UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

DIVISIÓN DEL SISTEMA UNIVERSIDAD ABIERTA

TUTORIAL

PARA LA ASIGNATURA

ANÁLISIS, DISEÑO E IMPLANTACIÓN DE ALGORITMOS

Fondo Editorial F C A

2003

DIRECTOR

C. P. C. y Mtro. Arturo Díaz Alonso

SECRETARIO GENERAL

L. A. E. Félix Patiño Gómez

JEFE DE LA DIVISIÓN-SUA

L. A. y Mtra. Gabriela Montero Montiel

COORDINACIÓN DE OPERACIÓN ACADÉMICA

L. A. Ramón Arcos González

COORDINACIÓN DE PROYECTOS EDUCATIVOS

L. E. Arturo Morales Castro

COORDINACIÓN DE MATERIAL DIDÁCTICO

L. A. Francisco Hernández Mendoza

COORDINACIÓN DE ADMINISTRACIÓN ESCOLAR

L. C. Virginia Hidalgo Vaca

COORDINACIÓN ADMINISTRATIVA

L. C. Danelia C. Usó Nava

TUTORIAL

PARA LA ASIGNATURA

ANÁLISIS, DISEÑO E IMPLANTACIÓN DE ALGORITMOS

2003

Fondo Editorial
F C A

Colaboradores

Diseño y coordinación general
L. A. Francisco Hernández Mendoza

Coordinación operativa
L. A. Francisco Hernández Mendoza

Asesoría pedagógica
Sandra Rocha

Corrección de estilo
Gregorio Martínez Moctezuma

Edición
L. C. Aline Gómez Angel

PRÓLOGO

En una labor editorial más de la Facultad de Contaduría y Administración, los Tutoriales del Sistema Universidad Abierta, representan un esfuerzo dirigido principalmente a ayudar a los estudiantes de este Sistema a que avancen en el logro de sus objetivos de aprendizaje.

Al poner estos Tutoriales a disposición tanto de alumnos como de asesores, esperamos que les sirvan como punto de referencia; a los asesores para que dispongan de materiales que les permitan orientar de mejor manera, y con mayor sencillez, a sus estudiantes y a éstos para que dispongan de elementos que les permitan organizar su programa de trabajo, para que le facilite comprender cuáles son los objetivos que se persiguen en cada matera y para que se sirvan de los apoyos educativos que contienen.

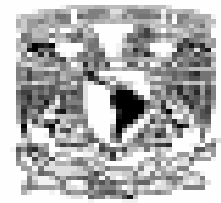
Por lo anterior y después de haberlos utilizado en un periodo experimental para probar su utilidad y para evaluarlos en un ambiente real, los ponemos ahora a disposición de nuestra comunidad, esperando que cumplan con sus propósitos.

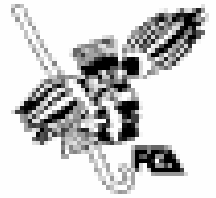
ATENTAMENTE

Cd. Universitaria D.F., mayo de 2003.

C.P.C. Y MAESTRO ARTURO DÍAZ ALONSO,

DIRECTOR.





Primera edición mayo de 2003
DR © 2001 Universidad Nacional Autónoma de México
Facultad de Contaduría y Administración
Fondo editorial FCA
Circuito Exterior de Cd. Universitaria, México D.F., 04510
Delegación Coyoacán

Impreso y hecho en México
ISBN

Contenido

Introducción 7

Características de la asignatura 11

Objetivos generales de la asignatura 11

Temario oficial (68 horas sugeridas) 11

Temario detallado 11

Unidad 1. Autómatas y lenguajes formales 15

Unidad 2. Computabilidad 25

Unidad 3. Funciones recursivas 33

Unidad 4. Diseño de algoritmos para la solución de problemas 37

Unidad 5. Evaluación de algoritmos 47

Unidad 6. Estrategias de programación para la implantación de algoritmos 52

Bibliografía 64

Apéndice. Elaboración de un mapa conceptual 65

Introducción

El principal propósito de este tutorial es orientar a los estudiantes que cursan sus estudios en el sistema abierto, que se caracteriza, entre otras cosas, porque ellos son los principales responsables de su propio aprendizaje.

Como en este sistema cada alumno debe estudiar por su cuenta, en los tiempos y lugares que más le convengan, se vuelve necesaria un material que le ayude a lograr los objetivos de aprendizaje y que le facilite el acceso a los materiales didácticos (libros, publicaciones, audiovisuales, etcétera) que requiere. Por estas razones, se han estructurado estos tutoriales básicamente en cuatro grandes partes:

1. Información general de la asignatura
2. Panorama de la asignatura
3. Desarrollo de cada una de las unidades
4. Bibliografía

A su vez, estas cuatro partes contienen las siguientes secciones:

La información general de la asignatura que incluye: portada, características oficiales de la materia, índice de contenido del tutorial y los nombres de las personas que han participado en la elaboración del material.

El panorama de la asignatura contiene el objetivo general del curso, el temario oficial (que incluye solamente el título de cada unidad), y el temario detallado de todas las unidades

Por su parte, el desarrollo de cada unidad que está estructurado en los siguientes apartados:

1. Temario detallado de la unidad que es, simplemente, la parte del temario detallado global que corresponde a cada unidad.
2. Desarrollo de cada uno de los puntos de cada unidad.
3. Bibliografía general sugerida. Como no se pretende imponer ninguna bibliografía a los profesores, es importante observar que se trata de una sugerencia, ya que cada

profesor está en entera libertad de sugerir a sus alumnos la bibliografía que le parezca más conveniente.

Esperamos que este tutorial cumpla con su cometido y, en todo caso, deseamos invitar a los lectores, tanto profesores como alumnos, a que nos hagan llegar todo comentario o sugerencia que permita mejorarla.

A t e n t a m e n t e

L. A. y Mtra. Gabriela Montero Montiel

Jefe de la División del Sistema Universidad Abierta

Mayo de 2003.

Características de la asignatura

Análisis, diseño e implantación de algoritmos	Clave: 1132
Plan: 98.	Créditos: 8
Licenciatura: Informática	Semestre: 1º
Área: Informática	Horas de asesoría: 2
Requisitos: Ninguno	Horas por semana: 4
Tipo de asignatura: Obligatoria (x)	Optativa ( )

Objetivos generales de la asignatura

El alumno conocerá los fundamentos básicos de la computación y utilizará las estrategias algorítmicas para la solución de problemas.

Temario oficial (68 horas sugeridas)

1. Autómatas y lenguajes formales (12 horas)
2. Computabilidad (12 horas)
3. Funciones recursivas (6 horas)
4. Diseño de algoritmos para la solución de problemas (10 horas)
5. Evaluación de algoritmos (6 horas)
6. Estrategias de programación para la implantación de algoritmos (16 horas)
7. Evaluación (6 horas)

Temario detallado

1. Autómatas y lenguajes formales
   - 1.1 Definición de alfabeto
   - 1.2 Definición de frase
   - 1.3 Definición de cadena vacía
   - 1.4 Definición de lenguaje
   - 1.5 Gramáticas formales
   - 1.6 Definición del lenguaje formal

- 1.7 Jerarquización de gramáticas
  - 1.7.1 Gramáticas sensibles al contexto
  - 1.7.2 Gramáticas independientes del contexto
  - 1.7.3 Gramáticas regulares
- 1.8 Propiedades de indecibilidad

2. Computabilidad
   - 2.1 Representación de un fenómeno descrito
   - 2.2 Modelo
   - 2.3 El problema de la decisión
     - 2.3.1 Concepto de algoritmo: máquina de Turing
   - 2.4 Máquina de Turing como función
   - 2.5 Procesos computables
   - 2.6 Procesos indecidibles
   - 2.7 Máquina universal
   - 2.8 Complejidad
     - 2.8.1 Polinomial
     - 2.8.2 Exponencial
     - 2.8.3 Notaciones
3. Funciones recursivas.
   - 3.1 Introducción a la inducción
   - 3.2 Definición de funciones recursivas
   - 3.3 Cálculo de la complejidad de una función recursiva
4. Diseño de algoritmos para la solución de problemas
   - 4.1 Niveles de abstracción para la construcción de algoritmos
   - 4.2 Estructuras básicas en un algoritmo
     - 4.2.1 Ciclos
     - 4.2.2 Contadores
     - 4.2.3 Acumuladores
     - 4.2.4 Condicionales
     - 4.2.5 Interruptores

- 4.3 Rutinas recursivas
- 4.4 Refinamiento progresivo
- 4.5 Procesamiento regresivo

5. Evaluación de algoritmos
6. Estrategias de programación para la implantación de algoritmos
   - 6.1 Programación imperativa
   - 6.2 Programación lógica
   - 6.3 Programación funcional
   - 6.4 Programación orientada a objetos

Unidad 1. Autómatas y lenguajes formales

Temario detallado

1. Autómatas y lenguajes formales
    - 1.1 Definición de alfabeto
    - 1.2 Definición de frase
    - 1.3 Definición de cadena vacía
    - 1.4 Definición de lenguaje
    - 1.5 Gramáticas formales
    - 1.6 Definición del lenguaje formal
    - 1.7 Jerarquización de gramáticas
        - 1.7.1 Gramáticas sensibles al contexto
        - 1.7.2 Gramáticas independientes del contexto
        - 1.7.3 Gramáticas regulares
    - 1.8 Propiedades de indecibilidad

1. Autómatas y lenguajes formales

Un autómata es un modelo computacional consistente en un conjunto de estados bien definidos, un estado inicial, un alfabeto de entrada y una función de transición. Este concepto es equivalente a otros como autómata finito o máquina de estados finitos.

En un autómata, un estado es la representación de su condición en un instante dado. El autómata comienza en el estado inicial con un conjunto de símbolos; su paso de un estado a otro se efectúa a través de la función de transición, la cual, partiendo del estado actual y un conjunto de símbolos de entrada, lo lleva al nuevo estado correspondiente.

Históricamente, los autómatas han existido desde

la Antigüedad, pero en el siglo XVII, cuando en Europa existía gran pasión por la técnica, se perfeccionaron las cajas de música compuestas por cilindros con púas, que fueron inspiradas por los pájaros autómatas que había en Bizancio y que podían cantar y silbar.

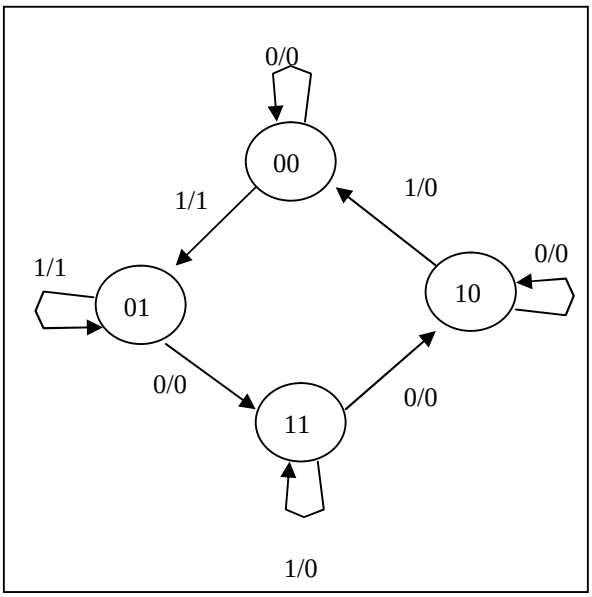
Así, a principios del siglo XVIII, el ebanista Roentgen y Kintzling mostraron a Luis XVI un autómata con figura humana llamado "La tañedora de salterio". Por su parte, la aristocracia se apasionaba por los muñecos mecánicos de encaje, los cuadros con movimiento y otros personajes.

Los inventores más célebres son Pierre Jacquet Droz, autor de "El dibujante" y "Los músicos", y M. de Vaucanson, autor de "El pato digeridor", un personaje que aleteaba, parloteaba, tragaba grano y evacuaba los residuos. Este último autor quiso pasar de lo banal a lo útil y sus trabajos culminaron en el telar de Jacquart y la máquina de Falcon dirigida por tarjetas perforadas.

Un ejemplo de lo anterior se puede ver en el siguiente *videoclip*, que muestra el movimiento de un autómata pintor del siglo XVII.

El autómata más conocido en el mundo es el denominado “Máquina de Turing”, elaborado por el matemático inglés Alan Turing.

En términos estrictos, actualmente se puede decir que un termostato es un autómata, puesto que regula la potencia de calefacción de un aparato (salida) en función de la temperatura ambiente (dato de entrada), pasando de un estado térmico a otro. Un ejemplo más de autómata en la vida cotidiana es un elevador, ya que es capaz de memorizar las diferentes llamadas de cada piso y optimizar sus ascensos y descensos.



Técnicamente existen diferentes herramientas para poder definir el comportamiento de un autómata, entre las cuales se encuentra el diagrama de

estado. En él se pueden visualizar los estados como círculos que en su interior registran su significado. Existen flechas que representan la transición entre estados y la notación de Entrada/Salida que provoca la transición entre estados. En el ejemplo de al lado se muestran cuatro diferentes estados de un autómata y se define lo siguiente: partiendo del estado "00", si se recibe una entrada "0", la salida es "0" y el autómata conserva el estado actual, pero si la entrada es "1", la salida será "1" y el autómata pasa al estado "01". Este comportamiento es homogéneo para todos los estados del autómata. Vale la pena resaltar que el autómata que se muestra aquí tiene un alfabeto binario (0 y 1).

Otra herramienta de representación del comportamiento de los autómatas es la tabla de estado que consiste de cuatro partes: descripción del estado actual, descripción de la entrada, descripción del estado siguiente, descripción de las salidas. La siguiente tabla es la correspondiente al diagrama que se presentó en la figura anterior.

Estado actual		**Entrada**	**Estado siguiente**		**Salida**
A	**B**	**x**	**A**	**B**	**y**
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	0	0	0
1	1	0	1	0	0
1	1	1	1	1	0

En la tabla se puede notar que el autómata tiene dos elementos que definen su estado, A y B, así como la reafirmación de su alfabeto binario. Además, podemos deducir la función de salida del autómata, la cual está definida por la multiplicación lógica de la negación del estado de A por la entrada x:

$$y = A' x$$

1.1. Definición de alfabeto

Un alfabeto se puede definir como el conjunto de todos los símbolos válidos o posibles para una aplicación. Por tanto, en el campo de los autómatas, un alfabeto está formado por todos los caracteres que utiliza para definir sus entradas, salidas y estados.

En algunos casos, el alfabeto puede ser infinito o tan simple como el alfabeto binario, que se utilizó en el ejemplo del punto anterior, donde sólo se usan los símbolos 1 y 0 para representar cualquier expresión de entrada, salida y estado.

1.2. Definición de frase

Una frase es la asociación de un conjunto de símbolos definidos en un alfabeto (cadena) que tiene la propiedad de tener sentido, significado y lógica.

Las frases parten del establecimiento de un vocabulario que define las palabras válidas del lenguaje sobre la base del alfabeto definido. Una frase válida es aquella que cumple con las reglas que define la gramática establecida.

1.3. Definición de cadena vacía

Se dice que una cadena es vacía cuando la longitud del conjunto de caracteres que utiliza es igual a cero, es decir, es una cadena que no tiene caracteres asociados.

Este tipo de cadenas no siempre implican el no cambio de estado en un autómata, ya que en la función de transición puede existir una definición de cambio de estado asociada a la entrada de una cadena vacía.

1.4. Definición de lenguaje

Se puede definir un lenguaje como un conjunto de cadenas que obedecen a un alfabeto fijado.

Un lenguaje, entendido como un conjunto de entradas, puede o no ser resuelto por un algoritmo.

1.5. Gramáticas formales

Una gramática es una colección estructurada de palabras y frases ligadas por reglas que definen el conjunto de cadenas de caracteres que representan los comandos completos que pueden ser reconocidos por un motor de discurso.

Las gramáticas definen formalmente el conjunto de frases válidas que pueden ser reconocidas por un motor de discurso.

Una forma de representar las gramáticas es a través de la forma Bakus-Naur (BNF), la cual es usada para describir la sintaxis de un lenguaje dado, así como su notación.

La función de una gramática es definir y enumerar las palabras y frases válidas de un lenguaje. La forma general definida por BNF es denominada regla de producción y se puede representar como:

<regla> = sentencias y frases. *****

***Nota**: En todos los ejemplos sintácticos y de código usados en este tutorial se omitirán los acentos.

Las partes de la forma general BNF se definen como sigue:

- El "lado izquierdo" o regla es el identificador único de las reglas definidas para el lenguaje. Puede ser cualquier palabra, con la condición de estar encerrada entre los símbolos <>. Este elemento es obligatorio en la forma BNF.
- El operador de asignación **=** es un elemento obligatorio.
- El "lado derecho", o sentencias y frases, define todas las posibilidades válidas en la gramática definida.

- El delimitador de fin de instrucción (punto) es un elemento obligatorio.

Un ejemplo de una gramática que define las opciones de un menú asociado a una aplicación de ventanas puede ser:

<raiz> = ARCHIVO
| EDICION
| OPCIONES
| AYUDA.

En este ejemplo podemos encontrar claramente el concepto de símbolos terminales y símbolos no-terminales. Un símbolo terminal es una palabra del vocabulario definido en un lenguaje, por ejemplo, "ARCHIVO", "EDICION", etc. Por otra parte, un símbolo no-terminal se puede definir como una regla de producción de la gramática, por ejemplo:

<raiz> = <opcion>.
<opcion> = ARCHIVO
| EDICION
| OPCIONES
| AYUDA.

Otro ejemplo más complejo que involucra el uso de frases es el siguiente:

<raiz> = Hola mundo | Hola todos.

En los ejemplos anteriores se usó el símbolo **|** (OR), el cual denota opciones de selección mutuamente excluyentes, lo que quiere decir que sólo se puede elegir una opción entre ARCHIVO, EDICION, OPCIONES y AYUDA, en el primer ejemplo, así como "Hola mundo" y "Hola todos", en el segundo.

Un ejemplo real aplicado a una frase simple de uso común como "Me puede mostrar su licencia", con la opción de anteponer el título Señorita, Señor o Señora, se puede estructurar de la manera siguiente en una gramática BNF:

<peticion> = <comando> | <titulo> <comando> .
<titulo> = Señor | Señora | Señorita.
<comando> = Me puede mostrar su licencia.

Hasta este momento sólo habíamos definido reglas de producción que hacían referencia a símbolos terminales, sin embargo, en el ejemplo anterior se puede ver que la regla <peticion> está formada sólo por símbolos no-terminales.

Otra propiedad más que nos permite visualizar el ejemplo anterior es la definición de frases y palabras opcionales, es decir, si analizamos la regla de producción <peticion>, podremos detectar que una petición válida puede prescindir del uso del símbolo <titulo>, mientras que el símbolo <comando> se presenta en todas las posibilidades válidas de <peticion>.

Una sintaxis que se puede utilizar para simplificar el significado de <peticion> es usando el operador "?":

<peticion> = <titulo>? <comando>.

Con la sintaxis anterior se define que el símbolo <titulo> es opcional, o sea que puede ser omitido sin que la validez de la <peticion> se pierda.

1.6. Definición del lenguaje formal

De lo anterior podemos decir que un lenguaje formal está constituido por un alfabeto, un vocabulario y un conjunto de reglas de producción definidas por gramáticas.

Las frases válidas de un lenguaje formal son aquellas que se crean con los símbolos y palabras definidas tanto en el alfabeto como en el vocabulario del lenguaje y que cumplen con las reglas de producción definidas en las gramáticas.

1.7. Jerarquización de gramáticas

Las gramáticas pueden ser de distintos tipos, de acuerdo con las características que rigen la formulación de reglas de producción válidas, todos los cuales parten de las gramáticas arbitrarias que son aquellas que consideran la existencia infinita de cadenas formadas por los símbolos del lenguaje. Los principales tipos derivados son:

1.7.1. Gramáticas sensibles al contexto

Este tipo de gramáticas tiene la característica de que el lado derecho de la regla de producción siempre debe ser igual o mayor que el lado izquierdo.

1.7.2. Gramáticas independientes del contexto

Una gramática independiente del contexto es aquella que cumple con las propiedades de la gramática sensible al contexto y que, además, tiene la característica de que el lado izquierdo de la regla de producción sólo debe tener un elemento y éste no puede ser un elemento terminal.

1.7.3. Gramáticas regulares

Una gramática es regular cuando cumple con las características de la gramática independiente del contexto y, además, se restringe a través de las reglas de producción para generar sólo reglas de los dos tipos anteriores.

1.8. Propiedades de indecidibilidad

Se dice que un lenguaje es indecidible si sus miembros no pueden ser identificados por un algoritmo que restrinja todas las entradas en un número de pasos finito. Otra de sus propiedades es que no puede ser reconocido como una entrada válida en una máquina de Turing.

Asociados a este tipo de lenguaje existen, de la misma manera, problemas indecidibles que son aquellos que no pueden ser resueltos, con todas sus variantes, por un algoritmo.

En contraposición con el lenguaje indecidible y los problemas asociados a este tipo de lenguajes existen los problemas decidibles, que están relacionados con lenguajes del mismo tipo y que tienen las características opuestas.

Este tipo de lenguajes se conoce, también, como lenguajes recursivos o lenguajes totalmente decidibles.

Direcciones electrónicas

http://www.cjn.or.jp/automata/index.html

http://www.ijp.si/vega/htmldoc/usages/AUTOMAT.HTM#Automaton

http://www.automaton.com/bdyProfile.asp

http://www.dekware.com/automaton/autousersman.html

http://www.cna.org/isaac/Glossb.htm

http://hissa.nist.gov/dads

Bibliografía de la Unidad

Unidad 2. Computabilidad

Temario detallado

2. Computabilidad
    - 2.1 Representación de un fenómeno descrito
    - 2.2 Modelo
    - 2.3 El problema de la decisión
        - 2.3.1 Concepto de algoritmo: máquina de Turing
    - 2.4 Máquina de Turing como función
    - 2.5 Procesos computables
    - 2.6 Procesos indecidibles
    - 2.7 Máquina universal
    - 2.8 Complejidad
        - 2.8.1 Polinomial
        - 2.8.2 Exponencial
        - 2.8.3 Notaciones

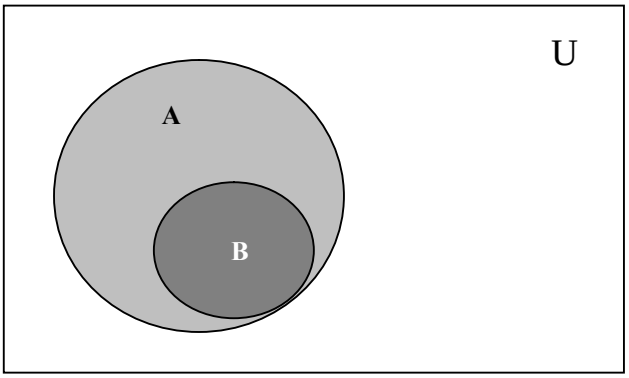
2. Computabilidad

Una de las funciones principales de la computación ha sido la resolución de problemas a través del uso de la tecnología. Sin embargo, esto no ha logrado realizarse en la totalidad de los casos debido a una propiedad particular que se ha asociado a éstos: la computabilidad.

La computabilidad es la propiedad que tienen ciertos problemas de poder resolverse a través de un algoritmo (entendido éste como una serie finita de pasos secuenciales) o de una Máquina de Turing.

Atendiendo a esta propiedad, los problemas pueden dividirse en tres categorías: irresolubles, solucionables y computables; estos últimos son un

subconjunto de los segundos. Representado en un diagrama de Venn, esto puede verse de la manera siguiente:



Donde *U* representa el universo de problemas existentes, *A* el conjunto de problemas con una solución conocida y *B* el conjunto de problemas computables.

2.1. Representación de un fenómeno descrito

Todos los fenómenos de la naturaleza poseen características intrínsecas que los particularizan y permiten diferenciar unos de otros, y la percepción que se tenga de éstas posibilita tanto su abstracción como su representación a partir de ciertas herramientas.

La percepción que se tiene de un fenómeno implica conocimiento; cuando se logra su representación, se dice que dicho conocimiento se convierte en un conocimiento transmisible. Esta representación puede realizarse utilizando diferentes técnicas de abstracción, desde una pintura hasta una función matemática; sin embargo, la interpretación que puede darse a los diferentes tipos de representación varía de acuerdo a dos elementos: la regulación de la técnica utilizada y el conocimiento del receptor.

De esta manera, un receptor, con un cierto nivel de conocimientos acerca de arte, podrá tener una interpretación distinta a la de otra persona con el mismo nivel cuando se observa una pintura; pero un receptor con un nivel de conocimientos matemáticos análogo al nivel de otro receptor siempre dará la misma interpretación a una expresión matemática. Esto se debe a que en el primer caso intervienen

factores personales de interpretación, que hacen válidas las diferencias, mientras que en el segundo caso se tiene un cúmulo de reglas que no permiten variedad de interpretaciones sobre una misma expresión. En esta unidad nos enfocaremos en la representación de fenómenos del segundo caso.

2.2. Modelo

La representación de los fenómenos se hace a través de modelos, los cuales son abstracciones que destacan las características más sobresalientes de ellos, o bien, aquellas características que sirvan al objetivo para el cual se realiza el modelo.

Los problemas computables pueden representarse a través de lenguaje matemático o con la definición de algoritmos. Es importante mencionar que todo problema que se califique como computable debe poder resolverse con una Máquina de Turing.

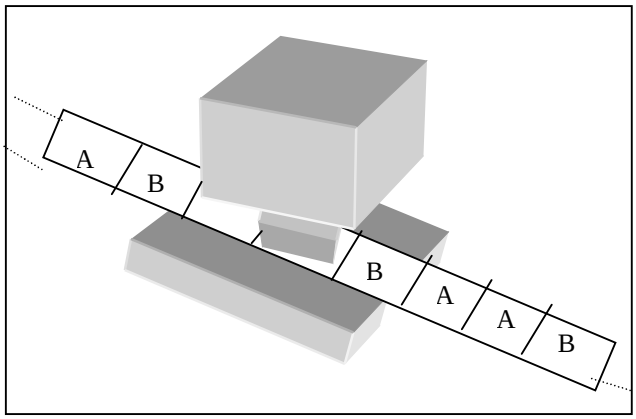
2.3. El problema de la decisión

Un problema de decisión es aquél cuya respuesta puede *mapearse* al conjunto de valores {0,1}, esto es, que tiene sólo dos posibles soluciones: sí o no. La representación de este tipo de problemas se puede hacer a través de una función cuyo dominio sea el conjunto citado.

Se dice que un problema es decidible cuando puede resolverse en un número finito de pasos por un algoritmo que recibe todas las entradas posibles para dicho problema. El lenguaje con el que se implementa dicho algoritmo se denomina lenguaje decidible o recursivo.

Por el contrario, un problema no decidible es aquel que no puede resolverse por un algoritmo en todos sus casos. Asimismo, su lenguaje asociado no puede ser reconocido por una Máquina de Turing.

2.3.1. Concepto de algoritmo: Máquina de Turing



Un algoritmo es un conjunto de pasos secuenciales para conseguir un fin específico. Este concepto fue implementado en 1936 por Alan Turing, matemático inglés, en la llamada Máquina de Turing.

La Máquina de Turing está formada por tres elementos: una cinta, una cabeza de lectura-escritura y un programa. La cinta tiene la propiedad de ser infinita y estar dividida en cuadrados del mismo tamaño, los cuales pueden almacenar cualquier símbolo o estar en blanco. La cinta puede interpretarse como el dispositivo de almacenamiento de la Máquina de Turing.

La cabeza de lectura-escritura es el dispositivo que lee y escribe en la cinta infinita. Tiene la propiedad de poder actuar en un cuadro y ejecutar sólo una operación a la vez. También tiene un número finito de estados, que cambian de acuerdo a la entrada y a las instrucciones definidas en el programa que lo controla.

El último elemento, el programa, es un conjunto secuencial de instrucciones que controla los movimientos de la cabeza de lectura-escritura indicándole hacia dónde debe moverse y si debe escribir o leer en la celda donde se encuentre.

Actualmente, la Máquina de Turing es una de las principales abstracciones utilizadas en la teoría moderna de la computación, ya que auxilia en la definición de lo que una computadora puede o no puede hacer.

2.4. Máquina de Turing como función

Si a partir del concepto definido en el punto anterior queremos definir la Máquina de Turing en términos de función, podríamos decir que es una función cuyo dominio se encuentra en la cinta infinita y es en esta misma donde se plasma su codominio, esto es, todos los posibles valores de entrada se encuentran en la cinta y todos los resultados de su operación se plasman también en ella.

La Máquina de Turing es el antecedente más remoto de un autómata y, al igual que éste, puede definirse con varias herramientas: diagrama de estado, tabla de estado y función.

2.5. Procesos computables

Los procesos computables son un derivado de los problemas computables y tienen las mismas características: un proceso computable es aquel que puede implementarse en un algoritmo o Máquina de Turing y definirse en un lenguaje decidible. Un proceso computable puede implementarse como el programa de la Máquina de Turing.

2.6. Procesos indecidibles

Al igual que los problemas no computables, el problema indecidible es aquel que no puede implementarse con una Máquina de Turing por no tener solución para todas sus posibles entradas. El lenguaje en que se especifica es un lenguaje indecidible que no puede ser interpretado por una Máquina de Turing.

2.7. Máquina universal

Una Máquina universal es una Máquina de Turing que es capaz de simular el funcionamiento de cualquier otra Máquina de Turing por codificación posterior.

2.8. Complejidad

La complejidad es la cantidad mínima de recursos necesarios, como memoria y tiempo de procesador, para resolver un problema o algoritmo.

Existen diferentes métodos para calcular la complejidad asociada a un algoritmo; de hecho, los algoritmos pueden clasificarse de acuerdo con sus límites de utilización de recursos en clases de complejidad. Las principales clases de complejidad se llaman clases canónicas por englobar en ellas la mayor parte de los problemas computacionales importantes; estas clases se definen a continuación.

Lineal. Es la clase de complejidad más simple que asocia algoritmos cuyo límite de complejidad está definido por una función lineal para un problema de tamaño *n* (número de elementos que lo forman) tal, que la medida de computación *m(n)*, que usualmente expresa tiempo de ejecución y cantidad utilizada de memoria, es igual a *O(n)*.

2.8.1. Polinomial

Se dice de la clase de algoritmos cuyo límite de complejidad está definido por una función polinomial para un problema de tamaño *n* tal, que la medida de computación *m(n)* es igual a $O(n^k)$.

Aunque en sentido estricto la complejidad lineal es del tipo polinomial con un exponente no mayor a 1, y la complejidad logarítmica es también polinomial, pero con fracciones como exponentes, dentro de este tipo de clase de complejidad acostumbra incluirse sólo aquellos algoritmos cuyos exponentes son mayores a 1.

2.8.2. Exponencial

Esta clase de complejidad se asocia con algoritmos cuyo límite de complejidad está definido por una función exponencial para un problema de tamaño *n*, donde existe $k > 1$ tal que $m(n) = Q(k^n)$, y existe *c* tal que $m(n) = O(c^n)$.

Logarítmica. Se dice de la clase de algoritmos cuyo límite de complejidad está definido por una función logarítmica para un problema de tamaño *n* tal, que la medida de computación *m(n)* es igual a *O(log n)*.

2.8.3. Notaciones

Las notaciones son medidas teóricas del tiempo y memoria necesarias para la ejecución de un algoritmo, dado un problema de tamaño *n* que representa el número de elementos que lo forman. La utilidad de este tipo de medidas radica en su capacidad de poder definir el costo de cada algoritmo proveyendo los elementos de decisión necesarios para la elección de la mejor opción.

Existen dos tipos de notaciones estándares que permiten hacer esta medición: *O* y *o*. La notación O, mejor conocida como O mayúscula, *big-O* o como la letra griega *Omicrón* mayúscula, es una medida teórica de la cantidad de recursos que consume un algoritmo en su ejecución. Si tenemos una ecuación $f(n) = O(g(n))$, significa que es menor a alguna constante múltiplo de $g(n)$ o, dicho de otra forma, existen las constantes positivas *c* y *k* tal que $0 \square f(n) \square cg(n)$ para toda $n \square k$. Los valores *c* y *k* deben ser fijos para la función *f,* e independientes de *n*. Por ejemplo, la complejidad de la expresión $a^2 + 2a + 10$ se puede representar como $O(n^2)$.

Esta medida nos puede orientar sobre la complejidad del algoritmo de ordenamiento *quicksort* que se representa en promedio como $O(n \log n)$ y que, corriendo en una computadora personal con una cantidad grande de elementos para ordenar, puede competir con el algoritmo *bubblesort*, con un comportamiento promedio de $O(n^2)$ corriendo en una supercomputadora. Esto se explica porque la cantidad de recursos requeridos para ordenar un millón de miembros le lleva seis millones de pasos al *quicksort*, mientras que le llevaría un billón de pasos al *bubblesort*.

La notación *o*, conocida como *o minúscula*, *little-o* o como la letra griega *Omicrón* minúscula, también es una medida teórica de *a* cantidad de recursos que consume un algoritmo en su ejecución. La ecuación $f(n) = o(g(n))$ significa que $f(n)$ se vuelve insignificante con relación a $g(n)$ cuando *n* se aproxima al infinito, o dicho de otra manera, para toda $c > 0$ existe $k > 0$ tal que $0 \square f(n) < cg(n)$ para toda $n \square k$. El valor de *k* debe ser independiente de *n* y puede depender de *c*.

Direcciones electrónicas

http://obiwan.uvi.edu/computing/turing/ture.htm

http://obiwan.uvi.edu/computing/turing/more.htm

http://www-csli.stanford.edu/hp/Turing1.html

http://hissa.nist.gov/dads

Bibliografía de la Unidad

Unidad 3. Funciones recursivas

Temario detallado

3. Funciones recursivas
   - 3.1 Introducción a la inducción
   - 3.2 Definición de funciones recursivas
   - 3.3 Cálculo de la complejidad de una función recursiva

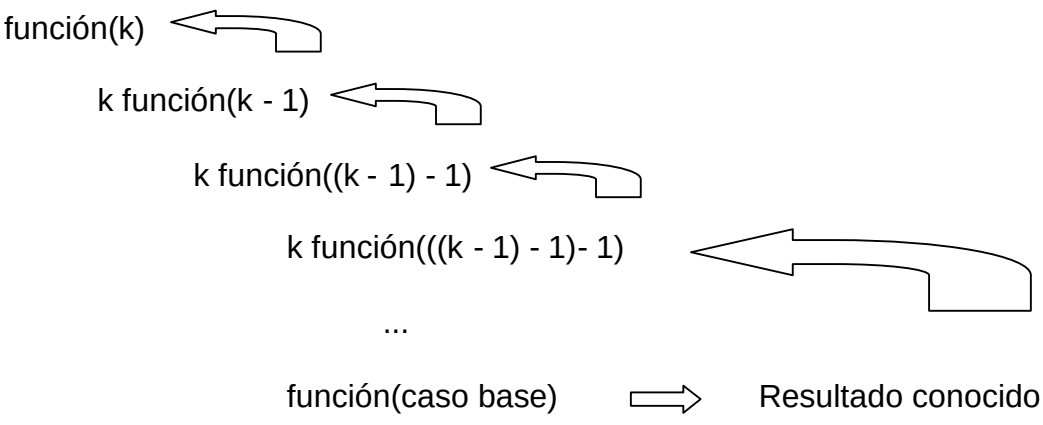
3. Funciones recursivas

Las funciones recursivas son aquellas que hacen llamadas a sí mismas en su definición, simplificando los valores originales de entrada.

Las funciones recursivas se pueden implementar cuando el problema que se desea resolver puede simplificarse en versiones más pequeñas del mismo problema, hasta llegar a casos simples de fácil resolución.

Los primeros pasos de una función recursiva corresponden a la cláusula base, que es el caso conocido hasta donde la función descenderá para comenzar a regresar los resultados, hasta llegar a la función con el valor que la invocó.

El funcionamiento de una función recursiva puede verse como:



3.1. Introducción a la inducción

La recursión se define a partir de tres elementos; uno de éstos es la cláusula de inducción. A través de la inducción matemática se puede definir un mecanismo para encontrar todos los posibles elementos de un conjunto recursivo partiendo de un subconjunto conocido, o bien, para probar la validez de la definición de una función recursiva a partir de un caso base.

El mecanismo que la inducción define, parte del establecimiento de reglas que permiten generar nuevos elementos tomando como punto de partida una semilla (caso base).

Existen dos principios básicos que sustentan la aplicación de la inducción matemática en la definición de recursión:

Primer principio. Si el paso base y el paso inductivo asociados a una función recursiva pueden ser probados, la función será válida para todos los casos que caigan dentro del dominio de la misma. Asimismo, establece que si un elemento arbitrario del dominio cumple con las propiedades definidas en las cláusulas inductivas, su sucesor o predecesor, generado según la cláusula inductiva, también cumplirá con dichas propiedades.

Segundo principio. Se basa en afirmaciones de la forma $\forall x\, P(x)$. Esta forma de inducción no requiere del paso básico, ya que asume que $P(x)$ es válido para todo elemento del dominio.

3.2. Definición de funciones recursivas

Como se mencionó anteriormente, la definición recursiva consta de tres cláusulas diferentes: básica, inductiva y extrema.

Cláusula básica. Especifica la semilla del dominio a partir de la cual se generarán todos los elementos del contradominio (resultado de la función), utilizando las reglas definidas en la cláusula inductiva. Este conjunto de elementos se denomina caso base de la función que se está definiendo.

Cláusula inductiva. Establece la manera en que los elementos del dominio pueden ser combinados para generar los elementos del contradominio. Esta cláusula

afirma que, a partir de los elementos del dominio, se puede generar un contradominio con propiedades análogas.

Cláusula extrema. Afirma que a menos que el contradominio demuestre ser un valor válido, aplicando las cláusulas base e inductiva un número finito de veces, la función no será valida.

A continuación se desarrolla un ejemplo de la definición de las cláusulas para una función recursiva que genera números naturales:

Paso básico. Demostrar que $P(n_0)$ es válido.

Inducción. Demostrar que para cualquier entero k n_0, si el valor generado por $P(k)$ es válido, el valor generado por $P(k+1)$ también es válido.

A través de la demostración de estas cláusulas, se puede certificar la validez de la función $P(n_0)$ para la generación de números naturales.

3.3. Cálculo de complejidad de una función recursiva

Generalmente las funciones recursivas, por su funcionamiento de llamadas a sí mismas, requieren mucha mayor cantidad de recursos (memoria y tiempo de procesador) que los algoritmos iterativos.

Un método para el cálculo de la complejidad de una función recursiva consiste en calcular la complejidad individual de la función y después elevar esta función a n, donde n es el número estimado de veces que la función deberá llamarse a sí misma antes de llegar al caso base.

Para calcular la complejidad de la función de manera individual, pueden utilizarse los métodos descritos en el apartado 2.8 de este tutorial.

Direcciones electrónicas

http://www-lsi.upc.es/~rbaeza/handbook/expand.html

http://www.cs.odu.edu/~toida/nerzic/content/recursive_alg/rec_alg.html
http://www.cs.odu.edu/~toida/nerzic/content/web_course.html

Bibliografía de la Unidad

Unidad 4. Diseño de algoritmos para la solución de problemas

Temario detallado

4. Diseño de algoritmos para la solución de problemas
    - 4.1 Niveles de abstracción para la construcción de algoritmos
    - 4.2 Estructuras básicas en un algoritmo
        - 4.2.1 Ciclos
        - 4.2.2 Contadores
        - 4.2.3 Acumuladores
        - 4.2.4 Condicionales
        - 4.2.5 Interruptores
    - 4.3 Rutinas recursivas
    - 4.4 Refinamiento progresivo
    - 4.5 Procesamiento regresivo

4. Diseño de algoritmos para la solución de problemas

Como se ha explicado en las unidades anteriores, dentro del universo de problemas existen unos sin solución y otros con solución; igualmente, como un subconjunto de estos últimos, hay problemas computables y no computables. A través de los algoritmos se pueden implementar diferentes formas de solución para los problemas computables.

En esta unidad se describirá un método por medio del cual se pueden construir algoritmos para la solución de problemas, además de las características de algunas estructuras básicas usadas típicamente en la implementación de estas soluciones.

4.1. Niveles de abstracción para la construcción de algoritmos

En la implementación de soluciones mediante algoritmos pueden utilizarse diferentes métodos; aquí se explica uno muy generalizado: el ciclo de vida. Este método se basa en la abstracción de las características del problema a través de un proceso de análisis, para seguir con el diseño de una solución basada en modelos, los cuales ven su representación tangible en el proceso de implementación del algoritmo.

El primer proceso, análisis, consiste en reconocer cada una de las características del problema, lo cual se logra señalando los procesos, subprocesos y variables que rodean al problema. Los procesos pueden identificarse como operaciones que se aplican a las variables del problema. Al analizar los procesos o funciones del problema, éstos deben relacionarse con sus variables, pudiendo diferenciar entre variables de entrada, salida e internas, aunque también existen variables que toman un doble rol, tanto de entrada como de salida, y que se ven afectadas a lo largo de todos los procesos del problema; éstas se conocen como variables generales y debe ponerse atención especial sobre ellas, ya que juegan un papel fundamental en la resolución del problema. El resultado esperado de esta fase de la construcción de un algoritmo es un modelo que represente la problemática encontrada y permita identificar sus características más relevantes. Al construir ese modelo, se debe tener en cuenta que sólo se busca representar las características del proceso que estén relacionadas con la solución que se quiere alcanzar, ya que, de no hacerlo así, se puede perder la objetividad del análisis y se corre el riesgo de recabar volúmenes de información irrelevantes que sólo harán más difícil la definición de una solución óptima.

Una vez que se han analizado las causas del problema y se ha identificado el punto exacto donde radica y sobre el cual se debe actuar para llegar a una solución, comienza el proceso de modelado de una solución factible, es decir, el diseño. En esta etapa se debe estudiar el modelo del problema, elaborar hipótesis acerca de posibles soluciones y comenzar a realizar pruebas con éstas. Existen numerosas técnicas para encontrar la solución al problema, entre ellas podemos citar el método

de aproximaciones sucesivas y el método de refinamiento regresivo; ambos métodos se describirán más adelante.

Por último, ya que se tiene modelada la solución, ésta debe implementarse usando el lenguaje de programación más adecuado para ello.

Hay que mencionar que no siempre se sigue este método debido a las circunstancias que rodean el desarrollo de la solución, esto es, en la mayoría de las ocasiones se busca una solución orientada hacia algún lenguaje predefinido o se debe partir del desarrollo de prototipos que se refinarán posteriormente; sin embargo, aquí sólo se ha descrito un método de los muchos que existen para este propósito.

4.2. Estructuras básicas en un algoritmo

En el modelado de soluciones mediante el uso de algoritmos es común encontrar ciertos comportamientos clásicos que tienen una representación a través de modelos ya definidos; a continuación se explican sus características.

4.2.1. Ciclos

Estas estructuras se caracterizan por iterar instrucciones en función de una condición que debe cumplirse en un momento bien definido. Existen dos tipos de ciclos: *while* y *until*. El primero se caracteriza por realizar la verificación de la condición antes de ejecutar las instrucciones asociadas al ciclo; el segundo evalúa la condición después de ejecutar las instrucciones una vez. Las instrucciones definidas dentro de ambos ciclos deben modificar, en algún punto, la condición para que sea alcanzable, de otra manera serían ciclos infinitos, un error de programación común. En este tipo de ciclos el número de iteraciones que se realizarán es variable y depende del contexto de ejecución del algoritmo.

El ciclo *while* tiene el siguiente pseudocódigo:

while <condicion> do

Instruccion1

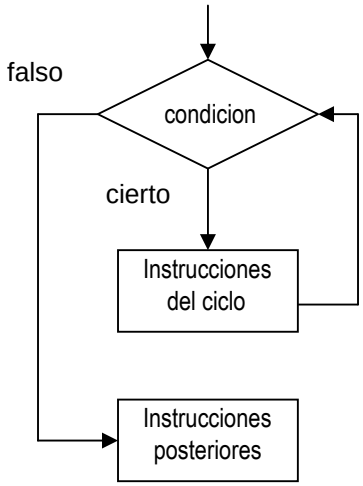
Instruccion2

...

Instruccion*n*

end

El diagrama asociado a este tipo de ciclo es el siguiente:



Por otro lado, el pseudocódigo asociado a la instrucción *until* se define como sigue:

do

Instruccion1

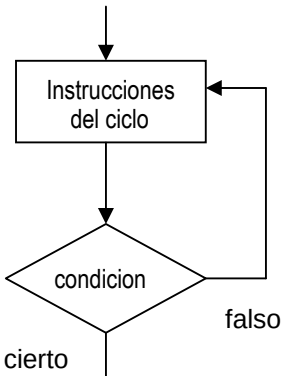
Instruccion2

...

Instruccion*n*

Until <condicion>

Su diagrama se puede representar como:



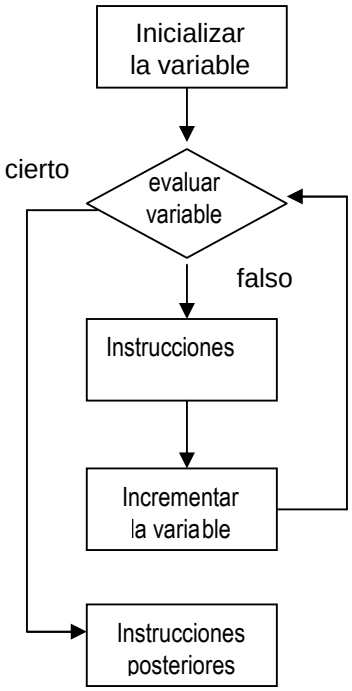
4.2.2. Contadores

Este otro tipo de estructura también se caracteriza por iterar instrucciones en función de una condición que debe cumplirse en un momento conocido y está representado por la instrucción *for*. En esta estructura se evalúa el valor de una variable a la que se asigna un valor conocido al inicio de las iteraciones; este valor sufre incrementos en cada iteración y suspende la ejecución de las instrucciones asociadas una vez que se alcanza el valor esperado. En algunos lenguajes se puede definir el incremento que tendrá la variable, sin embargo, se recomienda que el incremento siempre sea con la unidad. El pseudocódigo que representa la estructura *for* es el siguiente:

For <variable> = <valor inicial> to <valor tope> [step <incremento>]
do
Instruccion1
Instruccion2
...
Instruccion*n*
[<variable> = <variable> + <incremento>]
end

Aquí se puede observar entre los símbolos [ ] las dos opciones que existen para efectuar el incremento a la variable. En el primero, se realiza el incremento como parte del encabezado de la estructura, esta sintaxis no está disponible en todos los lenguajes de programación; en el segundo, el incremento se realiza en el cuerpo de la estructura.

A continuación se muestra el diagrama asociado a esta estructura:



4.2.3. Acumuladores

Los acumuladores son variables que tienen por objeto almacenar valores incrementales o decrementales a lo largo de la ejecución del algoritmo. Este tipo de variables utiliza la asignación recursiva de valores para no perder su valor anterior. Su misión es arrastrar un valor que se va modificando con la aplicación de diversas operaciones y cuyos valores intermedios, así como el final, son importantes para el resultado global del algoritmo. Este tipo de variables generalmente almacena el valor de la solución arrojada por el algoritmo.

La asignación recursiva de valor a este tipo de variables se ejemplifica a continuación:

<variable> = <variable> + <incremento>

4.2.4. Condicionales

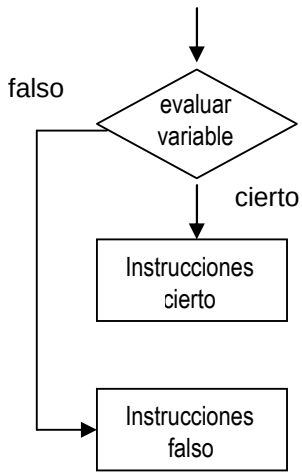
Este tipo de estructura se utiliza para ejecutar selectivamente secciones de código de acuerdo con una condición definida. Este tipo de estructura sólo tiene dos posibilidades: si la condición se cumple, se ejecuta una sección de código; si no, se ejecuta otra sección, aunque esta parte puede omitirse. Es importante mencionar que se pueden anidar tantas condiciones como lo permita el lenguaje de programación en el que se implementa el programa.

El pseudocódigo básico que representa la estructura *if* es el siguiente:

```
If <condicion> Then
        Instruccion1
        Instruccion2
[Else
        Instruccion3
        Instruccion n]
Endif
```

Dentro del bloque de instrucciones que se definen en las opciones de la estructura *if* se pueden insertar otras estructuras condicionales anidadas. Entre los símbolos [ ] se encuentra la parte opcional de la estructura.

El diagrama asociado a esta estructura se muestra a continuación:



4.2.5. Interruptores

Son variables globales del programa que se utilizan en combinación con las estructuras cíclicas y selectivas para habilitar o deshabilitar secciones de código. Un ejemplo de este tipo de variables son las banderas que, dependiendo del valor que almacenen, condicionan el comportamiento del programa.

4.3. Rutinas recursivas

Las rutinas recursivas son aquellas que hacen llamadas a sí mismas en su definición, simplificando los valores originales de entrada.

Este tipo de rutinas se puede implementar en los casos en que el problema que se desea resolver puede simplificarse en versiones más pequeñas del mismo problema, hasta llegar a casos simples de fácil resolución.

Las rutinas recursivas regularmente contienen una cláusula condicional (*if*) que permite diferenciar entre el caso base, situación final en que se regresa un valor como resultado de la rutina, o bien, un caso intermedio, que es cuando se invoca la rutina a sí misma con valores simplificados.

Es importante no confundir una rutina recursiva con una rutina cíclica, por ello se muestra a continuación el pseudocódigo genérico de una rutina recursiva:

```
<valor_retorno> Nombre_Rutina (<parametroa> [,<parametrob> ...])
        if <caso_base> then
                return <valor_retorno>
        else
Nombre_Rutina ( <parametroa -1> [, <parametrob -1> ...] )
End
```

Como se observa en el ejemplo, en esta rutina es obligatoria la existencia de un valor de retorno, una estructura condicional y, cuando menos, un parámetro.

El diagrama asociado a este tipo de rutinas ya se ha ejemplificado en la sección de funciones recursivas.

Por último, debemos mencionar que el pseudocódigo definido en esta sección no es la única forma de definición existente para funciones recursivas (existen otras sintaxis en lenguajes como LISP), pero es la más aceptada para diferentes tipos de paradigmas de programación.

4.4. Refinamiento progresivo

Es una técnica de análisis y diseño de algoritmos que se basa en la división del problema principal en problemas más simples. Partiendo de problemas más simples, se logra dar una solución más efectiva, ya que el número de variables y casos asociados a un problema simple es más fácil de manejar que el problema completo. Este tipo de procedimiento se conoce como *Top-Down* y también es aplicable a la optimización del desempeño y a la simplificación de un algoritmo.

4.5. Procesamiento regresivo

Ésta es otra técnica de análisis y diseño de algoritmos que parte de la existencia de múltiples problemas y se enfoca en la asociación e identificación de características comunes entre ellos para diseñar un modelo que represente la solución para todos los casos de acuerdo con la parametrización de las entradas. Esta técnica también es conocida como *Bottom-Up* y puede aplicarse en la optimización y simplificación de algoritmos.

Direcciones electrónicas

http://hissa.nist.gov/dads

http://www.basicguru.com/abc/qb/qbmain.html

http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

http://compy.ww.tu-berlin.de/IBMVoice/proguide/

Bibliografía de la Unidad

Unidad 5. Evaluación de algoritmos

Temario detallado

5. Evaluación de algoritmos

5. Evaluación de algoritmos

La evaluación del algoritmo es un proceso que tiene por objeto analizar su desempeño tanto en el aspecto del tiempo que tarda para encontrar una solución como en la cantidad de recursos que emplea para llegar a ella.

Existen numerosas técnicas para evaluar la eficiencia de un algoritmo, pero las más confiables son las que se enfocan a la medición de la complejidad del algoritmo usando funciones matemáticas bien definidas. Dentro de las técnicas de evaluación de algoritmos se pueden encontrar *O*, o, □, □, □ y ~. A continuación se explican las principales características de estas técnicas.

- O. Está basada en la definición de □̃
- *o*. Está basada en la definición de □.
- □. Si tenemos una ecuación *f(n) = O(g(n))*, significa que es menor a alguna constante múltiplo de *g(n).*
- □. Si tenemos la ecuación *f(n) = o(g(n))*, significa que *f(n)* se vuelve insignificante con relación a *g(n)* cuando *n* se aproxima al infinito.
- □. Dada la ecuación f(n) = □ (g(n)), significa que existe más de una constante múltiplo de g(n). De manera formal podemos decir que, dadas las constantes positivas *c* y *k*, tenemos que *0 cg*(*n*) *f*(*n*) para toda *n k*. Los valores *c* y *k* deben ser fijos para la función *f* y no depender de *n*, donde *n* es el número de elementos que componen el problema a resolver por el algoritmo.
- ~. Dado un problema de tamaño *n*, donde *n* representa la cantidad de elementos que lo componen, podemos enunciar la ecuación *f*(*n*)~*g*(*n*) cuyo

significado es que crece en la misma proporción que $g(n)$. Su significado formal es $\lim_{x \to \infty} f(x)/g(x) = 1$.

Este tipo de medidas de la complejidad de los algoritmos permite comparar la eficiencia de diferentes métodos para la resolución del mismo tipo de problemas. De esta manera, se puede encontrar que el algoritmo de ordenamiento *quicksort* resulta mucho más eficiente que el algoritmo *bubblesort* al ejecutarse sobre un número de elementos grande.

Como se ha observado, la evaluación teórica de los algoritmos es un auxiliar en la toma de decisiones acerca del algoritmo más eficiente para la solución de un problema, ya que permite identificar las diferencias más relevantes de los algoritmos al analizar su estructura y lógica de programación, sin necesidad de ejecutarlos.

Otro auxiliar en la evaluación de los algoritmos lo representan las pruebas de escritorio, que consisten en diseñar una tabla donde se van anotando los valores parciales de las variables que intervienen en la ejecución del algoritmo. De esta manera, se pueden identificar variables no usadas, ciclos infinitos, interrupciones no necesarias y pasos que pueden eliminarse. Además, muestra la cantidad de pasos necesarios para la solución del problema planteado.

En la actualidad se pueden encontrar en el mercado herramientas que permiten evaluar la eficiencia de los algoritmos programados a través del monitoreo de su ejecución. Estos programas basan su funcionamiento en la intercepción de los intercambios de información entre el programa que se ejecuta y el dispositivo de almacenamiento que guarda la información que se procesa (disco duro o memoria). De acuerdo con la lectura que estos programas hacen de los intercambios de información, elaboran informes estadísticos donde resaltan la relación de entradas/salidas que efectúa el programa, el uso de las variables, la eficiencia de los ciclos, etcétera, todo ello medido sobre la base de estándares predefinidos en el programa.

Ejemplos de este tipo de programas son HATES y Zirano. El primero realiza evaluación de programas en distintas plataformas detectando *bugs* y emitiendo un

diagnóstico de la eficiencia del programa; el segundo, se ejecuta sobre el DBMS Sybase y evalúa la ejecución de consultas, *triggers* y procedimientos almacenados.

Por último, ejemplificaremos la evaluación de dos algoritmos, uno basado en Lógica difusa (*Fuzzy Logic*) y otro basado en el método Monte Carlo, para la identificación de niveles de riesgo en condiciones de incertidumbre sobre la base de una serie de atributos clave:

I. Datos requeridos para el análisis

Algoritmo difuso	Algoritmo Monte Carlo
❑ Parametrización previa del sistema, modificadores y términos primarios. ❑ Expresión en lenguaje natural del usuario.	❑ Parametrización previa del sistema, establecimiento de la distribución o de un método alterno para lograr la distribución, probabilidades de las distintas rutas. ❑ Indicación de un punto inicial de partida arbitrario por parte del usuario.

II. Tipos de riesgo que analiza

Algoritmo difuso	Algoritmo Monte Carlo
❑ Cualquier tipo de riesgo. Ver el riesgo que se analiza depende de la definición del BNF, el cual dota de flexibilidad a este método.	❑ Analiza cuestiones más bien financieras, aunque puede adaptarse a un número limitado de áreas afines.

III. Consumo de recursos

Algoritmo difuso	Algoritmo Monte Carlo

❑ No tiene grandes requerimientos de recursos, y se vuelve aún más eficiente si, para aquellos casos en los que existe un gran número de variables, se usa el método, ya explicado, de aproximaciones sucesivas.	❑ Por la naturaleza compleja de las operaciones que realiza, además de la cantidad de estas operaciones que se requiere para generar un resultado preciso que posea altos parámetros de confiabilidad, este método resulta sumamente costoso.
❑ Su tiempo de respuesta se puede calificar como muy bueno en comparación a Monte Carlo.	❑ Requiere de gran cantidad de tiempo en comparación al método difuso.

IV. Manejo de dependencias

Algoritmo difuso	Algoritmo Monte Carlo
❑ Asume la existencia de dependencias dentro de la definición misma de sus términos primarios y sus modificadores.	❑ Asume que todas las variables son independientes.
❑ Con dependencias desconocidas, este método se porta conservador, lo que evita tomar riesgos innecesarios.	❑ Como no toma en cuenta las dependencias se muestra demasiado temerario en su interpretación.

V. Comprensión de las entradas – salidas

Algoritmo difuso	Algoritmo Monte Carlo
❑ Al valerse de lenguaje natural no requiere de procesos extras para comprender los resultados ni para preparar las entradas.	❑ Las entradas y salidas por sí mismas no nos dicen nada acerca de la naturaleza del problema o de su solución; requieren de procesos extra para comprensión.

VI. Comprensión del proceso

Algoritmo difuso	Algoritmo Monte Carlo
❑ El proceso es difícil de comprender para la gente	❑ Los desarrolladores del

familiarizada con el desarrollo de sistemas de análisis por el hecho de que no sólo se debe entender el método en sí, sino que este método implica la ruptura de antiguos paradigmas que tienen gran aceptación y arraigo.

- Existe un índice muy alto de resistencia al cambio.

análisis de riesgos conocen perfectamente el método Monte Carlo y su basamento teórico, lo cual facilita su implementación.

De acuerdo con la evaluación anterior, podemos concluir que el método basado en Lógica difusa es más eficiente que el método Monte Carlo, sólo que tiene en su contra el desconocimiento y la resistencia al cambio de paradigmas. Este último punto puede ser decisivo en la implantación de una solución en una empresa, influyendo directamente en su éxito.

Direcciones electrónicas

http://www.npac.syr.edu/users/miloje/Finance/Option/option/section3_4.html
http://vase.essex.ac.uk/projects/hate/
http://hissa.nist.gov/dads

Bibliografía de la Unidad

Unidad 6. Estrategias de programación para la implantación de algoritmos

Temario detallado

6. Estrategias de programación para la implantación de algoritmos.
   - 6.1 Programación imperativa
   - 6.2 Programación lógica
   - 6.3 Programación funcional
   - 6.4 Programación orientada a objetos

6. Estrategias de programación para la implantación de algoritmos

Si bien es cierto que la implantación de los algoritmos diseñados puede efectuarse prácticamente en cualquier tipo de lenguaje de programación, no debemos pasar por alto que existen lenguajes de programación que funcionan de manera más eficiente bajo ciertas características de los algoritmos. Con esto queremos decir que, dependiendo de las características de la solución diseñada usando algoritmos, debemos elegir el paradigma de programación que pueda implementar de manera más simple dicho diseño.

A continuación veremos una descripción de los principales paradigmas de programación sobre los que se puede implementar un algoritmo en la actualidad.

6.1. Programación imperativa

La programación imperativa, también conocida como programación procedural, está basada en los lenguajes imperativos, que son la forma tradicional de programación y es totalmente antagónica con la programación en lenguajes declarativos.

Este tipo de programación se basa en la especificación explícita de los pasos que se deben seguir para obtener el resultado deseado, es decir, toma canónicamente la definición de algoritmo, en el sentido de una serie de pasos con

lógica y secuencia para alcanzar un objetivo determinado, implantándola en su estilo de programación. Utiliza variables y operaciones de asignación para el almacenamiento de información, define procedimientos bien claros en donde se procesan las variables de acuerdo con operaciones explícitas.

El siguiente es un ejemplo de programa en lenguaje Basic:

```
DIM NOMBRE AS STRING,                                   ' Define las variables a usar
    NACIMIENTO AS INTEGER,
    ACTUAL AS INTEGER

PRINT "Escribe tu nombre"; NOMBRE                       'Solicita los valores de entrada
PRINT "Escribe el Año Actual: ";ACTUAL                  'y los asigna a variables
PRINT "Escribe el Año en que naciste:";NACIMIENTO
PRINT NOMBRE                                            'Imprime el valor de NOMBRE
IF (ACTUAL - 18)  > NACIMIENTO THEN                     Estructura de decisión...
        PRINT "Eres mayor de Edad"                      'Si la condición es cierta...
ELSE
        PRINT "Eres menor de Edad"                       'Si la condición es falsa...
END IF                                                  'Final de la estructura
END                                                     'Final del programa
```

Algunos de los lenguajes para programación imperativa son Basic, Pascal, C y Modula - 2.

6.2. Programación lógica

Es un tipo de programación declarativa y relacional que está basada en lógica de primer orden. La programación declarativa describe relaciones entre variables en términos de funciones y reglas de inferencia (procedimiento que infiere hechos a

partir de otros hechos conocidos), dejando en manos del traductor la aplicación de un algoritmo fijo sobre estas relaciones para producir un resultado. La programación relacional genera salidas en función de atributos y argumentos.

El lenguaje de programación lógica por excelencia es Prolog, que basa su funcionamiento en cláusulas formadas por un conjunto de literales atómicas donde por lo menos una es positiva; esto se expresa de la siguiente manera:

L L1,...,Ln

o

 L1,...,Ln

En este paradigma, el programador debe definir una base de datos de hechos como:

mojar (agua)

que relaciona agua con mojar, y una serie de reglas como:

mortal (X):- humano(X)

lo que indica que si X es humano, entonces X es mortal. Hechos y reglas son conocidos, en su conjunto, como cláusulas.

El usuario proporciona un predicado (notación que representa instrucciones lógicas que van más allá del cálculo proposicional) que el sistema trata de probar mediante el método de resolución o el método de encadenamiento en reversa. El primero es un método mecánico para probar declaraciones lógicas de primer orden, se aplica sobre dos cláusulas en una sentencia, eliminando por unificación aquellas literales que aparecen positivas en una de ellas y negativas en la otra, dando como resultado una cláusula de resolución. El segundo método utiliza un algoritmo que provee recursividad para alcanzar la solución a través de llamadas anidadas hasta llegar al caso base que puede verse como un hecho conocido.

Cada vez que se llega a puntos donde se debe tomar una decisión, se crea un punto de verificación que se almacena en una estructura de pila. Si la resolución a partir de la decisión tomada falla, se regresa al punto de verificación y se toma el camino alternativo. Este tipo de funcionamiento se conoce como rastreo en reversa (*backtracking*).

Al final, el usuario es informado del éxito o falla de su primer objetivo definiendo las variables usadas y sus valores, e identificando cuáles han sido la causa del resultado. De esta manera, el usuario puede modificar los valores para tratar de encontrar soluciones alternativas.

6.3. Programación funcional

Este paradigma basa su programación en la definición de un conjunto de funciones (posiblemente recursivas) y una expresión cuyo valor de salida se utilizará para representar el resultado del algoritmo.

La programación funcional está basada en un tipo de lenguaje declarativo, ya explicado en el punto anterior y en cálculo lambda tipificado con constantes. Las funciones de este tipo de programas no modifican su salida con entradas iguales, es decir, no tienen efectos laterales, lo que implica también que cumple con la regla de transparencia referencial.

Para aclarar el punto anterior desarrollemos el siguiente ejemplo. Suponiendo que existe una variable global denotada como *vg1*, tenemos el siguiente caso:

```
f(vl1)
{ return vl1+vg1; }

g(vl2)
{
  vl3 = f(1);
  vg1 = vg1 + vl2;
  return vl3 + f(1);
}
```

En este punto podemos observar que la función *f* no es referencialmente transparente, ya que depende del valor de la variable *vg1*, de manera tal que el valor que entregará *f(1)* en la asignación de *vl3* será diferente del valor que entregará la segunda vez que se invoque dentro de la función *g*. La versión de estas funciones que cumple con la transparencia referencial se desarrolla a continuación:

```
f(vf1, vf2)
{ return vf1+vf2; }

g(vg1,vg2)
{
  vg3 = f(1,vg2);
  vg4 = vg2 + vg1;
  return (vg3 + f(1, vg4), vg4);
}
```

En este segundo ejemplo se puede ver que el comportamiento de *f* no varía en relación con variables ajenas a su definición, es decir, no está en función de variables globales que puedan ser afectadas por otras secciones de código. De lo anterior, podemos concluir que en este ejemplo sí se cumple con la regla de transparencia referencial.

El orden en que se evalúan las subexpresiones está determinado por la estrategia de evaluación: en un lenguaje estricto, con llamadas por valor, se especifica que los argumentos son evaluados antes de ser pasados a la función; en un lenguaje no estricto, con llamadas por nombre, los argumentos de la función se pasan sin ser evaluados.

Los programas escritos en lenguaje funcional tienden a ser compactos y elegantes, aunque también son lentos y requieren de gran cantidad de memoria para su ejecución.

Algunos de los principales lenguajes de programación funcional son Clean, FP, Haskell, Hope, LML, Miranda, SML y LISP.

6.4. Programación orientada a objetos

El último paradigma de programación que veremos en este tutorial es el definido por la Programación Orientada a Objetos (OOP, por sus siglas en inglés). La programación orientada a objetos tiene por principio la representación de los objetos de la realidad en un lenguaje de programación que permita acercarse lo más posible al pensamiento humano, así como a su modo de concebir y representar la realidad.

A partir de este principio, la programación orientada a objetos postula la identificación de los objetos que participan en el problema o situación que se quiere solucionar a través de un algoritmo. Realizada la identificación de los objetos, se lleva a cabo una abstracción de sus características más relevantes, así como de las principales operaciones que efectúan (dentro del contexto del problema a resolver).

Posteriormente, se busca asociar objetos con características y operaciones comunes para llegar a la definición de una clase.

Una clase posee atributos, interfaz y métodos. Los atributos son características o datos que tienen las clases, la interfaz es la parte de la clase que posibilita la comunicación entre objetos, los métodos son las operaciones o funciones asociadas a los objetos a través de los cuales se pueden modificar los valores que tienen los atributos de los objetos de la clase.

Un objeto es una instancia de una clase, esto implica que un objeto siempre tendrá una clase asociada de la cual hereda su estructura (métodos y atributos). La herencia debe entenderse como la extensión de las características de una clase hacia otra clase (subclase) o hacia un objeto en particular.

La programación orientada a objetos también postula el principio de ocultamiento de información (encapsulamiento) que se basa en la definición de una parte pública y otra privada del objeto. La parte pública puede ser accedida y modificada por los métodos públicos del objeto (un atributo sólo puede accederse y

modificarse a través de sus métodos), mientras que la parte pública sólo puede accederse y modificarse usando los métodos privados; estos últimos sólo pueden ser invocados desde los métodos públicos. Según este principio, en un objeto se pueden conocer los argumentos de entrada y salida de los métodos, pero no se permite saber cómo está construido el objeto de manera interna.

El último principio de la programación orientada a objetos que aquí trataremos es el de reusabilidad, que se refiere a reciclar piezas de código ya definidas dentro de las clases para abreviar el tiempo de desarrollo del software. Con esta premisa, se puede pensar en construir grandes repositorios de clases que almacenen su definición y a los cuales se puede hacer referencia en caso de necesitarlos.

Algunos de los lenguajes de programación más usados en este paradigma son Eiffel, C++, Power Builder y Delphi (*Object Oriented Pascal*). Enseguida presentamos un ejemplo de la definición de una clase desarrollada en Eiffel:

```
class ADIOS inherit

   MOTIF_APP
      redefine
         build
      end

   COMMAND

creation

   make

feature -- Componentes de la Interfaz

   ok_button: PUSH_B
         -- Botón Aceptar
```

```
    form: FORM
        -- Forma donde se desplegará el diálogo

    text_area: TEXT_FIELD
        -- Campo donde el usuario digitará el nombre requerido

    label_area: LABEL
        -- Area que contendrá la etiqueta

    message_dialog: MESSAGE_D
        -- Caja de diálogo Popup

    welcome_message: STRING is "Hola, ¿Cómo te llamas?"
        -- Mensaje inicial que se desplegará

feature -- Inicialización

    build is
        -- Activa la ejecución al crear y anexar los widgets
        -- y activar los comandos.
      do
        create_widgets
        attach_widgets
        size_widgets
        add_commands
      end

    create_widgets is
        -- Crea los widgets que aparecerán en la forma.
      do
        !! form.make ("form", base)
```

```
            !! ok_button.make ("ok", form)
            !! text_area.make ("text", form)
            !! label_area.make ("label", form)
            !! message_dialog.make ("message", form)
            ok_button.set_text ("OK")
        end

    attach_widgets is
            -- Anexa los widgets cada uno para mantener una apariencia
            -- homogénea al modificar el tamaño.
        do
            form.attach_top (label_area, 10)
            form.attach_left (label_area, 10)
            form.attach_right (label_area, 10)
            form.attach_left (text_area, 10)
            form.attach_right (text_area, 10)
            form.attach_bottom_widget (text_area, label_area, 40)
            form.attach_bottom_widget (ok_button, text_area, 40)
            form.attach_left (ok_button, 10)
form.attach_bottom (ok_button, 10)
            form.attach_right (ok_button, 10)
        end

    size_widgets is
            -- Actualiza el tamaño de los widgets.
        do
            base.set_size (500, 200)
            text_area.set_size(250, 50)
            message_dialog.set_size (300, 100)
            label_area.set_center_alignment
```

```
            label_area.set_text (welcome_message)
            ok_button.set_size (230, 30)
        end

    add_commands is
            -- Actualiza el manejador de eventos para que los eventos de
            -- usuario respondan como se requiere.
        do
            -- Un click en el botón de Aceptar activará el diálogo
            ok_button.add_activate_action (Current, ok_arg)

            -- Presionar la tecla Enter en el campo de texto también
            -- activará el diálogo
            text_area.add_activate_action (Current, ok_arg)

            -- Un click en la ventana popup la hará salir
            message_dialog.add_ok_action (Current, close_arg)

-- Oculta los botones de ayuda y cancelación del popup
            message_dialog.hide_cancel_button
            message_dialog.hide_help_button
        end

feature -- Ejecución

    execute (arg: ANY) is
            -- Procesa el evento de usuario
        local
            m: STRING
        do
            if arg = close_arg then
```

```
            exit
        end
        if arg = ok_arg then
            m := "Adiós, "
            m.append (text_area.text)
            message_dialog.set_message (m)
            message_dialog.popup
        end
    end

feature {NONE} -- Implementación

    close_arg: ANY is
        once
            !!Result
        end

    ok_arg: ANY is
        once
            !!Result
        end

end -- clase ADIOS
```

Direcciones electrónicas

http://eiffel.com/eiffel/page.html
http://www.cs.waikato.ac.nz/~marku/languages.html
http://www.afm.sbu.ac.uk/logic-prog/
http://foldoc.doc.ic.ac.uk/foldoc/index.html

http://www.basicguru.com/abc/qb/qbmain.html

Bibliografía de la Unidad

Bibliografía

Apéndice. Elaboración de un mapa conceptual

Los alumnos del Sistema de Universidad Abierta (SUA), a diferencia de los del escolarizado, estudian por su cuenta las asignaturas del plan de estudios correspondiente. Para asimilar el contenido de éstas, requieren consultar y estudiar la bibliografía específica que se les sugiere en cada unidad, actividad nada sencilla, pero indispensable para que los alumnos puedan desarrollar las actividades de aprendizaje y prepararse para los exámenes. Un recurso educativo del que pueden valerse los estudiantes, es el mapa conceptual.

¿Qué es un mapa conceptual?

- ✓ Es un **resumen o apunte gráfico**.
- ✓ Es un esquema gráfico en **forma de árbol, que muestra la relación existente entre los aspectos esenciales estudiados**, relativos a una unidad de una asignatura o de una asignatura completa, o bien, de un capítulo de un libro o un libro completo.
- ✓ Es una **estructura jerárquica en cuya parte superior** se ubica el aspecto de **mayor nivel de implicación o "término conceptual"**, de éste se derivan otros de **menor grado de implicación** que se relacionan de manera subordinada, por lo que, se localizan en niveles inferiores y así sucesivamente en orden descendente, como se observa en el ejemplo de mapa conceptual de la Introducción a la teoría general de la Administración.

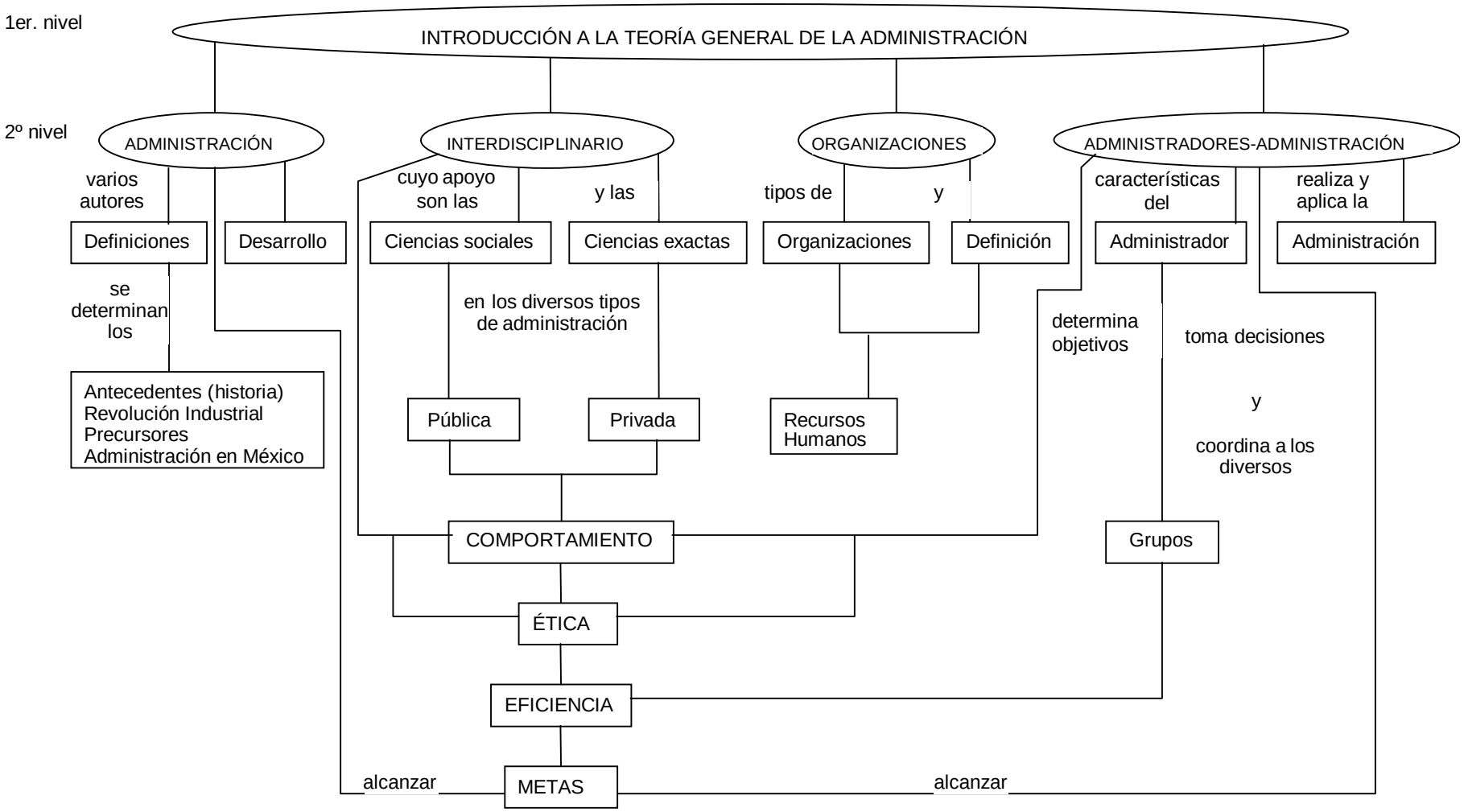
¿Qué ventajas tiene para el alumno un mapa conceptual?

- ✓ Cuando el alumno estudia nuevos contenidos, la construcción de un mapa conceptual le permite **reflexionarlos, comprenderlos y relacionarlos**, es decir, **reorganiza y reconstruye** la información de acuerdo con su propia lógica de entendimiento.
- ✓ Al encontrar las conexiones existentes entre los aspectos esenciales o "términos conceptuales" (clave) del contenido estudiado, el alumno aprenderá a **identificar la información significativa** y a dejar de lado la que no es relevante.
- ✓ El alumno aprende a identificar las ideas principales que el autor de un libro de texto expone, argumenta o analiza; así como a jerarquizarlas y relacionarlas con otros conocimientos que ya posee.
- ✓ La elaboración de un mapa conceptual ayuda a los estudiantes a reproducir con mucha aproximación el contenido estudiado.
- ✓ La construcción de un mapa conceptual estimula en el alumno el **razonamiento deductivo**.

¿Cómo se elabora o construye un mapa conceptual?

1. Realice una primera lectura del capítulo del libro que se le indica en la bibliografía específica sugerida. Preste atención a la introducción y a las notas que el autor hace acerca de los temas y subtemas, porque le ayudarán a comprender la estructura del capítulo; además revise los esquemas, las tablas, las gráficas o cualquier ilustración que se presente. Esta lectura le permitirá tener **una idea general** del contenido del capítulo.
2. Realice una **lectura analítica** del contenido del capítulo, léalo por partes guiándose por la división que el propio autor hace de los temas y subtemas, que por lo general, es más o menos extensa según el tema de que se trate y su complejidad.

3. Lea las ideas contenidas en los párrafos, **analícelos completamente**, ya que en ellos, el autor define, explica y argumenta los aspectos esenciales del capítulo; también describe sus propiedades o características, sus causas y efectos, da ejemplos y, si se requiere, demuestra su aplicación.
4. Al analizar las ideas contenidas en los párrafos, **identifique los "términos conceptuales" o aspectos esenciales** acerca de los cuales el autor proporciona información específica.
5. Elabore un **listado de los principales "términos conceptuales"**. Identifique el papel que juega cada uno de ellos y **ordénelos de los más generales e inclusivos a los más específicos o menos inclusivos**.
6. Liste para cada "término conceptual" lo que el autor aborda: definición, propiedades o características, causas y efectos, ejemplos, aplicaciones, etcétera.
7. Coloque los "términos conceptuales" con los aspectos que en ellos se señalan, **en forma de árbol**. **Encierre** en un círculo o rectángulo cada término. Coloque el de mayor inclusión **en el nivel superior** y el resto, **ordénelo de mayor a menor inclusión**. Verifique que la jerarquización sea correcta.
8. Relacione los "términos conceptuales" **mediante líneas** y si es necesario, **use flechas que indiquen la dirección** de las relaciones. Verifique que las relaciones horizontales y verticales sean correctas, así como las relaciones cruzadas (aquellas que se dan entre "términos conceptuales" ubicados opuestamente, pero que se relacionan entre sí).
9. Construya **frases breves o palabras de enlace** que establezcan o hagan evidente las relaciones entre los "términos conceptuales".
10. Analice el ejemplo del mapa conceptual de la Introducción a la teoría general de la Administración. Identifique los niveles, "los términos conceptuales", los aspectos que de ellos se derivan, las relaciones horizontales, verticales y cruzadas.



Ejemplo de mapa conceptual de la Introducción a la teoría general de la Administración (Profra. Rebeca Novoa)

Tutorial para la asignatura de Análisis, Diseño e implantación de algoritmos es una edición de la Facultad de Contaduría y Administración. Se terminó de imprimir en mayo de 2003. **Tiraje:** 150 ejemplares. **Responsable:** L. A. y Mtra. Gabriela Montero Montiel, Jefa de la División de Universidad Abierta. **Edición a cargo de:** L. A. Francisco Hernández Mendoza y L. C. Aline Gómez Angel. **Revisión a cargo de:** Lic. María del Carmen Márquez González y L. C. Nizaguié Chacón Albarrán.

Dr. Juan Ramón de la Fuente
Rector

Lic. Enrique del Val Blanco
Secretario General

Mtro. Daniel Barrera Pérez
Secretario Administrativo

Lic. Alberto Pérez Blas
Secretario de Servicios a la Comunidad

Dra. Arcelia Quintana Adriano
Abogada General

Dr. José Narro Robles
Coordinador General de Reforma Universitaria

●

C.P.C. y Maestro Arturo Díaz Alonso
Director

L.A.E. Félix Patiño Gómez
Secretario General

Dr. Ignacio Mercado Gasca
Jefe de la División de Estudios de Posgrado

C.P. Eduardo Herrerías Aristi
Jefe de la División de Contaduría

L.A. y Maestro Adrián Méndez Salvatorio
Jefe de la División de Administración

Ing. y Mtra. Graciela Bribiesca Correa
Jefa de la División de Informática

L.A. y Maestro Jorge Ríos Szalay
Jefe de la División de Investigación

L.Ps. y Mtro. Fco. Javier Valdez Alejandre
Jefe de la División de Educación Continua

L.A. y Mtra. Gabriela Montero Montiel
Jefa de la División de Universidad Abierta

L.C. José Lino Rodríguez Sánchez
Secretario de Intercambio Académico

L.A. Carmen Nolasco Gutiérrez
Secretaria de Planeación Académica

L.A. Rosa Martha Barona Peña
Secretaria de Personal Docente

L.A. Gustavo Almaguer Pérez
Secretario de Divulgación y Fomento Editorial

L.A. Hilario Corona Uscanga
Secretario de Relaciones

L.C. Adriana Padilla Morales
Secretaria Administrativa

L.A. María Elena García Hernández
Secretaria de Planeación y Control de Gestión

L.E. José Silvestre Méndez Morales
Subjefe de la División de Estudios de Posgrado

Dr. Sergio Javier Jasso Villazul
Coordinador del Programa de Posgrado en Ciencias de la Administración

L.A., L.C. y Mtro. Rafael Rodríguez Castelán
Subjefe de la División de Estudios Profesionales

L.C. y Mtro. Juan Alberto Adam Siade
Subjefe de la División de Investigación

L.A. y Maestro Eric Manuel Rivera Rivera
Jefe de la División Juriquilla

C.P. Rafael Silva Ramírez
Asesor de la Dirección

L.A. Balfred Santaella Hinojosa
Jefe de Administración Escolar