama del árbol que va a ser explorada, se deben conocer todas las ramas posibles.

Todos estos nodos que se van generando y que no han sido explorados se almacenan en lo que se denomina Lista de Nodos Vivos (a partir de ahora LNV), nodos pendientes de expandir por el algoritmo.

La LNV contiene todos los nodos que han sido generados pero que no han sido explorados todavía. Según como estén almacenados los nodos en la lista, el recorrido del árbol será de uno u otro tipo, dando lugar a las tres estrategias que se detallan a continuación.

Estrategia FIFO

En la estrategia FIFO (First In First Out), la LNV será una cola, dando lugar a un recorrido en anchura del árbol.

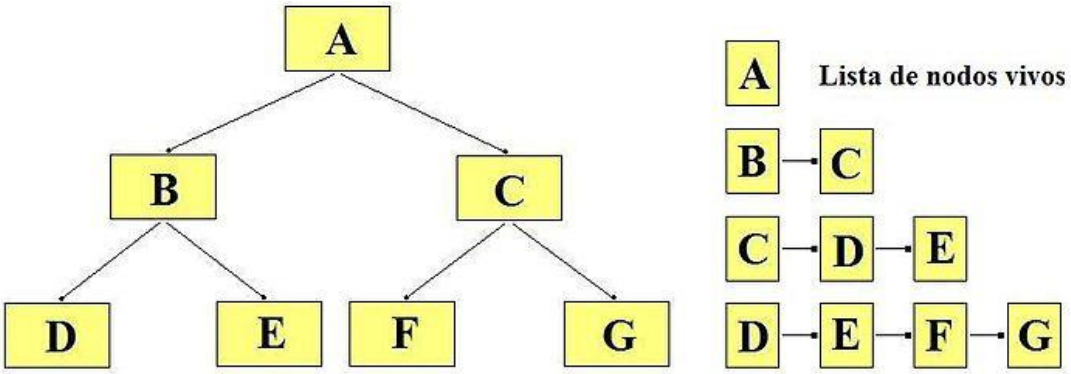


Fig. 10.1 Estrategias de ramificación FIFO.

En la figura se puede observar que se comienza introduciendo en la LNV el nodo A. Sacamos el nodo de la cola y se expande generando los nodos B y C que son introducidos en la LNV. Seguidamente se saca el primer nodo que es el B y se vuelve a expandir generando los nodos D y E que se introducen en la LNV. Este proceso se repite mientras que quede algún elemento en la cola.

Estrategia LIFO

En la estrategia LIFO (Last In First Out), la LNV será una pila, produciendo un recorrido en profundidad del árbol.

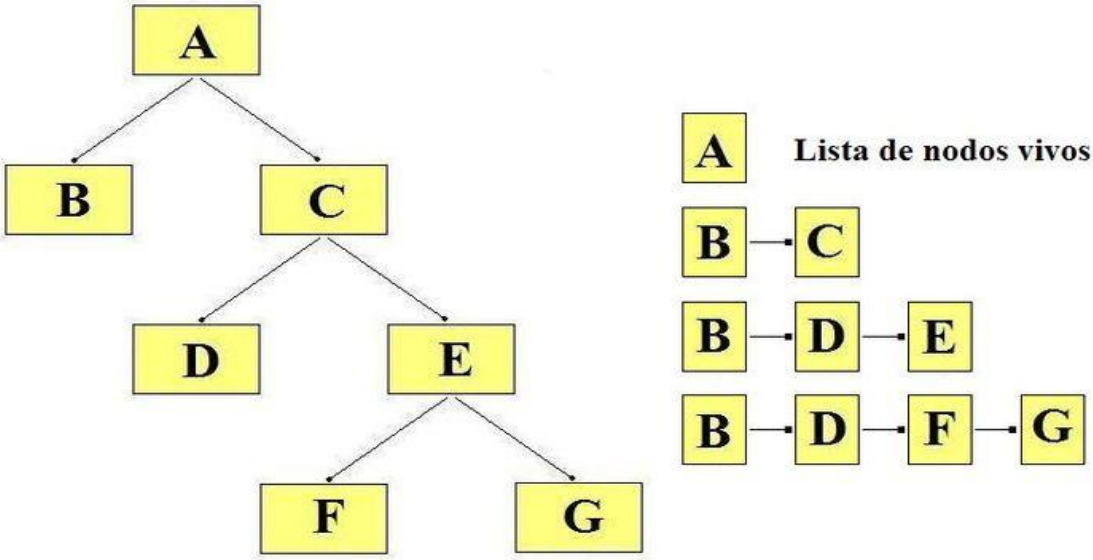


Fig. 10.2 Estrategias de ramificación LIFO

En la figura se muestra el orden de generación de los nodos con una estrategia LIFO. El proceso que se sigue en la LNV es similar al de la estrategia FIFO, pero en lugar de utilizar una cola, se utiliza una pila.

Estrategia de Menor Coste o LC

Al utilizar las estrategias FIFO y LIFO se realiza lo que se denomina una búsqueda "a ciegas", ya que expanden sin tener en cuenta los beneficios que se pueden alcanzar desde cada

nodo. Si la expansión se realizase en función de los beneficios que cada nodo reporta (con una "visión de futuro"), se podría conseguir en la mayoría de los casos una mejora sustancial.

Es así como nace la estrategia de Menor Coste o LC (Least cost), selecciona para expandir entre todos los nodos de la LNV aquel que tenga mayor beneficio (o menor coste). Por tanto, ya no estamos hablando de un avance "a ciegas".

Esto nos puede llevar a la situación de que varios nodos puedan ser expandidos al mismo tiempo. De darse el caso, es necesario disponer de un mecanismo que solucione este conflicto:

-Estrategia LC-FIFO: Elige de la LNV el nodo que tenga mayor beneficio y en caso de empate se escoge el primero que se introdujo.

-Estrategia LC-LIFO: Elige de la LNV el nodo que tenga mayor beneficio y en caso de empate se escoge el último que se introdujo.

Ramificación y Poda "Relajado"

Un variante del método de ramificación y poda más eficiente se puede obtener "relajando" el problema, es decir, eliminando algunas de las restricciones para hacerlo más permisivo.

Cualquier solución válida del problema original será solución válida para el problema "relajado", pero no tiene por qué ocurrir al contrario. Si conseguimos resolver esta versión del problema de forma óptima, entonces si la solución obtenida es válida para el problema original, esto querrá decir que es óptima también para dicho problema.

La verdadera utilidad de este proceso reside en la utilización de un método eficiente que nos resuelva el problema relajado. Uno de los métodos más conocidos es el de Ramificación y Corte (Branch and Cut (versión inglesa)).

Ramificación y Corte

Ramificación y corte es un método de optimización combinatoria para resolver problemas de enteros lineales, que son problemas de programación lineal donde algunas o todas las incógnitas están restringidas a valores enteros. Se trata de un híbrido de ramificación y poda con métodos de planos de corte.

Este método resuelve problemas lineales con restricciones enteras usando algoritmos regulares simplificados. Cuando se obtiene una solución óptima que tiene un valor no entero para una variable que ha de ser entera, el algoritmo de planos de corte se usa para encontrar una restricción lineal más adelante que sea satisfecha por todos los puntos factibles enteros. Si se encuentra esa desigualdad, se añade al programa lineal, de tal forma que resolverla nos llevará a una solución diferente que esperamos que sea "menos fraccional". Este proceso se repite hasta que ó bien, se encuentra una solución entera (que podemos demostrar que es óptima), ó bien no se encuentran más planos de corte.

En este punto comienza la parte del algoritmo de ramificación y poda. Este problema se divide en dos versiones: una con restricción adicional en que la variable es más grande o igual que el siguiente entero mayor que el resultado intermedio, y uno donde la variable es menor o igual que el siguiente entero menor. De esta forma se introducen nuevas variables en las bases

de acuerdo al número de variables básicas que no son enteros en la solución intermedia pero son enteros de acuerdo a las restricciones originales. Los nuevos programas lineales se resuelven usando un método simplificado y después el proceso es repetido hasta que una solución satisfaga todas las restricciones enteras.

Durante el proceso de ramificación y poda, los planos de corte se pueden separar más adelante y pueden ser o cortes globales válidos para todas las soluciones enteras factibles, o cortes locales que son satisfechos por todas las soluciones llenando todas las ramas de la restricción del subárbol de ramificación y poda actual.

10.4 TRAVELING SALESPERSON (El problema del agente viajero)

Para programación dinámica, el algoritmo del agente viajero tiene una complejidad de $O(n^2 2^n)$. Ahora, lo que se va a tratar de explicar es el algoritmo visto bajo la óptica de ramificación y acotamiento. El uso de una buena función de acotamiento permitirá que el algoritmo, bajo el paradigma de ramificación y acotamiento, en algunos casos pueda ser resuelto en mucho menor tiempo que el requerido en programación dinámica.
Sea G=(V,E) una gráfica definida como una instancia del problema del agente viajero y sea c_{ij} el costo de la arista <i, j>, $c_{ij} = \infty$ si <i, j> ¢ E y sea | V | = n. Sin perdida de generalidad, se puede asumir que cualquier recorrido inicia y termina en el vértice 1. De esta forma la solución del espacio S está dado por S= {1, ∏, 1| ∏ es la permutación de (2, 3, . . ., n)}. |S|=(n – 1)! . El tamaño de S puede ser reducido restringiendo S de modo que (1, i_1, i_2, . . . , i_{n-1}, 1) € S si y solo si < i_j, . . ., i_{j+1}> € E, $0 \le j \le n-1$, $i_0 = i_n = 1$. S puede estar organizado dentro de un árbol de estados. La siguiente figura muestra la organización del árbol para el caso de una gráfica completa con |V| = 4. Cada nodo hoja L es una solución y representa el recorrido definido por el trayecto desde la raíz hasta L. El nodo 14 representa el recorrido $i_0 = 1$, $i_1 = 3$, $i_2 = 4$, $i_3 = 2$ y $i_4 = 1$.

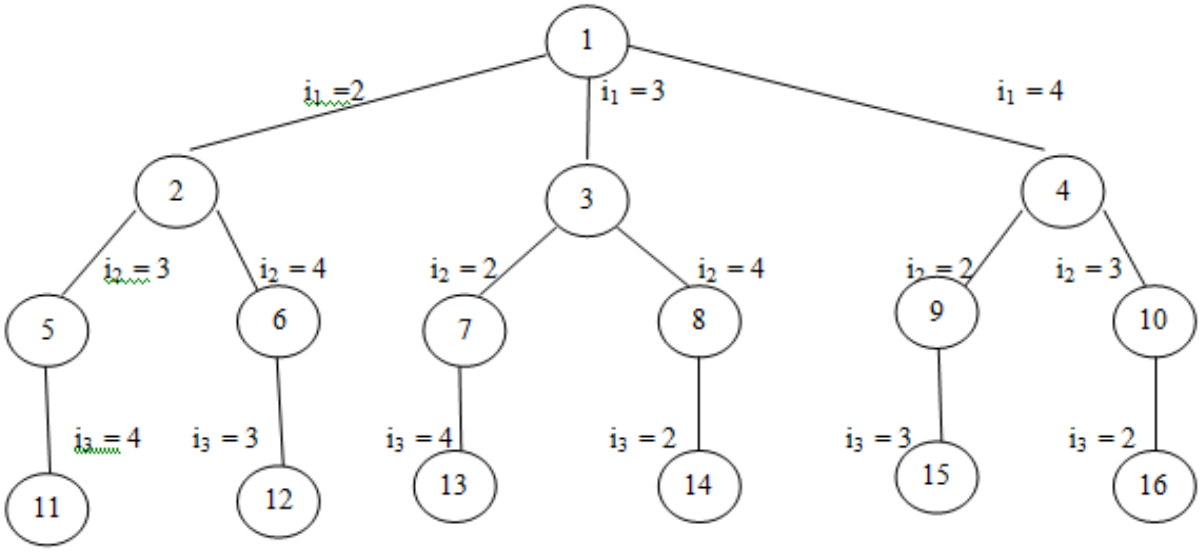


Fig. 10.3 Árbol de estados para un problema del agente viajero con n=4 y $i_0 = i_4 = 1$.

En orden a usar ramificación y acotamiento LC para buscar el árbol de estados del agente viajero, requiere definir una función de costo c(.) y otras dos funciones ĉ(.) y u(.) de tal forma que $\hat{c}(R) \le c(R) \le u(R)$ para todo nodo R. c(.) es el nodo solución si c(.) es el nodo de menor costo correspondiente al tour más corto en G. Una forma de escoger c(.) es:

$$C(A) = \begin{cases} \text{La longitud definida por la trayectoria desde la raíz a A si A es una hoja} \\ \text{El mínimo costo de la hoja en el subárbol A.} \end{cases}$$

Una simple ĉ(.) tal que ĉ(A) ≤ c(A) para todo A es obtenido definiendo ĉ(A) a ser el tamaño de la trayectoria definida en el nodo A. Por ejemplo, la trayectoria definida en el árbol anterior es i_0, i_1, i_2, = 1, 2, 4. Este consiste de las aristas <1,2> y <2, 4>. Un ĉ(.) mejor puede ser obtenido usando la matriz de costo reducida correspondiente a G. Una hilera (columna) se reduce si y solo si contiene al menos un cero y todos los demás valores son no negativos. Una matriz es reducida si y solo si toda hilera y columna es reducida. Como un ejemplo de la reducción del costo de una matriz de una gráfica dada G, consiste en la matriz de la siguiente figura:

$$\begin{vmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{vmatrix}$$

a) Matriz de costos

$$\begin{vmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{vmatrix}$$

b) Matriz de costos reducida
L = 25

Fig. 10.4 Un ejemplo.

La matriz corresponde a una gráfica con 5 vértices. Todo recorrido incluye exactamente una arista <i, j> con i=k, 1 ≤k ≤ 5 y exactamente una arista <i, j> con j=k, 1 ≤ k ≤ 5, substrayendo una constante t de todos los elementos en una hilera o una columna de la matriz de costos se reduce el tamaño de cada recorrido exactamente *t* unidades. Un recorrido de costo mínimo se mantiene después de esta operación de sustracción. Si *t* se escoge para hacer mínimo la entrada en la hilera *i* (columna *j*, restando *i* de todas las entradas en la fila *i* (columna *j*) presentará un cero en la fila *i* (columna *j*). Repitiendo este procedimiento tanto como sea necesario, la matriz de costos puede ser reducida. El monto total substraído de todas las columnas e hileras es el límite inferior y puede ser utilizado como el valor ĉ de la raíz del árbol de espacio de estado. Substrayendo 10, 2, 2, 3, 4, 1 y 3 de las hileras 1, 2, 3, 4, 5 y columnas 1 y 3 respectivamente de la matriz del inciso a) de la figura anterior se tiene la matriz reducida del inciso b) de la misma figura. El monto total substraído es 25. Por lo tanto, todo recorrido del origen a origen tiene una longitud al menos de 25 unidades.
Con todos los nodos del agente viajero del árbol de estados se puede asociar a una matriz de costos. Sea A la matriz de costos del nodo R. Sea S el hijo de R tal que la arista del árbol (R, S) corresponde a la arista <i, j> en el recorrido. Si S no es una hoja entonces la matriz de costo para S puede ser obtenida de la siguiente forma:

- Al escoger el trayecto <i, j>, cambiar todo valor en la hilera i y columna j de A por ∞. Esto previene el uso de algunas aristas salientes del vértice i o vértices entrantes de j.
- Colocar A(j,1) en ∞. Esto previene el uso de aristas <j, 1>.

- Reducir todas las hileras y columnas en la matriz resultante excepto para las hileras y columnas que contienen sólo ∞. Cada diferencia a cero se suma en la variable "r". La matriz resultante será B.
- ĉ(S) = ĉ(R) + A<i, j> + r
  Siendo "S" el número de nodo actual.
  Siendo "R" el número de nodo padre.

Los dos primeros pasos son validos y no existirá un recorrido en el sub árbol S que contenga las aristas del tipo <j, k> o <k, j> o <j, 1> (excepto para la arista <i, j>). En este momento "r" es el monto total substraído del paso 3, entonces ĉ(S) = ĉ(R) + A<i, j> + r. Para los nodos hoja ĉ(.) = c() es fácil calcular ya que cada rama hasta la hoja define un único recorrido. Para la función de la cota superior u, se requiere usar u(R) = ∞ para todo nodo R.

Para ver el árbol de transición se tomará el siguiente ejemplo:

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

Tomando la notación de la transición:

$I_s=y$

Dónde:

I. Indica en este caso, transición.
s. Indica el nivel. //nodo hijo.
y. Indica la columna.

De esta forma el árbol de trayectorias para el ejemplo quedaría de la siguiente forma:

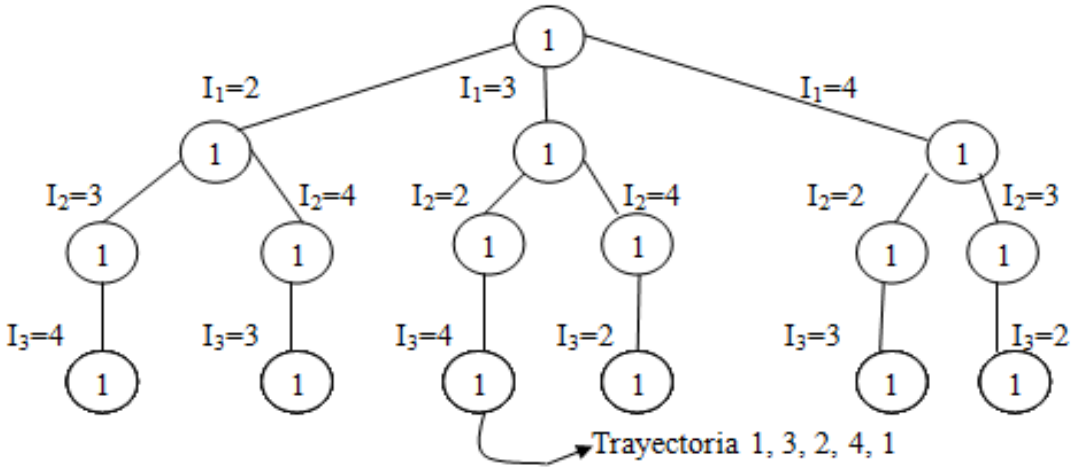


Fig. 10.5. Posibles trayectos.

Para localizar la cota inferior se realiza el siguiente paso:

Realizar la reducción de la matriz. La suma de todas las diferencias será la cota inferior ĉ(.).

Retomando el primer ejemplo del agente viajero se tiene:

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Restando por hileras:

A la h_1 se resta 10. Por lo tanto, r=10
A la h_2 se resta 2. Por lo tanto, r= 12
A la h_3 se resta 2. Por lo tanto r= 14
A la h_4 se resta 3. Por lo tanto r= 17
A la h_5 se resta 4. Por lo tanto r= 21

La matriz resultante es:

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞

Restando por columna se tiene

A la c_1 se resta 1. Por lo tanto, r=22
A la c_3 se resta 3. Por lo tanto, r=25

De esta forma ĉ(.)=25 y la cota superior U=∞.
Por lo que la matriz resultante es:

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Para S = 2 (1,2):

Ya sabiendo el costo menor, se escoge la primera parte del trayecto, donde A<2,1>=∞.

∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

A<1, 2> = 10
ĉ(2)=ĉ(.)+A<i, j> + r
ĉ(2)= 25 + 10 + 0 = 35

En este caso, en toda hilera y en toda columna existe un cero, por lo que r=0.

Para S=3 (1,3):

∞	∞	∞	∞	∞
12	∞	∞	2	0
∞	3	∞	0	2
15	3	∞	∞	0
11	0	∞	12	0

A<3,1>=∞.

En este caso, toda hilera tiene al menos un cero, pero la primer columna es diferente de cero, por lo que r=11.

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

$\hat{c}(3) = \hat{c}(.) + A<1, 3> + r$
$\hat{c}(3) = 25 + 17 + 11 = 53$

Para S = 4 (1,4):

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

A<4, 1> = ∞.
A<1, 4> = 0.
En este caso, toda hilera y toda columna tiene al menos un cero.

$\hat{C}(4) = 25 + 0 + 0 = 25.$

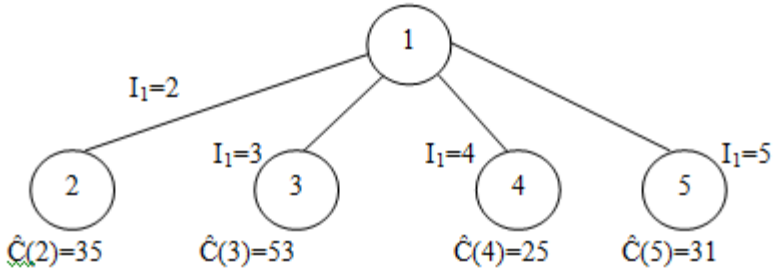
Para S = 5 (1,5):

∞	∞	∞	∞	∞
12	∞	11	2	∞
0	3	∞	0	∞
15	3	12	∞	∞
∞	0	0	12	∞

A<5, 1>=∞
En este caso, la hilera dos y la hiera cuatro tienen todos sus valores diferentes a cero, por lo que se crea otra matriz de la siguiente forma:

∞	∞	∞	∞	∞
10	∞	9	0	∞
0	3	∞	0	∞
12	0	9	∞	∞
∞	0	0	12	∞

Para la hilera 2, r=2. Para la hilera 4, r=3
r=2 + 3=5
$\hat{c}(4) = \hat{c}(.) + A<1, 5> + r$
$\hat{c}(4) = 25 + 1 + 5 = 31$



Se toma el nodo de menor costo. Por lo que el nodo padre es S=4.

Para S=6 (4,2):

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

A<2, 1> = ∞
Toda hilera y columna no marcada tiene al menos un cero. Por lo que r=0.
ĉ(6) = 25 + 3 =28

Para S=7 (4, 3):

∞	∞	∞	∞	∞
12	∞	∞	∞	0
∞	3	∞	∞	2
∞	∞	∞	∞	∞
11	0	∞	∞	∞

A<3,1> = ∞
En la columna 1 todos los números son diferentes de cero, por lo tanto r=11.
De esta forma, la nueva matriz es:

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	3	∞	∞	2
∞	∞	∞	∞	∞
0	0	∞	∞	∞

En la hilera 3 todos los números son diferentes de cero, por lo tanto r=11 + 2 = 13.
De esta forma, la nueva matriz es:

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	1	∞	∞	0
∞	∞	∞	∞	∞
0	0	∞	∞	∞

Checando el valor A<4,3> en S=4 se tiene el valor de 12. Por lo tanto:
ĉ(7)=25+12+13=50

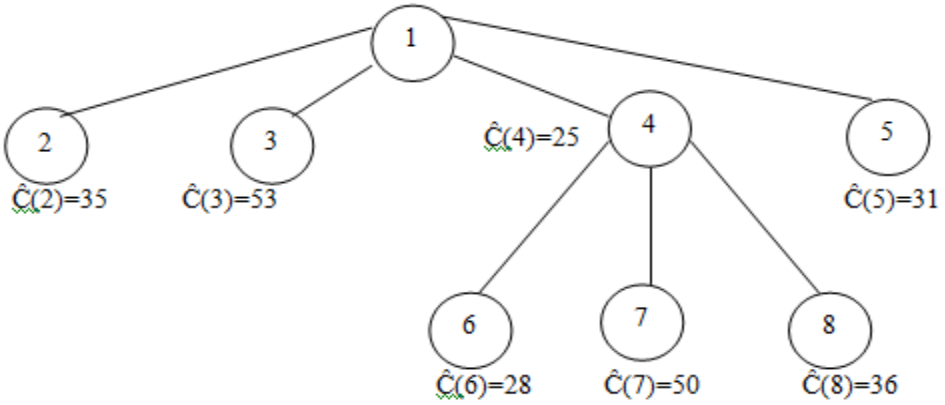
S=8 (4,5)

∞	∞	∞	∞	∞
12	∞	11	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

A<5,1>=∞
En la hilera 2 todos los números son diferentes de cero, por lo tanto r= 11.
De esta forma, la nueva matriz es:

∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

Checando el valor A<4, 5> en S=4 se tiene el valor de 0. Por lo tanto:
ĉ(8)=25 + 0 + 11 = 36



Siendo S=6 la ruta mínima se tiene:

S= 9 (2,3):

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	∞	∞	∞

A<3, 1>=∞
En las hileras 3 y 5 todos los números son diferentes de cero, por lo tanto:
r=2 + 11 = 13
Quedando la nueva matriz:

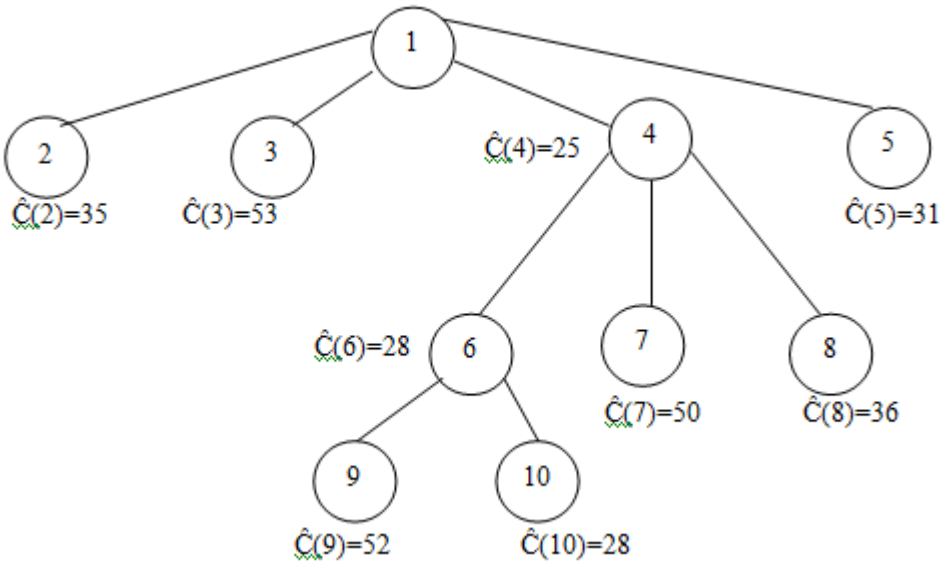
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	0
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

En S=6, A<2, 3> = 11
ĉ(9)=28 + 11 + 13 = 52

S=10 (2,5):

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

A<5, 1> = ∞
r=0
En S=6, A<2, 5>=0.
ĉ(10)=28 + 0 + 0 = 28



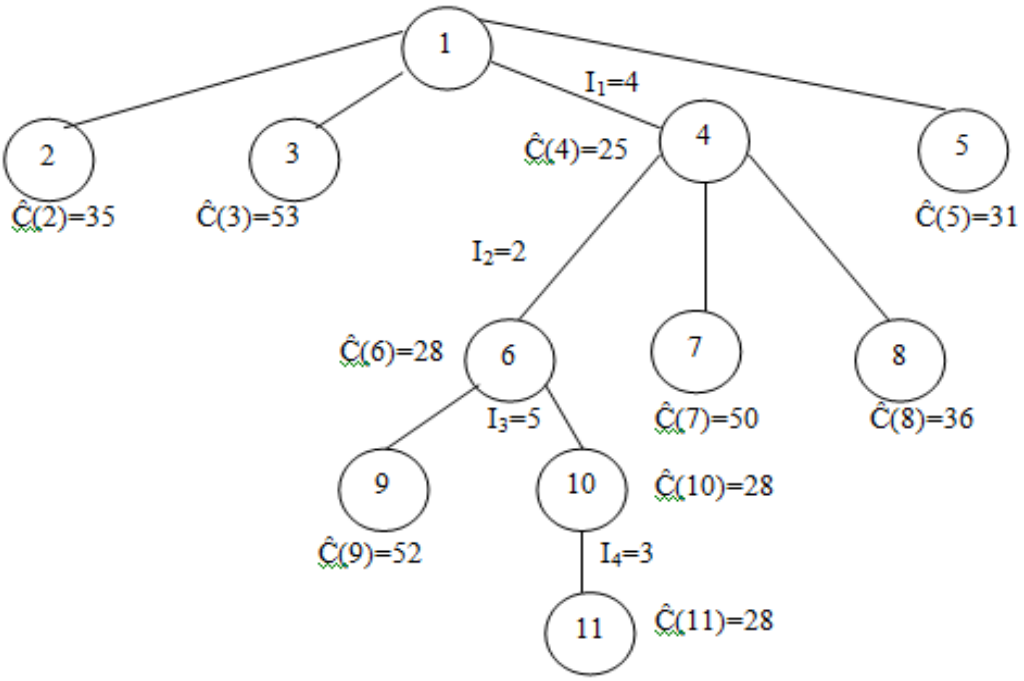
Se observará que el nodo hoja con el menor costo es S=10.

S=11, (5,3):

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

En este caso, es fundamental ir del nodo 3 al nodo 1. Por lo que A<3,1> no se modifica. Ahora bien, en S=6, A<5,3> = 0, de esta forma r=0.

Por lo tanto::
Ĉ(11)=28 + 0 + 0 = 28.



Sumando los trayectos a los nodos se tiene:

Del nodo 1 al nodo 4	10 unidades
Del nodo 4 al nodo 2	6 unidades
Del nodo 2 al nodo 5	2 unidades

Del nodo 5 al nodo 3 7 unidades
Del nodo 3 al nodo 1 3 unidades.
TOTAL 28 unidades.

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

struct Lista{
      int **matriz,*marcas,costo,contador,ciudad,mc;
      Lista *sig;
      };

void bloquear(Lista *p,int tam,int x,int y){              //bloquea la hilera y fila, ademas A<j,i>
                for(int i=0;i<tam;i++){
                      p->matriz[x][i]=999;
                      p->matriz[i][y]=999;
                      }
                      p->matriz[y][x]=999;
}

void restar_fila(Lista *q,int tam,int min,int i){         //Resta el minimo de cada fila
    for(int k=0;k<tam;k++)
    if(q->matriz[i][k]!=999 && q->matriz[i][k]!=0)
    q->matriz[i][k]-=min;
    }

void restar_columna(Lista *q,int tam,int min,int i){     //Resta el minimo de cada columna
    for(int k=0;k<tam;k++)
    if(q->matriz[k][i]!=999 && q->matriz[k][i]!=0)
    q->matriz[k][i]-=min;
    }

void costo(Lista *q,int tam){                            //Realiza el costo con el apoyo de la funcion
restar_columna y restar_fila
    int min;
    for(int i=0;i<tam;i++){
          min=q->matriz[i][0];
     for(int j=1;j<tam;j++){
           if(min>q->matriz[i][j])
           min=q->matriz[i][j];
           }
           if(min!=999)
           q->costo+=min;
           if(min!=0 && min!=999)
                  restar_fila(q,tam,min,i);
          }
     for(int i=0;i<tam;i++){
          min=q->matriz[0][i];
     for(int j=1;j<tam;j++){
           if(min>q->matriz[j][i])
```

```
            min=q->matriz[j][i];
            }
            if(min!=999)
            q->costo+=min;
            if(min!=0 && min!=999)
                  restar_columna(q,tam,min,i);
            }
      }

void generar(Lista *q,int tam){                        //Genera la matriz de costos minimos
      int i,j,cont=0,destino;
      for(i=0;i<tam;i++)
            for(j=0;j<tam;j++)
                  q->matriz[i][j]=999;
      printf("Para dejar de intorducir una ciudad introduce 99");
      while(cont<tam){
                  printf("\n       Ciudad %d       \n  ",cont+1);
                  do{
                  printf("De ciudad %d a: ",cont+1);
                  scanf("%d",&destino);
                  if(destino==99)
                  printf("Termino ciudad %d",cont+1);
                  else if(destino>tam || destino<=0)
                  printf("Esa ciudad no existe\n");
                  else if(cont==destino-1)
                  printf("Te encuentras en esta ciudad\n");
                  else{
                      printf("Introduce el costo de %d a %d: ",cont+1,destino);
                      scanf("%d",&q->matriz[cont][destino-1]);
                      }
                    }while(destino!=99);
                    cont++;
                  }
      }

void *guardar(void *p,Lista *mmin,int tam,int num,int ciudad,int j){
      //Genera la lista de matricez con costo, caminos por recorrer, marcas y ciudad en la //que se encuentra
      Lista *aux, *q;
      int min;
      q=(Lista *)malloc(sizeof(Lista));
      if(p==NULL){
                  q->matriz=(int **)malloc(sizeof(int *)*tam);
                  q->marcas=(int *)malloc(sizeof(int)*tam);
                    for(int i=0;i<tam;i++){
                          q->matriz[i]=(int *)malloc(sizeof(int)*tam);
                          q->marcas[i]=0;
                          }
                  q->contador=num;
                  q->marcas[ciudad]=1;
```

```
                q->sig=NULL;
                q->ciudad=j;
                q->costo=0;
                q->mc=1;
                generar(q,tam);
                costo(q,tam);
                p=q;
            }
    else{
                q->matriz=(int **)malloc(sizeof(int *)*tam);
                q->marcas=(int *)malloc(sizeof(int)*tam);
                  for(int i=0;i<tam;i++){
                        q->matriz[i]=(int *)malloc(sizeof(int)*tam);
                        q->marcas[i]=mmin->marcas[i];
                        }
                for(int i=0;i<tam;i++)
                for(int j=0;j<tam;j++)
                q->matriz[i][j]=mmin->matriz[i][j];
                q->contador=num;
                q->marcas[ciudad]=1;
                q->sig=NULL;
                q->ciudad=j;
                q->costo=0;
                q->mc=0;
                min=q->matriz[ciudad][j];
                if(min!=999)
                q->costo=mmin->costo+min;
                else
                q->costo=mmin->costo;
                bloquear(q,tam,ciudad,j);
                costo(q,tam);
        aux=(Lista *)p;
        while(aux->sig!=NULL)
                        aux=aux->sig;
        aux->sig=q;
        }
        return p;
    }

int main(){
      int tam,num=1,ciudad=0,min;
      Lista *mmin,*aux;
      void *p=NULL;
      printf("Introduce el numero de ciudades: ");
      scanf("%d",&tam);
      p=guardar(p,mmin,tam,num,0,0);
      mmin=(Lista *)p;
      do{
 //Mientras el num no sea igual al numero de ciudades el arbol no ha terminado, sigue
//recorriendo los caminos que hacen falta y detectando los menores
```

```
                aux=(Lista *)p;
                num=num+1;
                for(int i=0;i<tam;i++)
                if(mmin->marcas[i]!=1)
                p=guardar(p,mmin,tam,num,ciudad,i);
                min=999;
                while(aux!=NULL){
                        if(aux->costo<min && aux->mc!=1){
                                min=aux->costo;
                                mmin=aux;
                                }
                                aux=aux->sig;
                        }
                        ciudad=mmin->ciudad;
                        mmin->marcas[ciudad]=1;
                        mmin->mc=1;
                        num=mmin->contador;
                }while(num!=tam);
                printf("\n\n   Recorrido de costo minimo \n\nCosto minimo: %d\n",mmin->costo);
system("pause");
```

Ejercicios

1. Resolver el agente viajero para la siguiente matriz:

∞	11	10	9	6
8	∞	7	3	4
8	4	∞	4	8
11	10	5	∞	5
6	9	5	5	∞

2. Resolver el problema 1 utilizando programación dinámica.
3. Resolver el problema 1 utilizando backtracking

XI PROBLEMAS NP.

(Ellis horowitz, 1978)
(Dasgupta, Papadimitriou, & Vazirani, 2008)
(Garey & Johnson, 1975)
(Dewdney, 1989)
(Kewis & Papadimitriou, 1989)
(Penrose, 1989)
(Singh, 1995)
(Alfonseca Cubero, Alfonseca Moreno, & Moriyon, 2007)
(Deutsch)

11.1 Antecedentes de complejidad.

La idea de disponer de un algoritmo, o receta para efectuar alguna tarea ha existido durante miles de años. También durante muchos años la gente creyó que si cualquier problema podía iniciarse de manera precisa, entonces con suficiente esfuerzo sería posible encontrar una solución con el tiempo (o tal vez una prueba que no existe solución podría proporcionarse con el transcurso del tiempo). En otras palabras, se creía que no había problema que fuera tan intrínsecamente difícil que en principio nunca pudiera resolverse.

Se pensaba que las matemáticas son un sistema **consistente**, es decir, que no es posible llegar a contradicciones a partir de los axiomas iniciales. Posteriormente se amplió el problema (que se convirtió en el Entscheidungsproblem, o problema de la decisión), para incluir también la demostración que la aritmética también es **completa** (existe una prueba para toda proposición matemática correcta) y **decidible** (existe un método efectivo que decide, para cada proposición posible, si es verdadera o falsa)
Correcto y completo

Uno de los principales promotores de esta creencia fue el famoso matemático David Hilbert (1826-1943). Hilbert creía que en matemáticas todo podía y debía probarse a partir de los axiomas básicos. El resultado de ello sería demostrar de manera concluyente los dos elementos básicos del sistema matemático. En primer lugar, las matemáticas debían ser capaces, al menos en teoría, de responder a cualquier interrogante concreto. En segundo lugar, las matemáticas deberían estar libres de incongruencias, o lo que es lo mismo, una vez demostrada la veracidad de una premisa a través de un método no sería posible que mediante otro método se concluyera que esa misma premisa sea falsa. Hilbert estaba convencido de que, asumiendo tan sólo unos pocos axiomas, sería posible responder a cualquier pregunta matemática concebible sin temor a una contradicción.
El 8 de agosto de 1900 Hilbert pronunció una conferencia histórica en el Congreso Internacional de Matemáticas de París. Hilbert planteó veintitrés problemas matemáticos sin resolver que él consideraba de una perentoria importancia. Hilbert pretendía sacudir a la comunidad para que lo ayudaran a realizar su sueño de crear un sistema matemático libre de toda duda e incoherencia. Una ambición que inscribió en su lápida:

Wir műssen wissen,
Wir werden wissen.
Tenemos que saber,
Llegaremos a saber.

Al mismo tiempo, el lógico inglés Bertrand Russell, que también estaba contribuyendo al gran proyecto de Hilbert, había tropezado con una incoherencia. Russell evocó su propia reacción ante la temida posibilidad de que las matemáticas fueran intrínsecamente contradictorias.
No había escapatoria a la contradicción. El trabajo de Russell causó un perjuicio considerable al sueño de crear un sistema matemático libre de duda, incoherencia y paradoja.
La paradoja de Russell se explica a menudo con el cuento del bibliotecario minucioso.

> *Un día, deambulando entre las estanterías, el bibliotecario descubre una colección de catálogos. Hay diferentes catálogos para novelas, obras de consulta, poesía, y demás. Se da cuenta de que algunos de los catálogos se incluyen a sí mismos y otros en cambio, no.*
> *Con el objeto de simplificar el sistema, el bibliotecario elabora dos catálogos más: en uno de ellos hace constar todos los catálogos que se incluyen a sí mismos y en el otro, más interesante aún, todos aquellos que no se catalogan a sí mismos. ¿Debe catalogarse a sí mismo? Si se incluye, por definición no debería estar incluido; en cambio, si no se incluye, debería incluirse por definición. El bibliotecario se encuentra en una situación imposible.*

La incongruencia que atormenta al bibliotecario causará problemas en la estructura lógica de las matemáticas. Las matemáticas no pueden tolerar inconsistencias, paradojas o contradicciones. El poderoso instrumento de la prueba por contradicción, por ejemplo, se fundamenta en una matemática libre de paradojas. La prueba por contradicción establece que si una afirmación conduce al absurdo, entonces tiene que ser falsa; sin embargo, según Russell, incluso los axiomas puede llevar al absurdo. Así que la prueba por contradicción podría evidenciar que un axioma es falso, y no obstante los axiomas son el fundamento de las matemáticas y se reconocen como ciertos.
El trabajo de Russell sacudió los cimientos de las matemáticas y arrastró el estudio de la lógica matemática a una situación de caos. Una manera de abordar el problema era crear un axioma adicional que prohibiera a cualquier grupo ser miembro de sí mismo. Eso reduciría la paradoja de Russell y convertiría en superflua la cuestión de si hay que introducir en el catálogo aquellos otros que no se incluyen a sí mismos.
En 1910, Russell publicó el primero de los tres volúmenes de la obra *Principia Mathematica*, un intento de tratar el problema surgido de su propia paradoja. Cuando Hilbert se retiró en 1930, estaba seguro de que las matemáticas estaban bien encaminadas hacia su recuperación. Al parecer, su sueño de una lógica coherente y lo bastante sólida como para responder a cualquier pregunta se hallaba en vías de tornarse en realidad.
Pero ocurrió que en 1931 un matemático desconocido de veinticinco años de edad hizo público un artículo que destruiría para siempre las esperanzas de Hilbert. Kurt Gödel iba a forzar a los matemáticos a aceptar que las matemáticas nunca llegarían a alcanzar una lógica perfecta y sus trabajos llevaban implícita la idea de que problemas como el último teorema de Fermat pudiera ser imposible de resolver.
Gödel había demostrado que el intento de crear un sistema matemático completo y coherente era un imposible. Sus ideas se pueden recoger en dos proposiciones.

Primer teorema de indecidibilidad.
Si el conjunto de axiomas de una teoría es coherente, existen teoremas que no se pueden ni probar ni refutar.

Segundo teorema de indecibilidad.
No existe ningún proceso constructivo capaz de demostrar que una teoría axiomática es coherente.

El primer enunciado de Gödel dice básicamente que, con independencia de la serie de axiomas que se vaya a utilizar, habrá cuestiones que las matemáticas no puedan resolver; la completitud no podrá alcanzarse jamás. Peor aún, el segundo enunciado dice que los matemáticos jamás podrán estar seguros de que los axiomas elegidos no los conducirán a ninguna contradicción; la coherencia no podrá demostrarse jamás. Gödel probó que el programa de Hilbert era una tarea imposible.
A pesar de que el segundo enunciado de Gödel decía que era imposible demostrar que los axiomas fueran coherentes, eso no implicaba que fueran incoherentes. Muchos años después, el gran teórico de números André Weil dijo:

<< Dios existe porque las matemáticas son coherentes
y el diablo existe porque no podemos demostrarlo>>

La demostración de los teoremas de Gödel es tremendamente complicada pero se puede ilustrar con una analogía lógica que debemos a Epiménides y que se conoce como la paradoja cretense. Epiménides fue un cretense que afirmó:
Soy un mentiroso.
La paradoja surge cuando intentamos determinar si esta proposición es verdadera o falsa. Si la proposición es cierta, en principio afirmamos que Epiménides no es un mentiroso. Si la proposición es falsa, entonces Epiménides no es un mentiroso, pero hemos aceptado que emitió un enunciado falso y por lo tanto sí que es un mentiroso. Encontramos otra incoherencia, por lo tanto el enunciado no es ni verdadero ni falso.
Gödel dio una reinterpretación a la paradoja del mentiroso y le incorporó el concepto de demostración. El resultado fue un enunciado como el que sigue:

Este enunciado no tiene demostración.

Puesto que Gödel consiguió traducir la proposición de más arriba a una notación matemática, fue capaz de demostrar que hay enunciados matemáticos ciertos que jamás podrán probarse como tales; son los denominados enunciados **indecidibles**. Éste fue el golpe mortal para el programa de Hilbert.

Esto mostró para Hilbert que el Entscheidungsproblem no es computable. Es decir, no existe un algoritmo del tipo que buscaba Hilbert. Un cínico podría decir que los matemáticos dieron un suspiro de alivio, porque si hubiera tal algoritmo, todos se quedarían sin trabajo en cuando se le encontrara. Sin embargo, a los matemáticos les sorprendió este notable descubrimiento.

El ***Entscheidungsproblem*** (en castellano: problema de decisión) fue el reto en lógica simbólica de encontrar un algoritmo general que decidiera si una fórmula del cálculo de primer orden es un teorema. En 1936, de manera independiente, Alonzo Church y Alan Turing demostraron ambos que es imposible escribir tal algoritmo. Como consecuencia, es también imposible decidir con un algoritmo si ciertas frases concretas de la aritmética son ciertas o falsas.

La pregunta se remonta a Gottfried Leibniz, quien en el siglo XVII, luego de construir exitosamente una máquina mecánica de cálculo, soñaba con construir una máquina que pudiera manipular símbolos para determinar si una frase en matemáticas es un teorema. Lo primero que sería necesario es un lenguaje formal claro y preciso, y mucho de su trabajo

posterior se dirigió hacia ese objetivo. En 1928, David Hilbert y Wilhelm Ackermann propusieron la pregunta en su formulación anteriormente mencionada.

Una fórmula lógica de primer orden es llamada *universalmente válida* o *lógicamente válida* si se deduce de los axiomas del cálculo de primer orden. El teorema de completitud de Gödel establece que una fórmula lógica es universalmente válida en este sentido si y sólo si es cierta en toda interpretación de la fórmula en un modelo.

Antes de poder responder a esta pregunta, hubo que definir formalmente la noción de *algoritmo*. Esto fue realizado por Alonzo Church en 1936 con el concepto de "calculabilidad efectiva" basada en su cálculo lambda y por Alan Turing basándose en la máquina de Turing. Los dos enfoques son equivalentes, en el sentido en que se pueden resolver exactamente los mismos problemas con ambos enfoques.

La respuesta negativa al *Entscheidungsproblem* fue dada por Alonzo Church en 1936 e independientemente, muy poco tiempo después por Alan Turing, también en 1936. Church demostró que no existe algoritmo (definido según las funciones recursivas) que decida para dos expresiones del cálculo lambda si son equivalentes o no. Church para esto se basó en trabajo previo de Stephen Kleene. Por otra parte, Turing redujo este problema al problema de la parada para las máquinas de Turing. Generalmente se considera que la prueba de Turing ha tenido más influencia que la de Church. Ambos trabajos se vieron influidos por trabajos anteriores de Kurt Gödel sobre el teorema de incompletitud, especialmente por el método de asignar números a las fórmulas lógicas para poder reducir la lógica a la aritmética.

El argumento de Turing es como sigue: Supóngase que se tiene un algoritmo general de decisión para la lógica de primer orden. Se puede traducir la pregunta sobre si una máquina de Turing termina con una fórmula de primer orden, que entonces podría ser sometida al algoritmo de decisión. Pero Turing ya había demostrado que no existe algoritmo general que pueda decidir si una máquina de Turing se para.

Es importante notar que si se restringe el problema a una teoría de primer orden específica con constantes, predicados constantes y axiomas, es posible que exista un algoritmo de decisión para la teoría. Algunos ejemplos de teorías decidibles son: la aritmética de Presburger y los sistemas estáticos de tipos de los Lenguajes de programación.

Sin embargo, la teoría general de primer orden para los números naturales conocida como la aritmética de Peano no puede ser decidida con ese tipo de algoritmo. Esto se deduce del argumento de Turing resumido más arriba.

Además, el teorema de Gödel mostró que no existe algoritmo cuya entrada pueda ser cualquier proposición acerca de los enteros y cuya salida es o no verdadera. Siguiendo de cerca a Gödel, otros matemáticos como Alonso Church, Sephen Kleene, Emil Post, Alan Turing y muchos otros, encontraron más problemas que carecían de solución algorítmica. Tal vez la característica más notable de estos primeros resultados sobre problemas que no se pueden resolver por medio de computadoras es que se obtuvieron en la década de 1930 ¡antes de que se hubiera construido la primera computadora!

11.2 Tesis CHURCH – TURING.

Existe un obstáculo importante al probar que no existe un algoritmo para una tarea específica. Primero es necesario saber con exactitud qué significa algoritmo. Cada uno de los matemáticos mencionados en la sección anterior había superado este obstáculo y lo hizo definiendo algoritmo en forma diferente.
Gödel definió un algoritmo como una secuencia de reglas para formar funciones matemáticas complicadas a partir de funciones matemáticas más simples.
Church utilizó un formalismo denominado cálculo lambda.
Turing empleó una máquina hipotética conocida como la máquina de Turing. Turing definió un algoritmo como cualquier conjunto de instrucciones para su máquina simple.
Estas definiciones, en apariencia diferentes, y creadas de manera independiente, resultan ser equivalentes. Conforme los investigadores se dieron cada vez más cuenta de esta equivalencia en la década de los 30's, se creyó en forma amplia en las dos proposiciones siguientes:

1. Todas las definiciones razonables de "algoritmo" conocidas hasta el momento son equivalentes.
2. Cualquier definición razonable de "algoritmo" que se llegue a dar, a su vez será equivalente a las definiciones ya conocidas.

Estas creencias han llegado a denominarse *tesis de Church – Turing* en honor a dos de los primeros trabajadores que se dieron cuenta de la naturaleza fundamental del concepto que habían definido. Hasta el momento no ha existido evidencia en contra y se acepta ampliamente la tesis de Church – Turing.
En un planteamiento moderno, es posible definir "algoritmo" como cualquier cosa que pueda ejecutarse en una computadora. Dadas dos computadoras modernas, es posible escribir un programa para una de ellas que pueda comprender y ejecutarse en otra.
La equivalencia entre toda computadora moderna, y la máquina de Turing y con otros numerosos medios de definir "algoritmo", es una evidencia más de la tesis de Church – Turing. Esta propiedad de los algoritmos se conoce como *Universalidad.*
En términos informales, universalidad significa que cualquier computadora es equivalente a todas las otras en el sentido de que todas pueden efectuar las mismas tareas.

11.3 Complejidad.

El estudio de la computabilidad lleva a comprender cuáles son los problemas que admiten solución algorítmica y cuáles no. De aquellos problemas para los que existen algoritmos, también resulta de interés saber cuántos recursos de cómputo se necesitan para su ejecución. Sólo los algoritmos que utilizan una cantidad factible de recursos resultan útiles en la práctica. El campo de la ciencia de la computación denominado *teoría de la complejidad* es el que pregunta e intenta resolver cuestiones acerca del empleo de recursos de cómputo.
En la siguiente figura se muestra una representación pictórica del universo de problemas. Aquellos que pueden computarse en forma algorítmica forman un subconjunto infinitesimalmente pequeño. Los que son *factiblemente computables* tomando en cuenta sus necesidades de recursos, comprenden una diminuta porción del ya infinitesimalmente pequeño subconjunto. Sin embargo, la clase de problemas factibles computables es tan grande que la ciencia de la computación se ha vuelto una ciencia interesante, practica y floreciente.

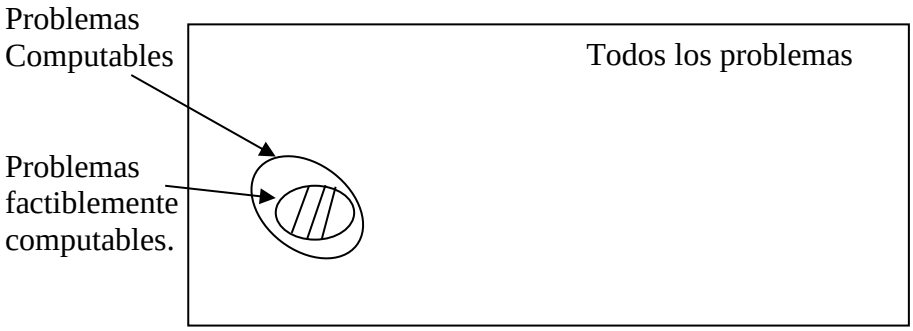


Fig. 11.1 Problemas computables y no computables.

11.4 Tesis de computabilidad secuencial.

En las secciones anteriores se describió la diferencia entre algoritmos que utilizaban cantidades polinomiales (n^c) y exponencial (c^n). Los algoritmos polin
omiales tienden a ser factibles para tamaños razonables de datos de entrada. Los algoritmos exponenciales tienden a exceder los recursos disponibles aun tratándose de cantidades pequeñas de datos de entrada. Uno de los objetivos de la teoría de complejidad es mejorar esta clasificación de los algoritmos y, por ende, la comprensión de la diferencia entre problemas factibles y no factibles.
Si un algoritmo se construye tomando dos algoritmos factibles y colocándolos en forma secuencial uno después del otro, el algoritmo así construido debe ser factible. De manera semejante, si algún algoritmo factible se reemplaza por una llamada a un módulo que representa a un segundo algoritmo factible, el nuevo algoritmo combinado también debe ser factible. Esta propiedad de cerradura en realidad se cumple para algoritmos de tiempo polinomial.
Cualquier algoritmo que se ejecuta en tiempo polinomial en una computadora puede correr en tiempo polinomial en cualquier otra. De ahí que tenga sentido hablar de algoritmos de tiempo polinomial en forma independiente de cualquier computadora especifica. Una teoría de algoritmos factibles basada en el tiempo polinomial es independiente de la máquina.

La creencia de que todas las computadoras secuenciales razonables que se llegan a crear tienen tiempos de ejecución relacionados polinomialmente recibe el nombre de *tesis de computación secuencial*. Esta tesis puede compararse con la tesis de Curch-Turing. Es una versión más fuerte de esta tesis, pues afirma no sólo que todos los problemas computables son los mismos para todas las computadoras, sino también que todos los problemas computables factibles son los mismos para todas las computadoras.

11.5 Problemas NP.

Esta sección contiene lo que tal vez el desarrollo más importante en investigación en algoritmos en la década de los 70's, no sólo en ciencias de la computación, sino también en ingeniería eléctrica, en investigación de operaciones y otras áreas relacionadas.
Una idea importante es la distinción entre un grupo de problemas cuya solución se obtiene en tiempo polinomial y un segundo grupo de problemas cuya solución no se obtiene en tiempo polinomial.

La teoría de NP-completo no provee algoritmos para resolver los problemas del segundo grupo en tiempo polinomial, tampoco dice que no exista algún algoritmo en tiempo polinomial. En lugar de eso, se explicará que todo aquel problema que no tiene en este momento un algoritmo en tiempo polinomial está computacionalmente relacionado. En realidad, se pueden establecer dos clases de problemas. Estos serán los problemas NP-duros y los NP-completos. Un problema que es NP-completo tendrá la propiedad de que se puede resolver en tiempo polinomial si y sólo si todos los demás NP-completo también se puede resolver en tiempo polinomial. Si un problema NP-duro se puede resolver en tiempo polinomial entonces todos los problemas NP-completos se pueden resolver en tiempo polinomial.
Mientras que se definen varios problemas con la propiedad de ser clasificados como NP-duros o NP-completos (problemas que no se resuelven en forma secuencial en tiempo polinomial), estos mismos problemas se pueden resolver en máquinas no determinísticas en tiempo polinomial.

11.5.1 Algoritmos no determinísticos.

Hasta este momento, los algoritmos que se han explicado tienen la propiedad de que el resultado de toda operación es única y bien definida. Algoritmos con esta propiedad se conocen como algoritmos determinísticos. Tales algoritmos se pueden ejecutar sin problema en una computadora secuencial. En una computadora teórica, se puede remover ésta restricción y permitir operaciones cuya salida no es única pero limitada a un conjunto de restricciones. La máquina que ejecute tales operaciones se le permitiría escoger alguno de los resultados sujeta a terminar bajo una condición especifica. Esto conduce al concepto de un algoritmo no determinístico. De esta forma, se definen una nueva función y dos nuevas declaraciones:

i. Choice(S) . . .en forma arbitraria se escoge un elemento del conjunto S.
j. Failure . . .señala una terminación sin éxito.
k. Success . . .señala una terminación con éxito.

X←choice(1:n) puede resultar en X la asignación de un entero en el rango [1,n]. No existe una regla especifica de cómo se escogió el número entero. La señal de éxito o no éxito se utiliza para definir un estado del algoritmo. Estas declaraciones son equivalentes a definir un stop y no son utilizadas para efectuar un retorno. Un algoritmo termina sin éxito si y sólo si no existe un conjunto de elecciones que conduzcan al éxito. Los tiempos de cómputo para choice, success, y failure son tomados como O(1). Una máquina que sea capaz de ejecutar un algoritmo no determinístico se conoce como máquina no determinística. Mientras que no exista una máquina no determinística (como la definida en este escrito), se tienen las suficientes razones intuitivas para concluir que ciertos problemas no se pueden resolver en algoritmos determinísticos.

Ejemplo. Considere el problema de buscar para un elemento x en un conjunto dado de elementos A(1:n), n ≥1. Se requiere determinar el índice tal que A(j) = x o j=0 si x no se encuentra en A. Un algoritmo no determinístico es:

```
j←choice(1:n)
if A(j) =x entonce imprime(j) success endif
print('0'); failure
```

En este ejemplo, en una computadora no determinística imprimirá un '0' si y sólo si no exista algún j tal que A(j) =x. El algoritmo es un algoritmo no determinístico con complejidad O(1).

Note que, como A no está ordenado, cualquier algoritmo determinístico de búsqueda tiene una complejidad $\Omega(n)$.
Una interpretación de un algoritmo no determinístico puede ser permitido utilizando una computadora paralela sin límites. Se puede hacer en cada instante cada choice(S), el algoritmo realiza varias copias de él mismo. Una copia para cada choice(S). Por lo que todas las copias son ejecutadas al mismo tiempo. La primera copia que termine con un sucessful obliga que terminen las demás copias. Si una copia termina en failure, sólo esa copia se detiene. Es importante indicar que una máquina no determinística no produce ninguna copia de algún algoritmo cada vez que un choice(S) sea ejecutado. Ya que la máquina es ficticia, no es necesario explicar como tal máquina determina si existe un success o un failure.

Definición. P es el conjunto de todos los problemas resoluble por un algoritmo determinístico en tiempo polinomial. NP es el conjunto de todos los algoritmos de decisión resolubles por un algoritmo no determinístico en tiempo polinomial.
Ya que algoritmos determinísticos son un caso especial de algoritmos no determinísticos, se puede concluir que P $\underline{C}$ NP. Lo que no conocemos, y lo que puede ser, tal vez, el más famoso problema no resuelto en ciencias de la computación es si P=NP o si P $\neq$ NP.
¿Es posible que para todo problema NP exista un algoritmo determinístico P en el cual no sea conocido en este momento? Esto parece no ser posible por el esfuerzo que se ha realizado por bastantes expertos para contestar tal interrogante, sin embargo, la prueba de que P $\neq$ NP es tan difícil de alcanzar y parece que requiere técnicas que aún no han sido descubiertas.
Considerando este problema, Cook formuló la siguiente pregunta: ¿ hay algún problema en NP único tal que si se lo mostró estar en P, entonces esto implicaría que N = NP? Cook respondió su propia pregunta con el siguiente teorema:

Teorema de Cook: satisfacibilidad es en P si y sólo si P = NP.

Como paréntesis se puede decir que en teoría de la complejidad computacional, el Problema de ***satisfacibilidad booleana*** (también llamado SAT) fue el primer problema identificado como perteneciente a la clase de complejidad NP-completo.

Stephen Cook demostró dicha pertenencia en 1971 utilizando una máquina de Turing no determinista (MTND) a través de la siguiente demostración:

> *Si se tiene un problema NP, entonces existe una MTND que lo resuelve en tiempo polinomial conocido p(n). Si se transforma esa máquina en un problema de satisfacibilidad (en un tiempo polinomial n) y se soluciona dicho problema, se habrá obtenido también la solución al problema NP original. En tal caso se habrá demostrado que todo problema NP se puede transformar a un problema de satisfacibilidad y por lo tanto el SAT es NP-Completo.*

El problema de satisfacibilidad booleana:

- Un problema es satisfacible si existe al menos una asignación de valores a las variables del problema que lo hagan verdadero ($\top$).

- Un problema es insatisfacible si todas las posibles asignaciones de valores hacen el problema siempre falso ($\bot$).

Veamos a esto con un ejemplo:

- Se va a partir de la siguiente proposición en forma normal disyuntiva:

$$(x_1 \vee \bar{x}_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3)$$

- Se realiza la siguiente asignación:

$$x_1 = \bot, x_2 = \bot \text{y } x_3 = \top$$

- Se sustituye en la expresión:

$$(\bot \vee \top) \wedge (\bot \vee \top) \wedge (\bot \vee \bot) \wedge (\top \vee \top)$$

- Se evalúa la expresión: $\bot$.
- Como no se ha encontrado una solución válida se hace una nueva asignación:

$$x_1 = \top, x_2 = \top \text{y } x_3 = \top$$

- Se evalúa la expresión: $\top$.

Como se ha encontrado una asignación de valores (modelo) que hacen a la expresión verdadera, se ha demostrado que este problema en concreto es *satisfacible*.

Estas son sólo dos de las ocho ($2^{n=3}$) posibles asignaciones. Se puede apreciar que el número de soluciones crece rápidamente al añadir nuevas variables, de ahí que su complejidad computacional sea elevada.

Un algoritmo creado para la resolución de problemas SAT es el siguiente:

- Algoritmo DPLL: utiliza una búsqueda hacia atrás sistemática (back tracking) para explorar las posibles asignaciones de valores a las variables que hagan al problema satisfacible.

En este momento se puede definir mejor los tipos de problemas NP-duros y NP –completo. Primero se definirá la noción de reducibilidad.

Definición: Sea L_1 y L_2 dos problemas. L_1 reduce a L_2 (L_1 α L_2) si y sólo si existe un camino para resolver L_1 con un algoritmo polinomial determinístico también se usará un algoritmo determinístico de tiempo polinomial que resuelva a L_2.
Esta definición implica que si existe un algoritmo determinístico en tiempo polinomial para L_2 entonces podemos resolver L_1 en tiempo polinomial. Este operador es transitivo, esto es, si L_1 α L_2 y L_2 α L_3 entonces L_1 α L_3.

Definición. Un problema L es NP-duro si y sólo si satisfacibilidad reduce a L (satisfacibilidad α L). Un problema L es NP-completo si y sólo si L es NP-duro y L∈NP.

Un problema NP-duro puede no ser NP-completo. Sólo un problema de decisión puede ser NP-completo. Sin embargo, un problema de optimización puede ser NP-duro. Además, si L_1 es un problema de decisión y L_2 es un problema de optimización, es bastante posible que L_1 α L_2. Se puede observar que el problema de decisión de la mochila se puede reducir al problema de optimización de la mochila. También se puede comentar que el problema de optimización se puede reducir a su correspondiente problema de decisión. Por lo tanto, problemas de optimización no pueden ser NP-completos, mientras que algunos problemas de decisión pueden ser del tipo NP-duro y no son NP-completos.

11.5.2 Ejemplo de un problema de decisión NP-hard (NP- duro).

Considere el problema del paro para un algoritmo determinístico. El problema del paro (the halting problem) es determinar para un arbitrario algoritmo determinístico A y una entrada I si el algoritmo A con la entrada I termina (o entra en un ciclo infinito). Este problema es indecidible. Por lo que no existe algoritmo (de ninguna complejidad) para resolver este problema. Por lo que el problema no es del tipo NP. Para poder mostrar satisfacibilidad α halting problem, simplemente se construye un algoritmo A cuya entrada es una formula proposicional X. Si X tiene n variables entonces A realiza los 2^n posibles asignaciones y verifica si X es satisfacible. Si lo es, entonces A se detiene. Si X no es satisfacible entonces A entra en un ciclo infinito. Si tenemos un algoritmo en tiempo polinomial para el halting problem entonces podemos resolver el problema de la satisfacibilidad en tiempo polinomial usando A y X como entrada del algoritmo para "the halting problem". De aquí que, el problema del paro es un problema NP-duro pero no está en NP.

Definición. Dos problemas L_1 y L_2 son polinomialmente equivalentes si y sólo si $L_1 \alpha L_2$ y $L_2 \alpha L_1$.
Para mostrar que un problema, L_2, es NP-duro es adecuado mostrar que $L_1 \alpha L_2$ donde L_1 es algún problema ya conocido como NP-duro. Ya que α es un operador transitivo, se muestra que si satisfacibilidad α L_1 y $L_1 \alpha L_2$ entonces satisfacibilidad α L_2. Para mostrar que un problema de decisión del tipo NP-duro es NP-completo, solo se debe de mostrar un algoritmo de tiempo polinomial no determinístico.

Los problemas de la clase NP-duro (y su subconjunto NP-completo) se encuentran en una gran variedad de disciplinas, algunos de ellos son:

NP-HARD Graph Problems:
-Clique Decision Problem (CDP)
-Node Cover Decision Problem
-Chromatic Number Decision Problem (CN)
-Directed Hamiltonian Cycle (DHC)
-Traveling Salesperson Decision Problem(TSP)
-AND/OR Graph Decision Problem (AOG)

NP-HARD Scheduling Problems:
-Scheduling Identical Processors
-Flow Shop Scheluling.

NP-HARD Code Generation Problems
-Code Generation With Common Subexpressions.
-Implementing Parallel Assignment Instructions.

La literatura muestra una gran cantidad de problemas NP-duros.

11.5.3 Cómo simplificar los problemas NP.
Una vez que se mostró el tiempo que se tarda en resolverse cualquier problema L del tipo NP-duro , nos podríamos inclinar por desechar la posibilidad de que L se pueda resolver en un tiempo polinomial determinístico. En este punto, sin embargo, uno puede realizar la siguiente pregunta: ¿Podría uno restringir el problema L a una subclase para poderse resolver en tiempo polinomial determinístico? Se puede observar que colocando las restricciones suficientes sobre un problema NP-duro (o definiendo una subclase lo suficientemente

representativa) se puede llegar a un problema que se pueda resolver en un tiempo polinomial, pero la solución tiene un relajamiento y no es el problema como tal.
Ya que es casi imposible que los problemas NP-duros se puedan resolver en tiempo polinomial, es importante determinar cuáles son las restricciones a relajar dentro de las cuales nosotros podamos resolver el problema en tiempo polinomial.

- Una posible máquina no determinística.

La **computación cuántica** es un paradigma de computación distinto al de la computación clásica. Se basa en el uso de qubits en lugar de bits, y da lugar a nuevas puertas lógicas que hacen posibles nuevos algoritmos. Una misma tarea puede tener diferente complejidad en computación clásica y en computación cuántica, lo que ha dado lugar a una gran expectación, ya que algunos problemas intratables pasan a ser tratables. Mientras un computador clásico equivale a una máquina de Turing, un computador cuántico equivale a una máquina de Turing cuántica.
En 1985, Deutsch presentó el diseño de la primera Máquina cuántica basada en una máquina de Turing. Con este fin enunció una nueva variante la tesis de Church-Turing dando lugar al denominado "principio de Church-Turing-Deutsch".
La estructura de una máquina de Turing cuántica es muy similar a la de una máquina de Turing clásica. Está compuesta por los tres elementos clásicos:

- una cinta de memoria infinita en donde cada elemento es un qubit,
- un procesador finito y
- un cabezal.

El procesador contiene el juego de instrucciones que se aplica sobre el elemento de la cinta señalado por el cabezal. El resultado dependerá del qubit de la cinta y del estado del procesador. El procesador ejecuta una instrucción por unidad de tiempo.
La cinta de memoria es similar a la de una máquina de Turing tradicional. La única diferencia es que cada elemento de la cinta de la máquina cuántica es un qubit. El alfabeto de esta nueva máquina está formado por el espacio de valores del qubit. La posición del cabezal se representa con una variable entera.

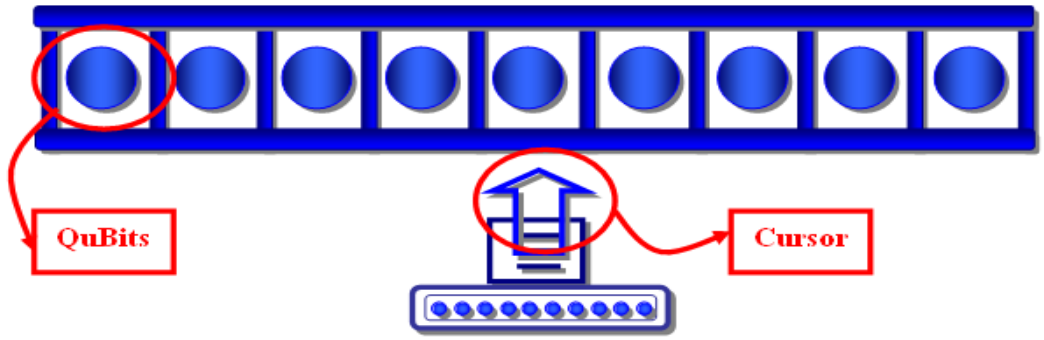


Fig. 7.2 Una máquina de Turing Cuantica.

11.6 Un poco de especulación.

Según ideas recientes de David Deutscth, es posible en principio construir una computadora cuántica para la que existen (clases de) problemas que no están en P, pero que podrían ser resueltos por dicho dispositivo en tiempo polinomial. No está claro todavía cómo podría construirse un dispositivo físico confiable que se comporte (confiablemente) como una

computadora cuántica – y además, la clase particular de problemas considerada hasta ahora es decididamente artificial-, pero subsiste la posibilidad teórica de que un dispositivo físico cuántico mejoraría una máquina de Turing.

¿Sería posible que un cerebro humano –que para nuestro estudio estoy considerando como un "dispositivo físico" sorprendentemente sutil, delicado en su diseño, así como complicado- estuviera sacando provecho de la teoría cuántica? ¿Comprendemos el modo en el que podrían ser aprovechados los efectos cuánticos para la solución de problemas y la formación de juicios? ¿Es concebible que tengamos que ir aún más allá de la teoría cuántica de hoy para hacer uso de esas ventajas? ¿En verdad los dispositivos físicos pueden mejorar la teoría de la complejidad para máquinas de Turing? ¿Qué sucede con la teoría de la computabilidad para dispositivos físicos reales?

Penrose deja una serie de interrogantes que permiten, en cierta forma, unir el procesamiento cerebral con el procesamiento de una computadora cuántica.

Trabajos citados

Abellanas, M., & Lodares, D. (1990). *Análisis de Algoritmos y Teoría de Grafos.* México: Macrobit-Ra-Ma.

Alfonseca Cubero, E., Alfonseca Moreno, M., & Moriyon, R. (2007). *Teoría de Autómatas y Lenguajes Formales.* México: Mac Graw Hill.

Booch, G. (1991). *Object Oriented Design with Applications.* California: The Benjamin/Cummings Publishing Company, Inc.

Cairó/Gardati. (2000). *Estructura de Datos.* México: Mc Graw hill.

Carlo Ghezzi, M. J. (1991). *Funamentals of Software Engineering.* New Jersey: Prentice Hall International.

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). *Algorithms.* New York: Mc Graw Hill.

Deheza, M. E. (2005). Importancia de la cohesión y acoplaimento en la Ingenierìa de Software. Texcoco, México: Universidad Francisco Ferreira y Arriola (Tesis de Licenciatura en Informática).

Deutsch, D. (s.f.). *Lectures on Cuantum Computation*. Recuperado el 20 de 09 de 2011, de http://www.quiprocone.org/Protected/DD_lectures.htm

Dewdney, A. K. (1989). *The Turing Omnibus. 61 Excursions in Computer Science.* New York: Computer Science Press.

Diccionario de la Real Academia Española. (s.f.). Recuperado el 30 de 09 de 2015, de http://lema.rae.es/drae/?val=algoritmo

Ellis horowitz, S. S. (1978). *Fundamentals of Computer Algorithms.* United States of America: Computer Science Press.

Garey, M. R., & Johnson, D. S. (1975). *Computer and intratability. A guide to the theory of NP Completeness.* U. S. A.: A series of Books in the Mathematical Science.

Goldshlager, L., & Lister, A. (1986). *Introducción Moderna a las Ciencias de la Computación con un enfoque algorítmico.* México: Prentice Hall.

Kewis, H. R., & Papadimitriou, C. H. (1989). *Elements of The Theory of Computation.* U. S. A.: Prentice Hall.

Levin, G. (2004). *Computaciónn y Programación Moderna, perspectiva integral de la Informática.* México: Addison Wesley.

Loomis, M. E. (2013). *Estructura de Datos y Organización de Archivos.* México: Prentice Hall.

Penrose, R. (1989). *La mente nueva del emperador. En torno a la cibernética, la mente y las leyes de la Física.* México: Fondo de Cultura Económica.

Singh, S. (1995). *El Enigma de Fermat.* México: Planeta.

ANEXO A Programa de Estudios.

Universidad Autónoma del Estado de México
UAEM

Secretaría de Docencia
Dirección de Estudios Profesionales

PROGRAMA DE ESTUDIOS POR COMPETENCIAS

PROGRAMACIÓN AVANZADA

I. IDENTIFICACIÓN DEL CURSO

<table>
<tr><td colspan="7">ORGANISMO ACADÉMICO: Facultad de Ingeniería</td></tr>
<tr><td colspan="3">PROGRAMA EDUCATIVO: Ingeniería en Computación</td><td colspan="4">ÁREA DE DOCENCIA: Programación e Ingeniería del Software</td></tr>
<tr><td colspan="2">APROBACIÓN POR LOS H.H. CONSEJOS ACADÉMICO Y DE GOBIERNO</td><td colspan="2">FECHA:</td><td colspan="2">PROGRAMA ELABORADO POR:
Dra. Lilia Ojeda Toche
Ing. Álvaro Árzate Trejo
Ing. Élfego Gutiérrez Ocampo</td><td>PROGRAMA REVISADO POR:
Integrantes de la Academia de Programación e Ingeniería de Software

Centro Universitario Valle de Chalco, Centro Universitario Zumpango</td></tr>
<tr><td colspan="4"></td><td colspan="2">FECHA DE ELABORACIÓN :
Junio 2007</td><td>FECHA DE REVISIÓN :
Mayo 2011</td></tr>
<tr><td>CLAVE</td><td>HORAS DE TEORÍA</td><td>HORAS DE PRÁCTICA</td><td>TOTAL DE HORAS</td><td>CRÉDITOS</td><td>TIPO DE UNIDAD DE APRENDIZAJE</td><td>NÚCLEO DE FORMACIÓN</td></tr>
<tr><td>L41053</td><td>3</td><td>3</td><td>6</td><td>9</td><td>Curso</td><td>Obligatoria</td></tr>
<tr><td colspan="2">PRERREQUISTOS:
Programación estructurada, Estructura de Datos</td><td colspan="3">UNIDAD DE APRENDIZAJE ANTECEDENTE:
Ninguna</td><td colspan="2">UNIDAD DE APRENDIZAJE CONSECUENTE:
Ninguna</td></tr>
<tr><td colspan="7">PROGRAMAS EDUCATIVOS O ESPACIOS ACADÉMICOS EN LOS QUE SE IMPARTE:
Licenciatura en Ingeniería en Computación (Facultad. de Ingeniería, Centros Universitarios: Atlacomulco, Ecatepec, Texcoco, Valle de Chalco, Valle de México, Valle de Teotihuacán, Zumpango)</td></tr>
</table>

II. PRESENTACIÓN

Una vez adquiridas las habilidades de programación básicas bajo el paradigma estructurado, el alumno debe conocer otros paradigmas como la programación modular y la programación recursiva.

Junto con estos paradigmas el alumno se adentra en cuestiones de algorítmica, con temas de análisis y diseño de algoritmos: funcionamiento y orden de complejidad de los métodos de ordenamiento y búsqueda, técnicas de diseño como algoritmos voraces, algoritmos divide y vencerás, programación dinámica, algoritmos vuelta atrás, algoritmos ramifica y poda.

Esta formación permitirá al futuro ingeniero enfrentarse a retos de programación de alta complejidad con la certeza de poder no sólo dar una solución a un problema dado sino de dar la solución óptima y ser capaz de evaluar que tan buena es la solución dada.

La estructura planteada consta de tres unidades de competencia. La primera estudia el paradigma de programación modular y programación recursiva. La segunda revisa diferentes algoritmos de ordenamiento y búsqueda. La tercera unidad de competencia se dedica a varias técnicas de diseño de algoritmos.

La evaluación debe considerar tanto la parte teórica como la práctica. La primera a través de exámenes escritos y la segunda por medio del planteamiento, codificación y ejecución de algoritmos aplicando los conocimientos teóricos obtenidos en el curso.

III. LINEAMIENTOS DE LA UNIDAD DE APRENDIZAJE

DEL DOCENTE	DEL DISCENTE
• Establecer las políticas del curso al inicio del mismo. • Respetar el horario del curso y la forma de evaluarlo. • Cumplir el temario y el número de horas asignadas al curso o justificar la ausencia por adelantado (asistencia a conferencias, etc.) • Asesorar y guiar el trabajo de las unidades de aprendizaje. • Retroalimentar el trabajo de los alumnos. • Fomentar la creatividad en los alumnos a través del desarrollo de proyectos.	• Contar con la asistencia establecida en el reglamento de Facultades: o 80% para examen ordinario o 60% para examen extraordinario o 30% para examen a titulo de suficiencia • Cumplir con las actividades encomendadas entregando con calidad en tiempo y forma los trabajos requeridos. • Participar activa y críticamente en el proceso de enseñanza-aprendizaje.

• Evaluar y Calificar a los alumnos. • Preparar el material didáctico para las clases y prácticas.	• Hacer uso adecuado de las instalaciones y equipo de cómputo. • Realizar las evaluaciones que se establezcan. • Mantener unas pautas de comportamiento socialmente aceptables cuando se encuentre en clases y laboratorio. • Cuando se requiera, entregar a tiempo y forma los trabajos requeridos.

IV. PROPÓSITO DE LA UNIDAD DE APRENDIZAJE

Servir de enlace entre el aprendizaje de los paradigmas estructurado y orientado a objetos, a través de la programación modular. Presentar al alumno técnicas de programación avanzada como la recursividad. Proporcionar las habilidades necesarias para evaluar la complejidad de un algoritmo de ordenamiento o de búsqueda, así como estrategias para resolver problemas de alta complejidad, mediante técnicas de diseño avanzadas.

V. COMPETENCIAS GENÉRICAS

Tal y como se establece en el apartado 4.2.1.1 Saberes del Plan Flexible 2004 por Competencias

- Analizar y diseñar sistemas de información
- Comunicarse con expertos de otras áreas
- Utilizar eficazmente los lenguajes de programación, dispositivos electrónicos y sistemas comerciales de vanguardia
- Responder eficazmente a nuevas situaciones informáticas
- Realizar investigación de tecnología de punta
- Analizar soluciones del entorno y problemas propios de ser tratados mediante sistemas computacionales
- Aplicar los conocimientos en la práctica
- Crear nuevas ideas para la solución de problemas
- Desarrollar la habilidad análisis y síntesis de información

Algunas de estas competencias se adquieren en conjunto al estudiar el resto de unidades de aprendizaje bajo el área de competencia de Programación e Ingeniería del Software.

VI. ÁMBITOS DE DESEMPEÑO

- Empresas públicas y privadas
- Investigación de nuevas soluciones computacionales
- Docencia a cualquier nivel de aprendizaje escolarizado
- Desarrollo de proyectos.
- Análisis, diseño, implementación y mantenimiento de sistemas computacionales

VII. ESCENARIOS DE APRENDIZAJE

- Salón de Clases
- Sala de cómputo

VIII. ESTRUCTURA DE LA UNIDAD DE APRENDIZAJE

1. Paradigmas de programación modular y recursiva.
2. Análisis de algoritmos de ordenamiento y búsqueda
3. Diseño de algoritmos empleando diversas técnicas.

IX. DESARROLLO DE LA UNIDAD DE APRENDIZAJE

UNIDAD DE COMPETENCIA I	ELEMENTOS DE COMPETENCIA		
	CONOCIMIENTOS	HABILIDADES	ACTITUDES/ VALORES
Paradigmas de programación modular y recursiva.	- Programación modular - Programación Recursiva	Desarrollar algoritmos bajo los paradigmas de: Programación modular	Receptiva Analítica Responsabilidad para cumplir con las tareas asignadas

		Programación recursiva.	Tolerancia y participación Desarrollar la capacidad analítica ante nuevos problemas Respetar al docente y a los compañeros mediante un comportamiento socialmente aceptable
ESTRATEGIAS DIDÁCTICAS: – Presentaciones acompañadas de apuntes preparados por el profesor. – Revisión y análisis de material bibliográfico – Solución de ejercicios – Desarrollo de prácticas de programación en laboratorio		**RECURSOS REQUERIDOS** Pizarrón, Libro de texto y apuntes del docente Laboratorio de prácticas con un lenguaje de programación modular Videoproyector	**TIEMPO DESTINADO** 12 horas teóricas 12 horas práctica
CRITERIOS DE DESEMPEÑO I		**EVIDENCIAS**	
		DESEMPEÑO	**PRODUCTOS**
Resolución de problemas Práctica de laboratorio		Ejercicios teóricos Programas	Programación Modular y programación recursiva. Programación Modular y programación recursiva. Lenguaje de programación

UNIDAD DE COMPETENCIA II	ELEMENTOS DE COMPETENCIA		
	CONOCIMIENTOS	HABILIDADES	ACTITUDES/ VALORES
Análisis de algoritmos de ordenamiento y búsqueda	- Orden de complejidad de un algoritmo - Método de la burbuja - Método de Selección - Método de Inserción - Método de Ordenamiento rápido - Método de mezcla - Búsqueda Secuencial - Búsqueda binaria	• Conocer el funcionamiento de un método de ordenamiento y búsqueda • Obtener el orden de complejidad de un algoritmo de ordenamiento y búsqueda.	Receptiva Analítica Responsabilidad para cumplir con las tareas asignadas Tolerancia y participación Desarrollar la capacidad analítica ante nuevos problemas Respetar al docente y a los compañeros mediante un comportamiento socialmente aceptable
ESTRATEGIAS DIDÁCTICAS: – Presentaciones acompañadas de apuntes preparados por el profesor. – Revisión y análisis de material bibliográfico – Solución de ejercicios – Desarrollo de prácticas de programación en laboratorio		**RECURSOS REQUERIDOS** Pizarrón, Libro de texto y apuntes del docente Laboratorio de prácticas con un leguaje de programación Videoproyector	**TIEMPO DESTINADO** 18 horas teóricas 18 horas práctica
CRITERIOS DE DESEMPEÑO II		**EVIDENCIAS**	
		DESEMPEÑO	**PRODUCTOS**
Resolución de problemas		Ejercicios teóricos	Funcionamiento y órdenes de complejidad de los métodos de

		ordenamiento y búsqueda.
Práctica de laboratorio	Programas	Funcionamiento y órdenes de complejidad de los métodos de ordenamiento y búsqueda. Lenguaje de programación.

UNIDAD DE COMPETENCIA III	ELEMENTOS DE COMPETENCIA		
	CONOCIMIENTOS	HABILIDADES	ACTITUDES/ VALORES
Técnicas de Diseño de algoritmos.	Estrategias de diseño de algoritmos: - Algoritmos voraces - Divide y vencerás - Programación dinámica - Vuelta atrás - Ramifica y poda	Desarrollar algoritmos empleando diferentes técnicas de diseño dependiendo de la naturaleza del problema.	Receptiva Analítica Responsabilidad para cumplir con las tareas asignadas Tolerancia y participación Desarrollar la capacidad analítica ante nuevos problemas Respetar al docente y a los compañeros mediante un comportamiento socialmente aceptable
ESTRATEGIAS DIDÁCTICAS: – Presentaciones acompañadas de apuntes preparados por el profesor. – Revisión y análisis de material bibliográfico		**RECURSOS REQUERIDOS** Pizarrón, Libro de texto y apuntes del	**TIEMPO DESTINADO** 18 horas teóricas 18 horas práctica

– Solución de ejercicios – Desarrollo de prácticas de programación en laboratorio	docente Laboratorio de prácticas con un lenguaje de programación modular Videoproyector	
CRITERIOS DE DESEMPEÑO III	**EVIDENCIAS**	
	DESEMPEÑO	**PRODUCTOS**
Resolución de problemas	Desarrollo de aplicaciones utilizando las diferentes estrategias de diseño de algoritmos	Estrategias de diseño de Algoritmos

X. EVALUACIÓN Y ACREDITACIÓN

Para tener derecho a examen ordinario requiere al menos un 80% de asistencia.
Para tener derecho a presentar el examen extraordinario se requiere al menos un 60 % de asistencia.
Para tener derecho a presentar el examen a título de suficiencia se requiere al menos un 30% de asistencia.

Se realizarán dos exámenes parciales con valor del 60% (30% cada uno)
y prácticas de programación con un valor de 40%

Si la calificación es mayor o igual a 80% el alumno exenta.

La calificación final ordinaria estará compuesta por 60 % de examen escrito final y un 40 % de un proyecto acumulativo práctico

La calificación final extraordinaria estará compuesta por 70 % del examen escrito final correspondiente y un 30 % de un proyecto final práctico

La calificación final a título de suficiencia estará compuesta al 100 % del examen escrito final respectivo.

Universidad Autónoma del Estado de México
UAEM

Secretaría de Docencia
Dirección de Estudios Profesionales

XI. REFERENCIAS

BIBLIOGRAFÍA BÁSICA:

- ✓ Base, S.; Van Gelder, A.(2002). "Algoritmos Computacionales: Introducción al análisis y diseño" Ed. Addison Wesley
- ✓ Bovet, D. P.; Crescenci, P. (2006). "*Introduction to the theory of complexity*" Ed. Creative Commons
- ✓ Brassard, G.; Bratley, P. (1997). "Fundamentos de Algoritmia", Ed. Prentice Hall.
- ✓ Cairó, Osvaldo y Guardati, Silvia. (2006). *Estructuras de datos* (3a. Edición). McGraw-Hill.
- ✓ Lee R. Teng. S. Chang R. y Tsai Y. (2007). *Introducción al diseño de algoritmos*. McGraw- Hill
- ✓ Levitin Anany (2002). *Introduction to Design and Analysis of Algoritms.* Addison Wesley.

BIBLIOGRAFÍA COMPLEMENTARIA:

- ✓ Drozdeck, Adam. (2007). *Estructuras de datos y algoritmos en Java* (2ª Edición). Thomson.
- ✓ Joyanes, Luis. M. Fernández, L: Sánchez, I. Zahonero. (2005). *Estructuras de datos en C.* McGraw-Hill. Schaum.
- ✓ Koffman, Elliot y Wolfgang, Paul. (2008). *Estructura de datos con C++.* Objetos, abstracciones y diseño. McGraw-Hill.