UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN
DIVISIÓN SISTEMA UNIVERSIDAD ABIERTA Y EDUCACIÓN A DISTANCIA

APUNTE ELECTRÓNICO

Análisis, Diseño e Implantación de Algoritmos

Licenciatura en Informática

SUAyED UNAM
Sistema Universidad Abierta y Educación a Distancia

Plan 2012 2016 actualizado

COLABORADORES

DIRECTOR DE LA FCA
Dr. Juan Alberto Adam Siade

SECRETARIO GENERAL
Mtro. Tomás Humberto Rubio Pérez

– – – –

COORDINACIÓN GENERAL
Mtra. Gabriela Montero Montiel
Jefe de la División SUAyED-FCA-UNAM

COORDINACIÓN ACADÉMICA
Mtro. Francisco Hernández Mendoza
FCA-UNAM

– – – –

COAUTORES
L.C. Gilberto Manzano Peñaloza
Mtro. René Montesano Brand
Mtro. Luis Fernando Zúñiga López

REVISIÓN PEDAGÓGICA
Lic. Chantal Ramírez Pérez
Mayra Lilia Velasco Chacón

CORRECCIÓN DE ESTILO
L.F. Francisco Vladimir Aceves Gaytán

DISEÑO DE PORTADAS
L.CG. Ricardo Alberto Báez Caballero
Mtra. Marlene Olga Ramírez Chavero

DISEÑO EDITORIAL
Mtra. Marlene Olga Ramírez Chavero

Dr. Enrique Luis Graue Wiechers
Rector

Dr. Leonardo Lomelí Vanegas
Secretario General

Dr. Juan Alberto Adam Siade
Director

Mtro. Tomás Humberto Rubio Pérez
Secretario General

Mtra. Gabriela Montero Montiel
Jefa del Sistema Universidad Abierta
y Educación a Distancia

Análisis Diseño e Implantación de Algoritmos
Apunte electrónico

Edición: agosto de 2017.

D.R. © 2017 UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
Ciudad Universitaria, Delegación Coyoacán, C.P. 04510, México, Ciudad de México.

Facultad de Contaduría y Administración
Circuito Exterior s/n, Ciudad Universitaria
Delegación Coyoacán, C.P. 04510, México, Ciudad de México.

ISBN: En trámite
Plan de estudios 2012, actualizado 2016.

"Prohibida la reproducción total o parcial de por cualquier medio sin la autorización escrita del titular de los derechos patrimoniales"

"Reservados todos los derechos bajo las normas internacionales. Se le otorga el acceso no exclusivo y no transferible para leer el texto de esta edición electrónica en la pantalla. Puede ser reproducido con fines no lucrativos, siempre y cuando no se mutile, se cite la fuente completa y su dirección electrónica; de otra forma, se requiere la autorización escrita del titular delos derechos patrimoniales."

Hecho en México

OBJETIVO GENERAL

Al finalizar el curso, el alumno será capaz de implementar algoritmos en un lenguaje de programación.

TEMARIO OFICIAL

(64 horas)

	Horas
1. Fundamentos de algoritmos	12
2. Análisis de algoritmos	12
3. Diseño de algoritmos para la resolución de problemas	12
4. Implantación de algoritmos	12
5. evaluació de algoritmos	16
TOTAL	**64**

INTRODUCCIÓN GENERAL

Los algoritmos son una secuencia lógica y detallada de pasos para solucionar un problema. Su campo es amplio y dinámico e intervienen directamente en la vida de las organizaciones resolviendo problemas mediante programas de computadora en las distintas áreas de la empresa. Así, dada su importancia, son objeto de estudio de la asignatura Análisis, Diseño e Implantación de Algoritmos, desarrollada en cinco unidades.

En la primera unidad, se estudian los conceptos necesarios para comprender los algoritmos y sus características, así como los autómatas y los lenguajes formales utilizados; y se aborda el autómata finito determinista, conocido como máquina de Turing, y algunos ejemplos de su aplicación.

En la segunda, se exponen el análisis del problema, los problemas computables y no computables, recursividad y algoritmos de ordenación y búsqueda; y los problemas que se pueden resolver mediante la máquina de Turing (problemas computables o decidibles) y los que no se pueden solucionar de esta forma, o tarda bastante su proceso por la complejidad del algoritmo (problemas no computables). Asimismo, se examina la recursividad, es decir, la capacidad de una función de invocarse a sí misma.

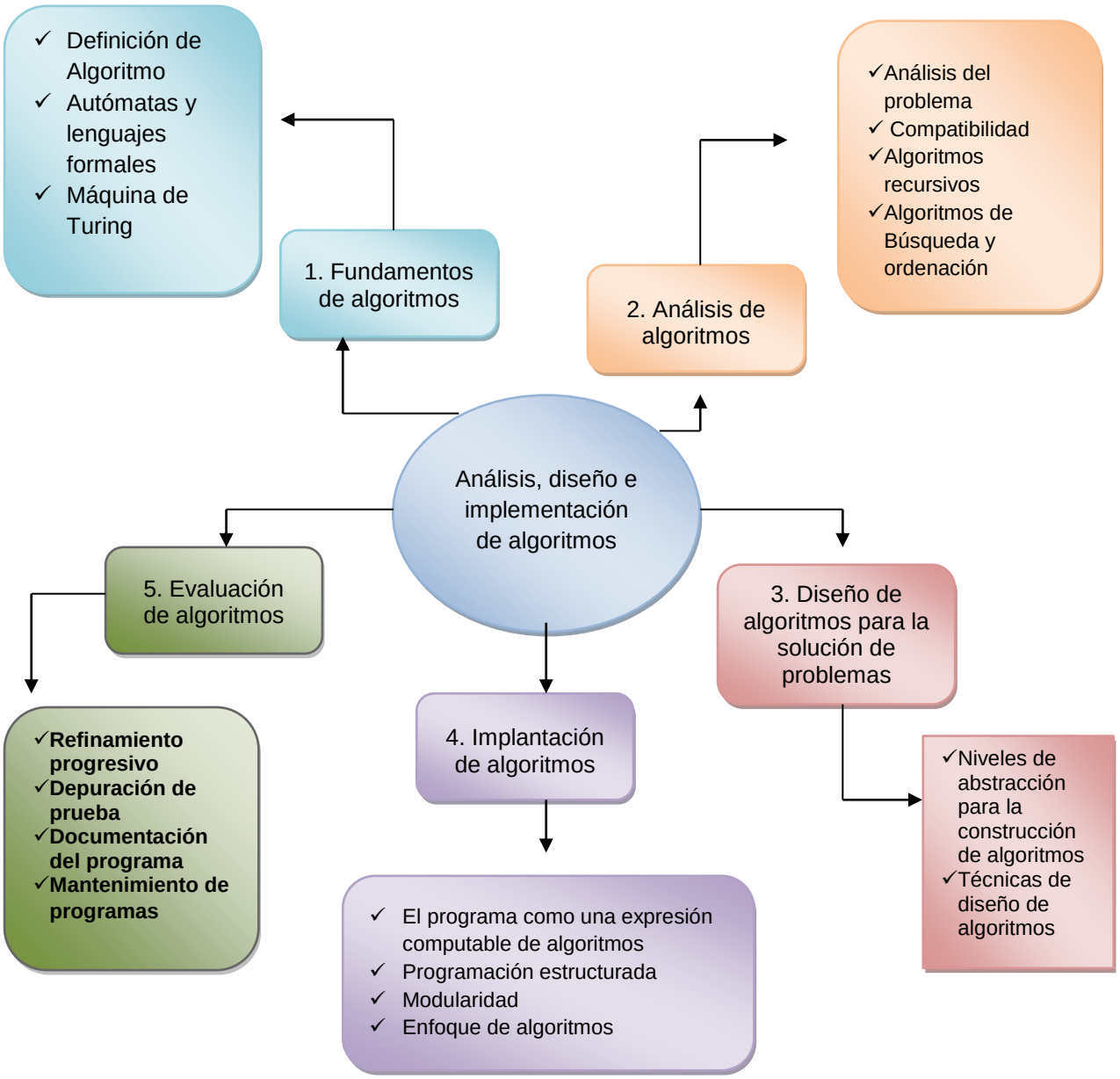
Dado que en la mayor parte de las aplicaciones empresariales se utilizan, también se analizan los algoritmos de ordenación, como burbuja, inserción, selección y rápido ordenamiento (*quick sort*); y de búsqueda, como secuencial, binaria o dicotómica, y la técnica *hash*.

En la tercera unidad, se muestra la importancia de la abstracción en la construcción de algoritmos, así como el estudio de las técnicas de diseño de algoritmos para la solución de problemas, como algoritmos voraces, programación dinámica, divide y vencerás, vuelta atrás, y ramificación y poda.

En la cuarta, se explica la manera de implementar los algoritmos mediante programas de cómputo en los que se utiliza la programación estructurada, que consiste en emplear estructuras de control como *si condición entonces sino, mientras condición hacer, hacer mientras condición, hacer hasta condición,* y *para x desde límite1 hasta límite2 hacer*. También se estudian los enfoques de diseño de algoritmos como el descendente *(top down*) y ascendente (*bottom up*); el primero conforma una solución más integral del sistema; y el segundo, aunque menos eficiente, es mucho más económico en su implantación, ya que aprovecha las aplicaciones informáticas de los distintos departamentos o áreas funcionales.

Y en la quinta, se trata el refinamiento progresivo de los algoritmos mediante la depuración y prueba de programas. Se estudian la documentación de los programas y los tipos de mantenimiento preventivo, correctivo y adaptativo.

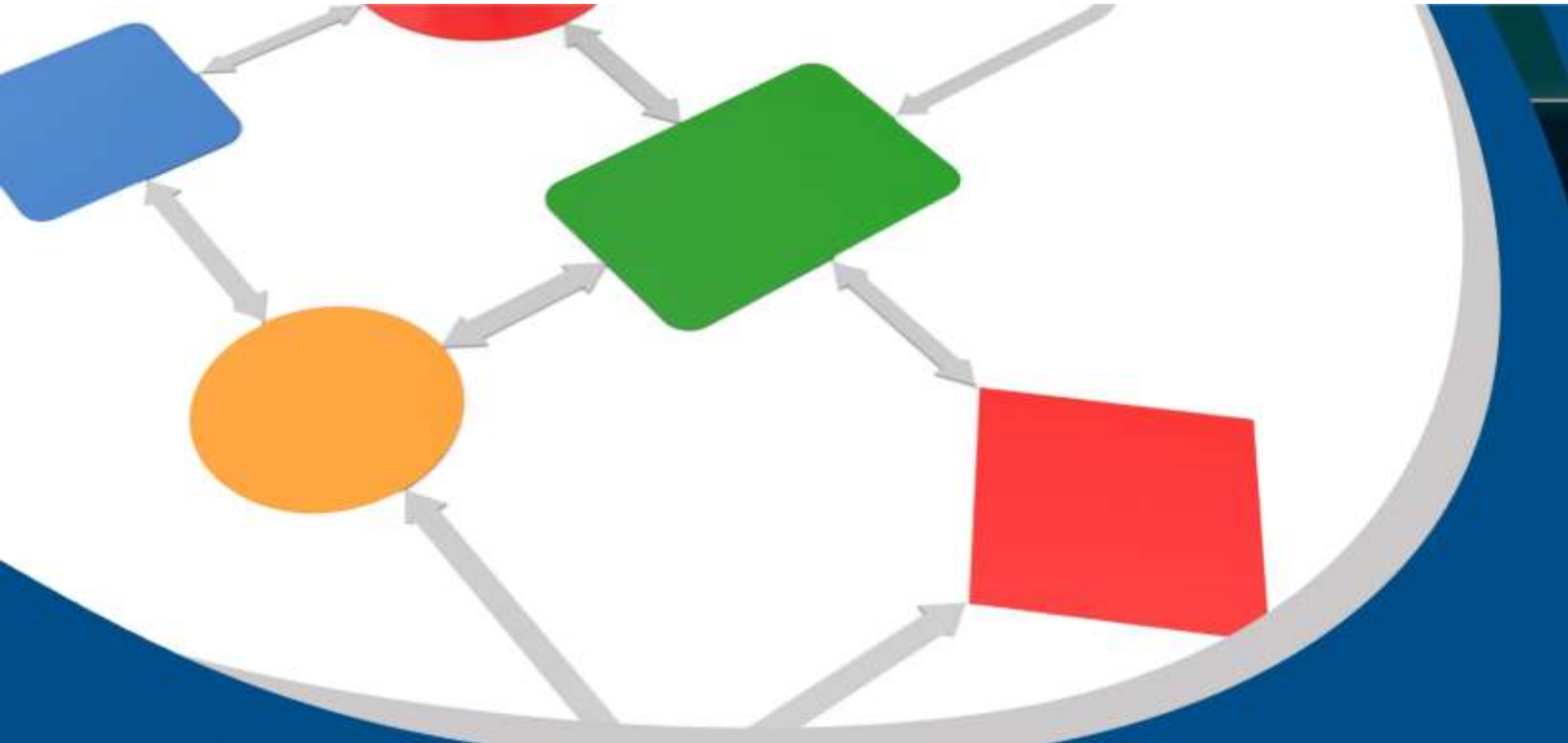
ESTRUCTURA CONCEPTUAL



Licenciatura: Informática

UNIDAD 1

Fundamentos de algoritmos



SUAyED UNAM
Sistema Universidad Abierta y Educación a Distancia

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Al finalizar la unidad, el alumno podrá identificar los componentes y propiedades de los algoritmos.

TEMARIO DETALLADO

(12horas)

1. Fundamentos de algoritmos

1.1. Definición de algoritmo
1.2. Propiedades de los algoritmos
1.3. Autómatas y lenguajes formales
1.4.Máquina de Turing

INTRODUCCIÓN

Hoy día, el algoritmo (de *Al-Khowarizmi,* sobrenombre del célebre matemático Mohamed Ben Musa) es una forma ordenada de describir los pasos para resolver problemas. Es una manera abstracta de reducir un problema a un conjunto de pasos que le den solución. Hay algoritmos muy sencillos y de gran creatividad, aunque también algunos conllevan un alto grado de complejidad.

Una aplicación de los algoritmos la tenemos en los autómatas, los cuales, basados en una condición de una situación dada, llevarán a cabo algunas acciones que ya se encuentran programadas en ellos. En este orden, será de gran utilidad involucrarse en su funcionamiento y terminología para entender que, en ese contexto de autómatas, los conceptos *alfabeto*, *frase*, *cadena vacía, lenguaje*, *gramática*, etcétera, cobran particular relevancia.

Se definirá y estudiará la máquina de Turing, un caso de autómata finito determinista, que realiza sólo una actividad en una situación dada. Y para conocer su diseño y funcionamiento, se desarrollan algunos ejemplos.

Así, el análisis de los algoritmos y los autómatas es medular para ejercitar un pensamiento lógico y abstracto al abordar los problemas de la informática.

1.1. Definición de algoritmo

Un algoritmo es un conjunto detallado y lógico de pasos para alcanzar un objetivo o resolver un problema.Por ejemplo, el instructivo para armar un modelo de avión a escala; cualquier persona, si atiende en forma estricta la secuencia de los pasos, llegará al mismo resultado.

Los pasos deben ser suficientemente detallados para que el procesador los entienda. En nuestro ejemplo, el procesador es el cerebro de quien arma el modelo; pero el ser humano tiende a obviar muchos aspectos y es factible que haga en forma automática algunos de los pasos del instructivo, sin detenerse a pensar en cómo llevarlos a cabo.

Esto sería imposible en una computadora, pues requiere de indicaciones muy puntuales para poder ejecutarlas.

Considérese, por ejemplo, si a una persona se le pide intercambiar los números 24 y 9;con cierta lógica, responderá inmediatamente “9 y 24”.

En tanto, en el procesador de una computadora se tendría que indicar de qué tipo son los datos a utilizar –para este caso, números enteros–; darle nombre a tres variables, digamos,*num1*, *num2* y *aux;* asignarle a la variable *num1* el número 24 y a *num2* el 9.

Posteriormente, señalarle que a la variable *aux* se le asigne el valor contenido en la variable *num1*; a *num1*, el contenido en la variable *num2*; y a esta última, el de la variable *aux.* Luego, se deben imprimir los valores de las variables *num1* y *num2* que exhibirán los números 9 y 24.

Se requiere, entonces, una gran cantidad de pasos para indicarle a una computadora que realice la misma tarea que un ser humano; mas es incapaz de efectuar muchas tareas aún.

Otro caso donde podemos notar la forma como el hombre obvia pasos es pidiendo a una persona que describa el proceso para preparar agua de limón. Seguro nos diría que toma una jarra de agua, le pone jugo de limón, azúcar, y listo. Para una computadora lo anterior no tendría sentido, ya que carece de unidades exactas y pasos detallados. Por tanto, los pasos deben tener mayor nivel de precisión, en esta secuencia:

- Inicio del proceso.
- Tomar una jarra de 2 litros de capacidad
- Llenar la jarra a 4/5 partes con agua natural.
- Tomar 4 limones.
- Cortar los limones por la mitad.
- Exprimir los limones dejando caer el jugo sobre el agua en la jarra.
- Tomar el recipiente del azúcar.
- Agregar 4 cucharadas soperas en la jarra con el agua.
- Revolver el agua con el jugo y el azúcar por 2 minutos.
- Fin del proceso.

En conclusión, cuando se elabora un algoritmo, se tomará en cuenta que la computadora es como un niño pequeño al cual se le está enseñando a realizar algo por primera vez; y es necesario concretar lo más posible cada paso que debe da.

1.2. Propiedades de los algoritmos

Para que un algoritmo realmente lo sea, cumplirá con las características o propiedades que se describen a continuación.

FINITO	• Dentro de la secuencia de pasos para realizar la tarea, debe tener una situación o condición que lo detenga; de lo contrario, se pueden dar ciclos infinitos que impidan llegar a un término.
PRECISO	• Un algoritmo no debe dar lugar a criterios. Por ejemplo: qué sucedería si a dos personas en distintos lugares se les ordenara preparar un pastel; suponemos que saben cómo hacerlo, y siguiendo las indicaciones de la receta, llegan a un paso en el que se indica que se agregue azúcar al gusto. Entonces, cada persona incorporaría azúcar de acuerdo con sus preferencias y el resultado no sería el mismo: los dos pasteles serían diferentes en sus características. • Concluimos que este ejemplo no es un algoritmo, puesto que existe una ambigüedad en el paso descrito.
OBTENER EL MISMO RESULTADO	• En cualquier circunstancia, si se atienden en forma estricta los pasos del algoritmo, siempre se debe llegar a un mismo resultado. Ejemplos: obtener el máximo común divisor de dos números enteros positivos, armar un modelo a escala, resolver una ecuación, etcétera**.**

Si carecen de cualquiera de estas características o propiedades, los pasos en cuestión no son algoritmo.

1.3. Autómatas y lenguajes formales[1]

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, que podían cantar y silbar.

Así, a principios del siglo XVIII, los ebanistas 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.

[1]Véase G. Manzano, *Tutorial para la asignatura Análisis, diseño e implantación de algoritmos*, México: Fondo Editorial FCA, 2003.

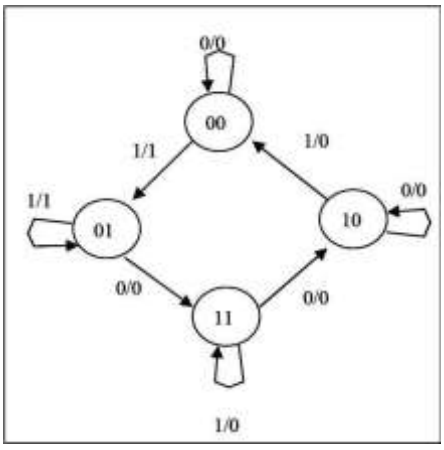
Los inventores más célebres son **Pierre Jacquet Droz, autor de**l Dibujante y los Músicos; y Jacques Vaucanson, hacedor del Pato con Aparato Digestivo, un autómata que aleteaba, parloteaba, tragaba grano y evacuaba los residuos. Este autor quiso pasar de lo banal a lo útil y sus trabajos culminaron en el telar de Joseph Marie Jaquard y la máquina de Jean Falcon, dirigida por tarjetas perforadas.

El autómata más conocido en el mundo es la máquina de Turing, elaborado por el matemático inglés Alan Mathison Turing.

En términos estrictos, 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, hay diferentes herramientas para definir el comportamiento de un autómata, entre las cuales se encuentra el diagrama de estado. En éste se pueden visualizar los estados como círculos que en su interior registran su significado, y flechas que representan la transición entre estados y la notación de entrada/salida, que provoca el cambio entre estados.



En el ejemplo se muestran cuatro estados diferentes 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 representa aquí tiene un alfabeto binario (0 y 1).

Otra herramienta de representación del comportamiento de los autómatas es la tabla de estado, que consiste en cuatro partes: descripción del estado actual, descripción de la entrada, descripción del estado siguiente y descripción de las salidas.

A continuación, se muestra la tabla 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	**0**
0	1	0	0	0	**0**
0	1	1	1	1	**1**
1	0	0	0	0	**0**
1	0	1	1	0	**1**
1	1	0	0	0	**0**
1	**1**	**1**	**1**	**0**	**1**

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, se deduce 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*:

$$y = A' x$$

ALFABETO

Un alfabeto es el conjunto de todos los símbolos válidos o posibles para una aplicación. En el campo de los autómatas, 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 empleado en el ejemplo del punto anterior, donde sólo se aplican los símbolos *1* y *0* para representar cualquier expresión de entrada, salida y estado.

FRASE

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

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

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 sin caracteres asociados.

Este tipo de cadenas no siempre implica 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.

LENGUAJE

Un lenguaje es un conjunto de cadenas que obedecen a un alfabeto fijado. Y, entendido como un conjunto de entradas, puede o no ser resuelto por un algoritmo.

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.

Una forma de representar las gramáticas es la Bakus-Naur (BNF), 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 representa como:

<regla> = sentencias y frases. *

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 los conceptos de símbolo terminal y símbolo no-terminal. El primero es una palabra del vocabulario definido en un lenguaje, por ejemplo, **ARCHIVO**, **EDICION**, etc.

Y el segundo, es 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 casos 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 observar que la regla <peticion> está formada sólo por símbolos no-terminales. Otra propiedad que nos permite visualizar el ejemplo es la definición de frases y palabras opcionales, es decir, si analizamos la regla de producción <peticion>, detectamos 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 para simplificar el significado de <peticion> es 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 se pierda la validez de la <peticion>.

LENGUAJE FORMAL

De lo anterior podemos concluir 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.

JERARQUIZACIÓN DE GRAMÁTICAS

Las gramáticas pueden ser diversas, de acuerdo con las características que rigen la formulación de reglas de producción válidas, todas los cuales parten de gramáticas arbitrarias, o sea, las que consideran la existencia infinita de cadenas formadas por los símbolos del lenguaje.

Los principales tipos de gramáticas se presentan a continuación.

GRAMÁTICAS SENSIBLES AL CONTEXTO

- En este tipo de gramáticas, el lado derecho de la regla de producción siempre debe ser igual o mayor que el lado izquierdo.

GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO

- Cumplen con las propiedades de la gramática sensible al contexto, y se distinguen porque el lado izquierdo de la regla de producción sólo debe tener un elemento, y éste no puede ser terminal.

GRAMÁTICAS REGULARES

- Cumplen con las características de la gramática independiente del contexto y, además, se restringen a través de las reglas de producción para generar sólo reglas de los dos tipos anteriores.

PROPIEDADES DE INDECIDIBILIDAD

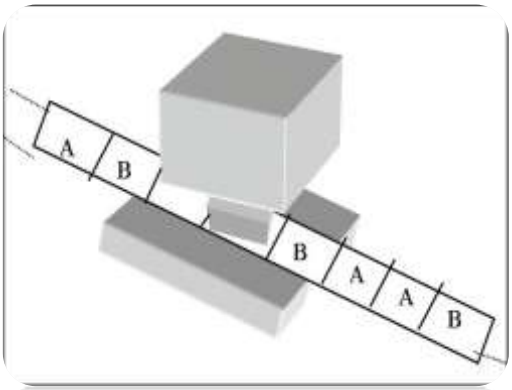
- Se dice que un lenguaje es indecidible si sus miembros no pueden ser identificados por un algoritmo que restrinja todas las entradas en un número de pasos finito. Otra de sus propiedades es que no es reconocido como una entrada válida en una máquina de Turing.
  Asociados a esta modalidad de lenguaje existen, de la misma manera, problemas indecidibles: aquellos que no pueden ser resueltos, con todas sus variantes, por un algoritmo.
- En contraposición con el lenguaje indecidible y los problemas vinculados a este tipo de lenguajes, existen los problemas decidibles, relacionados con lenguajes del mismo tipo y que tienen las características opuestas.
  Esta clase de lenguajes se conocen también como *recursivos* o *totalmente decidibles*.

1.4. Máquina de Turing

El concepto de algoritmo como un conjunto de pasos lógicos y secuenciales para solucionar un problema fue implementado en 1936 por Alan Turing, matemático inglés, en la máquina de Turing. Ésta se integra de tres elementos: cinta, cabeza de lectura-escritura y programa. La cinta tiene la propiedad de ser infinita (no acotada por sus extremos) y estar dividida en segmentos del mismo tamaño, los cuales almacenan cualquier símbolo o estar vacíos. Asimismo, puede interpretarse como el dispositivo de almacenamiento.

La cabeza de lectura-escritura es el dispositivo que lee y escribe en la cinta. Actúa en un segmento y ejecuta sólo una operación a la vez. También tiene un número finito de estados que cambian de acuerdo con la entrada y las instrucciones definidas en el programa que lo controla.

El último elemento, el programa, es un conjunto de instrucciones que controlan los movimientos de la cabeza de lectura-escritura, indicándole hacia dónde ha de 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, pues auxilia en la definición de lo que una computadora puede o no puede hacer.

Máquina de Turing como función

La máquina de Turing es una función cuyo dominio se encuentra en la cinta infinita, donde se plasma su co-dominio, esto es: todos los posibles valores de entrada se hallanen la cinta y todos los resultados de su operación se expresan también ahí.

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

Hay problemas que pueden resolverse mediante una máquina de Turing, y otros no. Los primeros se llaman *computables*; y los segundos, *no computables* o *indecidibles*. De ellos derivan, respectivamente, los procesos computables y los no computables.

Proceso computable

Puede implementarse en un algoritmo o máquina de Turing; definirse en un lenguaje decidible; e implementarse como el programa de la máquina de Turing.

Proceso no computable

No puede implementarse con una máquina de Turing por no tener solución para todas sus posibles entradas. Se especifica en un lenguaje indecidible imposible de ser interpretado por una máquina de Turing.

Un ejemplo de la máquina de Turing lo tenemos en la enumeración de binarios, como se muestra a continuación.

Ejemplo 1

Diseñar una máquina de Turing que enumere los códigos binarios de la siguiente forma:

0, 1,10, 11, 110, 111,1110,...

Solución:

Se define la máquina mediante:

Q = {q1} (Conjunto de estados)

∑ = {0, 1} (Alfabeto de salida)

Γ = {0, b} (Alfabeto de entrada)

s = q1 (Estado inicial)

Y ∂ dado por las siguientes instrucciones:

$$\partial (q1, 0) = (q1, 1, D)$$

Que se lee como: si se encuentra en estado *q1* y lee un *cero,* entonces cambia a estado *q1*, escribe *uno* y desplazar a la *derecha.*

$$\partial (q1, b) = (q1, 0, \text{Sin Desplazamiento})$$

Que se lee como: si se encuentra en estado *q1* y lee una *cadena vacía,* entonces cambia a estado *q1*, escribe un *cero* y *no hay desplazamiento*.

Si en esta máquina de Turing se comienza con la cabeza de lectura-escritura sobre el 0, tenemos la secuencia:

$$(q1, \underline{0}b) \vdash (q1, 1\underline{b}) \vdash (q1, 1\underline{0}b) \vdash (q1, 11\underline{b}) \vdash (q1, 11\underline{0}b) \vdash$$

NOTA: el carácter subrayado indica que la cabeza lectora/grabadora de lamáquina de Turing está posicionada sobre ese carácter.

Ejemplo 2

Diseñar una máquina de Turing que acepte el lenguaje L= $\{a^n b^m | \ n \text{ y } m >= 1\}$ por medio de la eliminación de las aes y bes que están en los extremos opuestos de la cadena. Es decir, usando *c* y *d,* la cadena *aaabbb* primero sería transformada en *caabbd*; después, en *ccabdd*;y por último, en *cccddd*.

Solución: Consideremos lamáquina de Turing definida mediante:

Q = {q1, q2, q3, q4, q5} (Conjunto de estados)

∑ = {a, b, c, d} (Alfabeto de salida)

Γ = {a, b, ß} (Alfabeto de entrada)

F = {q4} (Estado final)

s = {q1} (Estado inicial)

Y ∂ dado por las siguientes instrucciones:

∂ (q1, a) = (q2, c, D)
∂ (q1, b) = (q2, c, D)
∂ (q1, c) = (q4, d, ALTO)
∂ (q1, d) = (q4, d, ALTO)
∂ (q2, a) = (q2, a, D)
∂ (q2, b) = (q2, b, D)
∂ (q2, ß) = (q5, ß, I)
∂ (q2, d) = (q5, d, I)
∂ (q3, a) = (q3, a, I)
∂ (q3, b) = (q3, b, I)
∂ (q3, c) = (q1, c, D)
∂ (q5, a) = (q3, a, I)
∂ (q5, b) = (q3, d, I)
∂ (q5, c) = (q4, c, ALTO)

Si en esta máquina de Turing se comienza con la cabeza lectora-escritora sobre la primera de la izquierda, se tiene esta secuencia de movimientos:

(q1, aaabbb) ⊢ (q2, caabbb) ⊢ (q2, caabbb) ⊢ (q2, caabbb) ⊢ (q2, caabbb) ⊢
(q2, caabbb) ⊢ (q2, caabbb ß) ⊢ (q5, caabbb ß) ⊢ (q3, caabbdß) ⊢ (q3, caabbd) ⊢
(q3, caabbd) ⊢ (q3, caabbd) ⊢ (q3, caabbd) ⊢ (q1, caabbd) ⊢ (q2, ccabbd) ⊢
(q2, ccabbd) ⊢ (q2, ccabbd) ⊢ (q2, ccabbd) ⊢ (q5, ccabbd) ⊢ (q3, ccabdd) ⊢
(q3, ccabdd) ⊢ (q3, ccabdd) ⊢ (q1, ccabdd) ⊢ (q2, cccbdd) ⊢ (q2, cccbdd) ⊢
(q5, cccbdd) ⊢ (q3, ccddd) ⊢ (q1, cccddd) ⊢ (q4, cccddd)
ALTO.

Con lo anterior queda ejemplificado el diseño de unamáquina de Turing y su desarrollo.

RESUMEN DE LA UNIDAD

Se presentaron conceptos y principios básicos de los algoritmos, sus características y terminología, para aplicarlos en la resolución de problemas (razón de ser de los algoritmos). Con el apoyo de ejemplos, se generó una mejor comprensión de los puntos tratados, ya que los algoritmos pueden ser muy sencillos o muy complejos.

Además, se abordó el tema de los autómatas, una aplicación de los algoritmos. Los autómatas, basados en una condición de una situación dada, llevarán a cabo acciones que ya se encuentran programadas. En particular, se definió y estudió la máquina de Turing, un ejemplo de los autómatas finitos deterministas que realizan sólo una actividad en una situación dada.

BIBLIOGRAFÍA DE LA UNIDAD

SUGERIDA

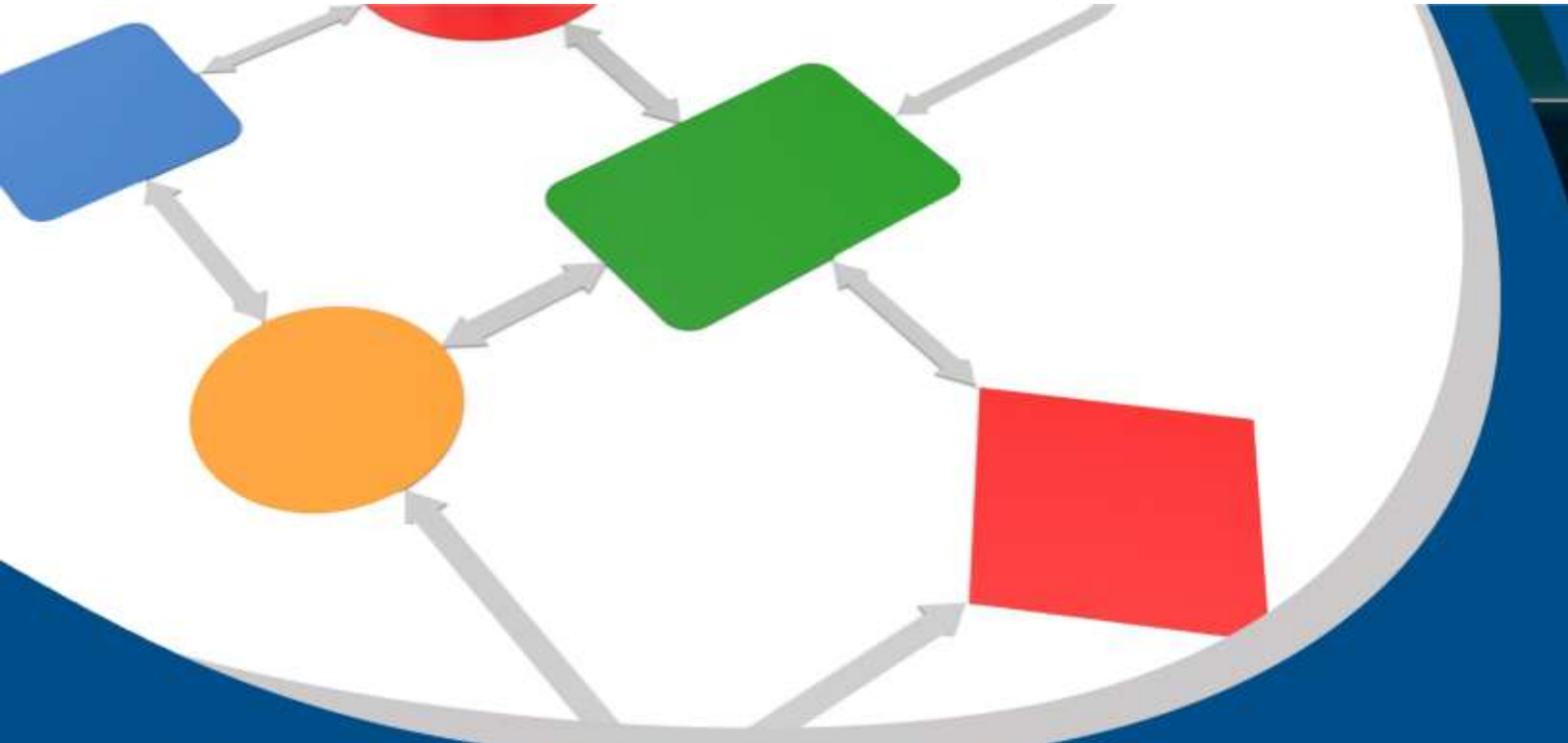
Autor	Capítulo	Páginas
Du y Ker-I Ko (2012)	7. Linear Programming	245-295

Du, Ding-Zhu; Ker-I, Ko & Xiaodong, Hu. (2012). "Restriction". En: *Design and analysis of approximation algorithms*. NY: Springer.

Licenciatura: Informática

UNIDAD 2

Análisis de algoritmos



SUAyED UNAM
Sistema Universidad Abierta y Educación a Distancia

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Podrá analizar un problema determinado y buscar una solución a partir de un algoritmo.

TEMARIO DETALLADO

(12 horas)

2. Análisis de algoritmos

2.1. Análisis del problema

2.2. Computabilidad

2.3. Algoritmos cotidianos

2.4. Algoritmos recursivos

2.5. Algoritmos de búsqueda y ordenación

INTRODUCCIÓN

En este tema, se realiza una descripción de la etapa de análisis para recabar la información necesaria que indique una acción para la solución de un problema, y se calcula el rendimiento del algoritmo considerando la cantidad de datos a procesar y el tiempo que tarde su procesamiento.

Se aborda, además, la computabilidad como solución de problemas a través del algoritmo de la máquina de Turing, de modo que se pueda interpretar un fenómeno a través de un cúmulo de reglas establecidas. Y se utiliza la construcción de modelos para abstraer una expresión a sus características más sobresalientes que sirvan al objetivo del modelo mismo.

También se tratan los problemas decidibles, que pueden resolverse por un conjunto finito de pasos con una variedad de entradas.

Otro punto a abarcar es la recursividad, propiedad de una función de invocarse repetidamente a sí misma hasta encontrar un caso base que le asigne un resultado y retorne esta solución hasta la función que la invocó. La recursión, entonces, puede definirse a través de la inducción.

La solución recursiva implica la abstracción, pero dificulta la comprensión de su funcionamiento. Su complejidad puede calcularse a partir de una función y elevarse al número de veces que la función recursiva se llame a sí misma. Por último, se estudian los diferentes métodos de ordenación y búsqueda, de uso frecuente en la solución de problemas de negocios, por lo que se hace indispensable su comprensión. Ordenar es organizar un conjunto de datos de manera que faciliten la tarea del usuario de la información, en su búsqueda y acceso a un elemento determinado.

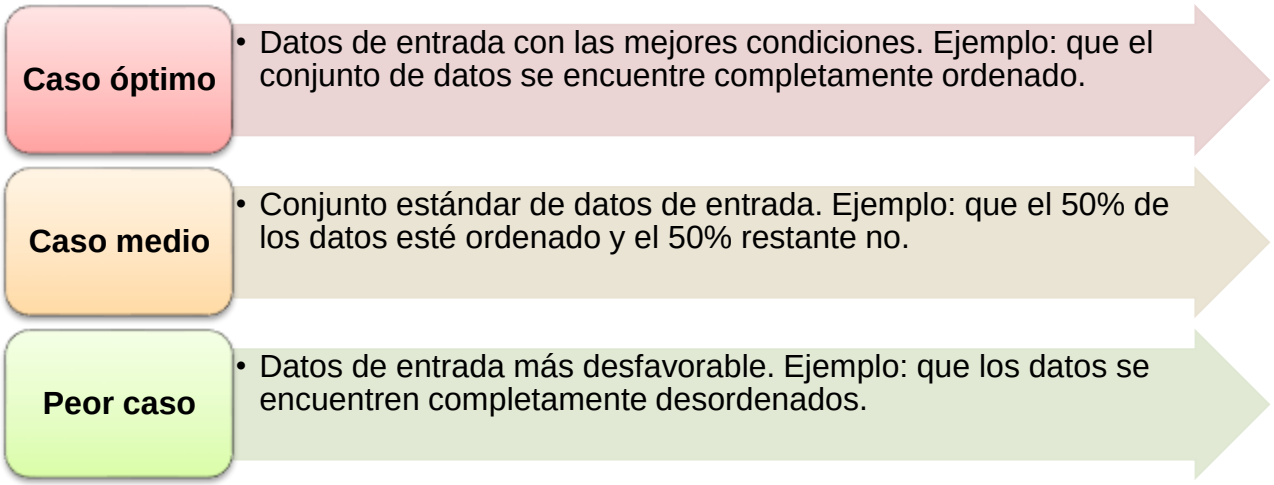
2.1. Análisis del problema

El análisis del problema es un proceso para recabar la información necesaria para emprender una acción que lo solucione.

Diversos problemas requieren algoritmos diferentes. Un problema puede llegar a tener más de un algoritmo que lo solucione, mas la dificultad se centra en saber cuál está mejor implementado, es decir, el que, dependiendo del tipo de datos a procesar, tenga un tiempo de ejecución óptimo. En este sentido, para determinar el rendimiento de un algoritmo se deben considerar dos aspectos:

- Cantidad de datos de entrada a procesar.
- Tiempo necesario de procesamiento.

El tiempo de ejecución depende del tipo de datos de entrada, clasificados en tres casos que se presentan a continuación.



Mediante el empleo de fórmulas matemáticas es posible calcular el tiempo de ejecución del algoritmo y conocer su rendimiento en cada uno de los casos ya presentados. Sin embargo, hay algunos inconvenientes para no determinar con exactitud ese rendimiento:

- Algunos algoritmos son muy sensibles a los datos de entrada, modificando cada vez su rendimiento y causando que entre ellos no sean comparables en absoluto.
- A veces, se presentan algoritmos bastante complejos, de los cuales es imposible obtener resultados matemáticos específicos.

No obstante, lo anterior en la mayoría de los casos es factible calcular el tiempo de ejecución de un algoritmo, de modo que se puede seleccionar el algoritmo con mejor rendimiento para un problema específico.

2.2. Computabilidad

Una de las funciones principales de la computación ha sido la solución de problemas a través del uso de la tecnología. Pero esto no ha logrado realizarse en la totalidad de los casos debido a la computabilidad, característica que tienen ciertos problemas de poder resolverse a través de un algoritmo, por ejemplo, una máquina de Turing.

Con base en esta propiedad, los problemas se dividen en tres categorías: irresolubles, solucionables y computables (estos últimos son un subconjunto de los segundos).

Representación de un fenómeno descrito

Todos los fenómenos de la naturaleza poseen características intrínsecas que los particularizan y diferencian unos de otros; y la percepción que se tenga de esos rasgos posibilita tanto su abstracción como su representación a partir de ciertas herramientas.

Ahora bien, la percepción de un fenómeno implica conocimiento. Y cuando se logra su representación, se dice que dicho conocimiento se convierte en transmisible. Esta representación puede realizarse utilizando diferentes técnicas de abstracción, desde una pintura hasta una función matemática; más la interpretación dada a los diferentes tipos de representación varía de acuerdo con dos elementos: la regulación de la técnica empleada y el conocimiento del receptor.

De esta manera, un receptor con ciertos 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 rango 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 se tiene un cúmulo de reglas que no permiten variedad de interpretaciones sobre una misma expresión.

Nos enfocaremos en la representación de fenómenos del segundo caso.

Modelo

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

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

Problema de decisión

Un problema de decisión es aquel cuya respuesta puede mapearse al conjunto de valores {0,1}, esto es, que tiene sólo dos posibles soluciones: sí o no. Su representación serealiza a través de una función cuyo dominio sea el conjunto citado.

Se dice que un problema es decidible cuando se resuelve 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.*

Al contrario, un problema no decidible es aquel que no puede solucionarse por un algoritmo en todos sus casos, ni su lenguaje asociado puede ser reconocido por una máquina de Turing.

2.3. Algoritmos cotidianos

Son todos aquellos algoritmos que nos ayudan a solucionar problemas de la vida cotidiana y de los cuales seguimos su metodología sin percibirlo en forma consciente, como en el siguiente ejemplo.

Algoritmo para cambiar una llanta ponchada:

Paso 1: poner el freno de mano del automóvil.

Paso 2: sacar el gato, la llave de cruz y la llanta de refacción.

Paso 3: aflojar los birlos de la llanta con la llave de cruz.

Paso 4: levantar el auto con el gato.

Paso 5: quitar los birlos y retirar la llanta desinflada.

Paso 6: colocar la llanta de refacción y colocar los birlos.

Paso 7: bajar el auto con el gato.

Paso 8: apretar los birlos con la llave de cruz.

Paso 9: guardar la llanta de refacción y la herramienta.

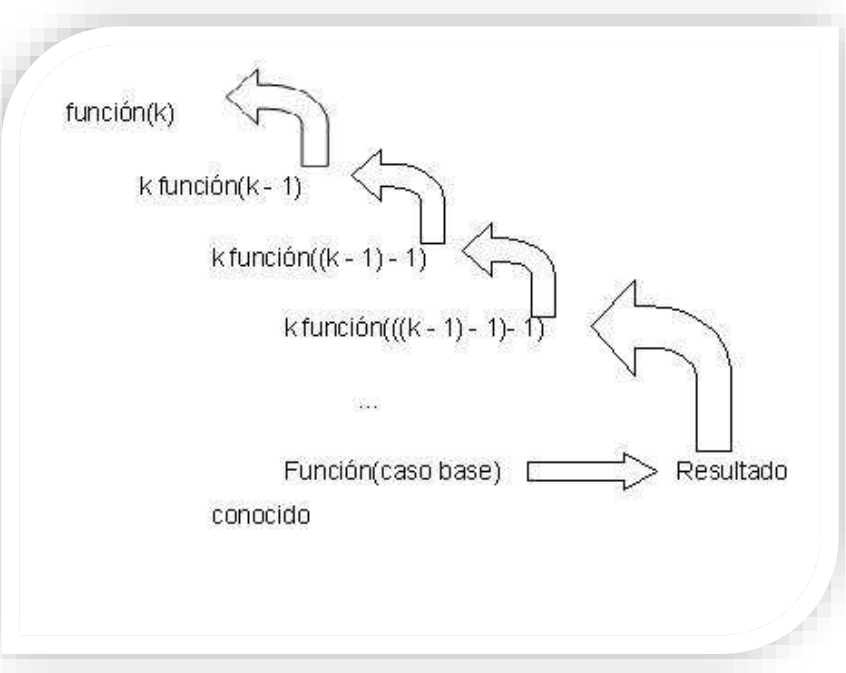
Resultado: llanta de refacción montada.

2.4. Algoritmos recursivos

Las funciones recursivas son aquellas que hacen llamadas a sí mismas en su definición, simplificando los valores originales de entrada. Se implementan cuando el problema a resolver puede simplificarse en versiones más pequeñas del mismo problema, hasta llegar a casos sencillos 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.

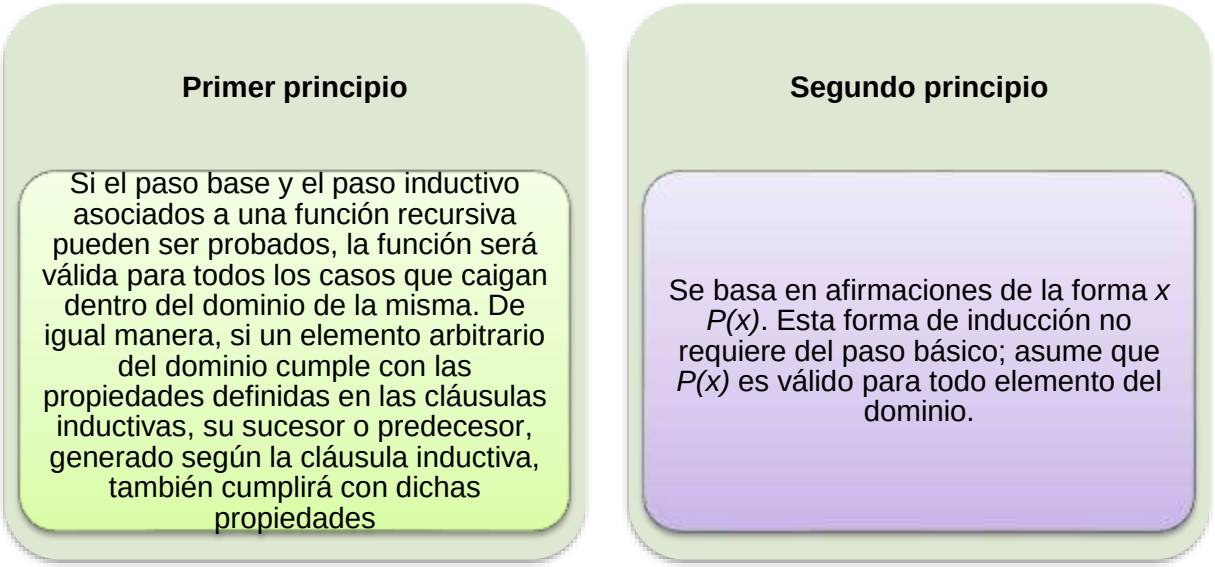
El funcionamiento de una función recursiva puede verse como:



Introducción a la inducción

La recursión se define a partir de tres elementos, uno de éstos es la cláusula de inducción matemática. A través de ella, se define 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 desde un caso base.

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



Definición de funciones recursivas

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

BÁSICA	•Especifica la semilla del dominio a partir de la cual surgirán 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*.
INDUCTIVA	•Establece la manera como 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 llega a un contradominio con propiedades análogas.
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 válida.

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*(*n0*) es válido.

Inducción

Demostrar que para cualquier entero *k n0*, si el valor generado por *P(k)* es válido, el valor generado por *P(k+1)* también lo es.

A través de la demostración de estas cláusulas, se puede certificar la validez de la función *P(n0)* para la generación de números naturales.

A continuación, se ejemplifica el empleo de la inducción matemática en una función sencilla cuyo dominio es el conjunto de los números naturales.

$$Sea\ P(n) = 1+3+5+\ldots+(2n-1) = n^2, \quad para\ cualquier\ valor\ n \in N$$

El primer valor para el cual debe ser válida la función es para cuando n = 1.

$$Si\ P(n): \sum(2n-1) = n^2\ debe\ cumplir:$$
$$P(1) = \sum((2*1)-1) = 1^2$$
$$de\ lo\ anterior\ observamos\ (2*1)-1 = 2-1 = 1\ y\ 1^2 = 1$$

La función P(n) cumple para el valor n=1. Ahora debemos suponer que la función P(n) cuando n=k (donde *k* representa cualquier número natural) también cumple; por tanto, decimos que es verdadera. De esta forma, suponemos que realizamos un proceso continuo de evaluación a partir del 1 hasta *k* realizando incrementos de 1 en 1, con lo que obtenemos:

$$P(1): 1 = 1$$
$$P(2): 1+3 = 2^2 = 4$$
$$P(3): 1+3+5 = 3^2 = 9$$
$$\vdots$$
$$P(k): 1+3+5+\cdots+(2k-1) = k^2\ que\ es\ nuestra\ hipótesis.$$

Después, se comienza con el proceso de inducción, donde se demostrará que para valores de n=k+1 (o en otras palabras, para el número siguiente de *k*) la función p(n) también es válida.

$$P(k+1): 1+3+5+\cdots+(2k-1)+(2(k+1)-1) = (k+1)^2$$

Como la función se genera a partir de una serie de sumas de elementos, podemos decir que si a nuestra función hipótesis P(k) le agregamos el elemento k+1 en ambos lados de la ecuación, debemos llenar al resultado obtenido en P(k+1), en este caso:

$$1+3+5+\cdots+(2k-1)+(2(k+1)-1) = k^2+(2(k+1)-1)$$

Entonces, observamos que tomamos la función P(k) y solamente le agregamos el elemento (2(k+1) – 1) situado en el lado izquierdo de la sumatoria de P(k+1).A continuación, desarrollando matemáticamente el lado derecho de la última ecuación, llegamos a demostrar que $k^2 + (2(k+1)-1) = (k+1)^2$

$$k^2 + \left(2\left(k+1\right)-1\right) = k^2 + 2k + 2 - 1 = k^2 + 2k + 1$$

$$como\ \left(k+1\right)^2 = k^2 + 2k + 1$$

Así, la demostración está completa y P(n) es aplicable a cualquier valor de *n*.
De este modo, la recursividad nace del proceso de incremento de valores de *n* desde 1 hasta *k*, para ir validando los resultados de la función P(n), donde la misma función es evaluada una y otra vez hasta llegar al valor deseado.

Cálculo de complejidad de una función recursiva

Generalmente, las funciones recursivas, por su funcionamiento de llamadas a sí mismas, requieren más 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.

2.5. Algoritmos de búsqueda y ordenación

Al utilizar matrices o bases de datos, las tareas que se utilizan con más frecuencia son la ordenación y la búsqueda de los datos, para las cuales existen diferentes métodos más o menos complejos, según lo rápidos o eficaces que sean.

Algoritmos de búsqueda

SECUENCIAL	BINARIA O DICOTÓMICA
• Este método de búsqueda, también conocido como *lineal*, es el más sencillo. Consiste en buscar desde el principio de un arreglo desordenado el elemento deseado, y continuar con cada uno de los elementos del arreglo hasta hallarlo, o hasta que ha llegado al final del arreglo y terminar.	• Para este tipo de búsqueda es necesario que el arreglo esté ordenado. El método consiste en dividir el arreglo por su elemento medio en dos subarreglos más pequeños, y comparar el elemento con el del centro. Si coinciden, la búsqueda termina. Cuando el elemento es menor, se busca en el primer subarreglo; y si es mayor, en el segundo.

Por ejemplo, para buscar el elemento 41 en el arreglo {23, 34, 41, 52, 67, 77, 84, 87, 93}, se realizarían los siguientes pasos:

1. Se toma el elemento central y se divide el arreglo en dos:

{23, 34, 41, 52}-67-{77, 84, 87, 93}

2. Como el elemento buscado (41) es menor que el central (67), debe estar en el primer subarreglo:

{23, 34, 41, 52}

3. Se vuelve a dividir el arreglo en dos:

{23}-34-{41, 52}

4. Como el elemento buscado es mayor que el central, debe estar en el segundo subarreglo:

{41, 52}

5. Se vuelve a dividir en dos:

{}-41-{52}

6. Como el elemento buscado coincide con el central, lo hemos encontrado. Si el subarreglo a dividir está vacío {}, el elemento no se encuentra en el arreglo y la búsqueda termina.

*Tabla*hash

Una tabla *hash* es una estructura de datos que asocia claves con valores. Su uso más frecue las operaciones de búsqueda, ya que permite el acceso a los elementos almacenados en la una clave generada.

Las tablas *hash* se aplican sobre arreglos que almacenan grandes cantidades de información; sin embargo, como utilizan posiciones pseudoaleatorias, el acceso a su contenido es bastante lento.

Función hash

La función *hash* realiza la transformación de claves (enteros o cadenas de caracteres) a números conocidos como *hash*, que contengan enteros en un rango [0..Q-1], donde Q es el número de registros que podemos manejar en memoria, los cuales se almacenan en la tabla *hash*.

La función *h(k)* debe

- ser rápida y fácil de calcular.
- minimizar las colisiones.

Hashing *multiplicativo*

Esta técnica trabaja multiplicando la clave *k* por sí misma o por una constante, usando después alguna porción de los bits del producto como una localización de la tabla *hash*.

Tiene como inconvenientes que las claves con muchos ceros se reflejarán en valores *hash* también con ceros; y que el tamaño de la tabla está restringido a ser una potencia de 2.

Para evitar las restricciones anteriores se debe calcular:

h(k) = entero [Q * Frac(C*k)]

Donde *Q* es el tamaño de la tabla; *k*, el valor a transformar; Frac, el valor de la fracción del producto a tomar y0 <= C <= 1.

Hashing *por división*

En este caso, la función se calcula simplemente como h(k) = modulo (k,Q), usando el 0 como el primer índice de la tabla *hash* de tamaño *Q*. Es importante elegir el valor de *Q* con cuidado. Por ejemplo, si *Q* fuera par, todas las claves pares serían aplicadas a localizaciones pares, lo que constituiría un sesgo muy fuerte. Una regla simple para seleccionar *Q* es tomarlo como un número primo.

Algoritmos de ordenación

Ordenar significa reagrupar o reorganizar un conjunto de datos u objetos en una secuencia específica, de forma ascendente (de menor a mayor) o descendente (de mayor a menor).La selección de uno u otro método depende de si se requiere hacer una cantidad considerable de búsquedas; es importante el factor tiempo.

Los métodos de ordenación más conocidos son burbuja, selección, inserción y rápido ordenamiento (*quick sort*), que se analizan a continuación.

Burbuja

Es el método más sencillo, pero menos eficiente. Se basa en la comparación de elementos adyacentes e intercambio de los mismos si éstos no guardan el orden deseado; se van comparando de dos en dos los elementos del vector. El elemento menor sube por el vector como las burbujas en el agua, y los elementos mayores van descendiendo por el vector.

Pasos a seguir para ordenar un vector con este método:

Paso	Descripción
1	Asigna a *n* el tamaño del vector. Si el tamaño del vector es igual a 10 elementos, entonces *n* vale 10.
2	Colocarse en la primera posición del vector. Si el número de posición del vector es igual a *n,* entonces FIN.
3	Comparar el valor de la posición actual con el valor de la siguiente posición. Si el valor de la posición actual es mayor que el de la siguiente posición, entonces intercambiar los valores.
4	Si el número de la posición actual es igual a *n* -1, entonces restar 1 a *n* y regresar al paso 2; si no, avanzar a la siguiente posición para que quede como posición actual, y regresar al paso 3.

Veámoslo con un ejemplo. Si el vector está formado por cinco enteros positivos, entonces *n* es igual a 5. Se procede como sigue

Valores											
Posición	n=5				n=4			n=3		n=2	n=1
1	7	5	5	5	5	3	3	3	2	2	1
2	5	7	3	3	3	5	2	2	3	1	2
3	3	3	7	2	2	2	5	1	1	3	3
4	2	2	2	7	1	1	1	5	5	5	5
5	1	1	1	1	7	7	7	7	7	7	7

NOTA: las celdas de color gris claro representan la posición actual; y las más oscuras, la posición siguiente, según los pasos que se van realizando del algoritmo.

Selección

En este método se hace la selección repetida del elemento menor de una lista de datos no ordenados, para colocarlo como el siguiente elemento de una lista de datos ordenados que crece.

La totalidad de la lista de elementos no ordenados debe estar disponible para que sea posible elegir el elemento con el valor mínimo en esa lista. Sin embargo, la lista ordenada podrá ser puesta en la salida, a medida que avancemos.

Los métodos de ordenación por selección se basan en dos principios:

- Seleccionar el elemento más pequeño del arreglo.
- Colocarlo en la posición más baja del arreglo

Por ejemplo, consideremos el siguiente arreglo con n=10 elementos no ordenados:

14, 03, 22, 09, 10, 14, 02, 07, 25 y 06

El primer paso de selección identifica al 2 como valor mínimo, lo saca de dicha lista y lo agrega como primer elemento a una nueva lista ordenada:

Elementos restantes no ordenados	Lista ordenada
14, 03, 22, 09, 10, 14, 07, 25, 06	02

En el segundo paso, reconoce al 3 como el siguiente elemento mínimo y lo retira de la lista para incluirlo en la nueva lista ordenada:

Elementos restantes no ordenados	Lista ordenada
14, 22, 09, 10, 14, 07, 25, 06	02, 03

Después del sexto paso, se tiene la siguiente lista:

Elementos restantes no ordenados	Lista ordenada
14, 22, 14, 25	02, 03, 06, 07, 09,10

El número de pasadas o recorridos del arreglo es n-1, pues en la última pasada se colocan los dos últimos elementos más grandes en la parte superior del arreglo.

Inserción

Consiste en insertar un elemento del vector en la parte izquierda del mismo, que ya se encuentra ordenada. Este proceso se repite desde el segundo hasta el décimo elemento.

Por ejemplo, supóngase que se desea ordenar los siguientes números del vector: 9, 3, 4, 7 y 2.

PRIMERA COMPARACIÓN
Si (valor posición 1 > valor posición 2): 9 > 3? Verdadero, intercambiar. Quedando como 3, 9, 4, 7 y 2
SEGUNDA COMPARACIÓN
Si (valor posición 2 > valor posición 3): 9 > 4? Verdadero, intercambiar. Quedando como 3, 4, 9, 7 y 2 Si (valor posición 1 > valor posición 2): 3 > 4? Falso, no intercambiar.
TERCERA COMPARACIÓN
Si (valor posición 3 > valor posición 4): 9 > 7? Verdadero, intercambiar. Quedando como 3, 4, 7, 9 y 2 Si (valor posición 2 > valor posición 3): 4 > 7? Falso, no intercambiar.

Con esta circunstancia se interrumpen las comparaciones puesto que ya no se realiza la comparación de la posición 2 con la 1: ya están ordenadas correctamente.

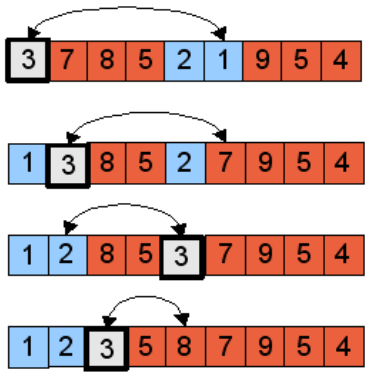
La tabla de abajo muestra las diversas secuencias de la lista de números conforme se van sucediendo las comparaciones e intercambios.

Comparación	1	2	3	4	5
1ª.	3	9	4	7	2
2ª.	3	4	9	7	2
3ª.	3	4	7	9	2
4ª.	2	3	4	7	9

Quick Sort

El algoritmo de ordenación rápida es fruto de la técnica de solución de algoritmos *divide y vencerás*, la cual se basa en la recursión: dividir el problema en subproblemas más pequeños, solucionarlos cada uno por separado (aplicando la misma técnica) y al final unir todas las soluciones.

Este método supone que se tiene *M,* el arreglo a ordenar, y *N,* el número de elementos dentro del arreglo. El ordenamiento se hace a través de un proceso iterativo. Para cada paso se escoge un elemento "a" de alguna posición específica dentro del arreglo.



Ese elemento "a" es el que se colocará en el lugar que le corresponda. Por conveniencia, se seleccionará "a" como el primer elemento del arreglo y se procederá a compararlo con el resto de los elementos del arreglo.

Una vez que se terminó de comparar "a" con todos los elementos, "a" ya se encuentra en su lugar. Asu izquierda, quedan todos los elementos menores a él; y a su derecha, todos los mayores.

Como se describe a continuación, se tienen como parámetros las posiciones del primero y último elementos del arreglo, en vez de la cantidad *N* de elementos.

Consideremos a *M* como un arreglo de *N* componentes:

Técnica

Se selecciona arbitrariamente un elemento de *M*, sea "a" dicho elemento:

$$a = M[1]$$

Los elementos restantes se arreglan de tal forma que "a" quede en la posición *j*, donde:

1. Los elementos en las posiciones M[j-1] deben ser menores o iguales que "a".
2. Los elementos en las posiciones M[j+1] deben ser mayores o iguales que "a".
3. Se toma el subarreglo izquierdo (los menores de "a") y se aplica el mismo procedimiento. Se toma el subarreglo derecho (los mayores de "a") y se efectúa el mismo procedimiento. Este proceso se realiza hasta que los subarreglos sean de un elemento (solución).
4. Al final, los subarreglos conformarán el arreglo *M,* el cual contendrá elementos ordenados en forma ascendente.

Shell

A diferencia del algoritmo de ordenación por inserción, este algoritmo intercambia elementos distantes. La velocidad del algoritmo dependerá de una secuencia de valores (*incrementos*) con los cuales trabaja, utilizándolos como distancias entre elementos a intercambiar.

Se considera la ordenación de *shell* como el algoritmo más adecuado para ordenar muchas entradas de datos (decenas de millares de elementos), ya que su velocidad, si bien no es la mejor de todos los algoritmos, es aceptable en la práctica, y su implementación (código) es relativamente sencilla.

RESUMEN DE LA UNIDAD

Es fundamental recabar la información necesaria para indicar una acción que solucione un problema en forma adecuada, puesto que permite calcular el rendimiento del algoritmo a través de la cantidad de datos a procesar y el tiempo que tarde su procesamiento.

La compresión de conceptos como computabilidad es muy importante, pues ayuda a resolver problemas mediante el algoritmo de la máquina de Turing, y ayuda a interpretar un fenómeno a través de un cúmulo de reglas establecidas.

En cuanto a la recursividad, es cuando una función se invoca repetidamente a sí misma hasta encontrar un resultado base, y éste retorna a la función que la invocó. A través de la inducción, se genera una solución recursiva que implica la abstracción, lo que dificulta la comprensión de su funcionamiento.

En la resolución de problemas a través de algoritmos, los métodos de ordenación y búsqueda se utilizan con bastante frecuencia. Ordenar los datos para su mejor manipulación facilita la tarea a los usuarios de la información, y simplifica su búsqueda y el acceso a un elemento determinado.

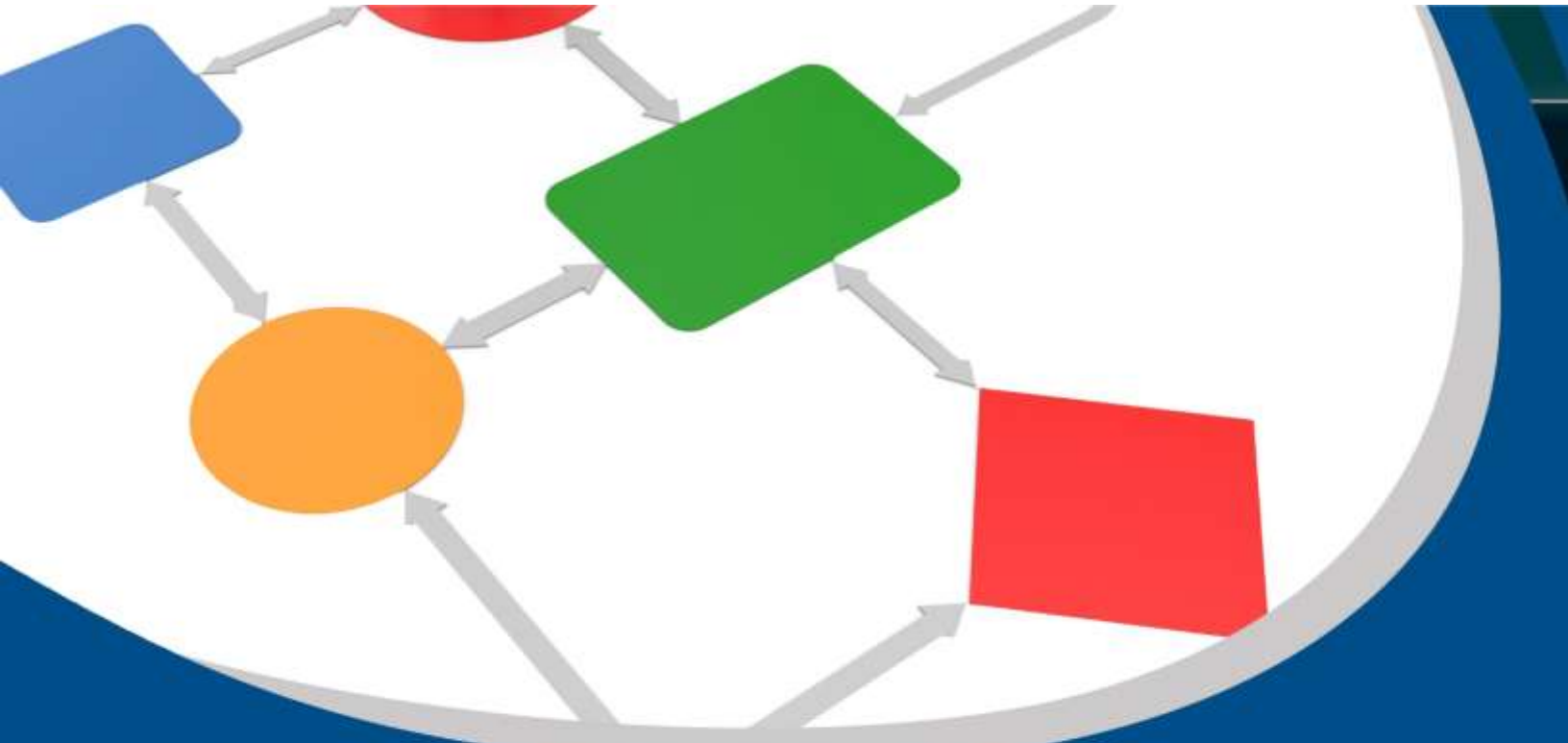
BIBLIOGRAFÍA DE LA UNIDAD

SUGERIDA

Autor	Capítulo	Páginas
Smit y Eiben (2010)	12. Using Entropy for Parameter Analysis od Evolutionary Algotithms	287-308

UNIDAD 3

Diseño de algoritmos para la solución de problemas



SUAyED UNAM
Sistema Universidad Abierta y Educación a Distancia

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Podrá plantear, desarrollar y seleccionar un algoritmo determinado para solucionar un problema.

TEMARIO DETALLADO

(12 horas)

3. Diseño de algoritmos para la solución de problemas

3.1. Niveles de abstracción para la construcción de algoritmos
3.2. Técnicas de diseño de algoritmos
3.3. Alternativas de solución
3.4. Diagramas de flujo

INTRODUCCIÓN

En esta unidad, se describe un método por medio del cual es posible construir algoritmos para la solución de problemas, además de las características de algunas estructuras básicas usadas típicamente en la implementación de estas soluciones y las técnicas de diseño de algoritmos.

En la construcción de algoritmos, se debe considerar el análisis del problema para hacer una abstracción de las características de éste, el diseño de una solución basada en modelos y, por último, a implementación del algoritmo a través de la escritura del código fuente, con la sintaxis de algún lenguaje de programación.

Todo algoritmo tiene estructuras básicas presentes en el modelado de soluciones. En este material de estudio se abordan las siguientes: ciclos, contadores, acumuladores, condicionales y rutinas recursivas.

También se analizan las diferentes técnicas de diseño de algoritmos para construir soluciones que satisfagan los requerimientos de los problemas, entre las que destacan las siguientes.

ALGORITMOS VORACES	DIVIDE Y VENCERÁS	PROGRAMACIÓN DINÁMICA
• Son utilizados para la solución de problemas de optimización; fáciles de diseñar y eficientes al encontrar una solución rápida al problema.	• Fragmentan el problema en forma recursiva y solucionan cada subproblema; y la suma de estas soluciones es la solución del problema general.	• Define subproblemas superpuestos y subestructuras óptimas; busca soluciones óptimas del problema en su conjunto.

VUELTA ATRÁS (*BACKTRACKING*)	RAMIFICACIÓN Y PODA
•Encuentra soluciones a problemas que satisfacen restricciones. Va creando todas las posibles combinaciones de elementos para obtener una solución.	•Halla soluciones parciales en un árbol en expansión de nodos, utiliza diversas estrategias (LIFO, FIFO y LC) para encontrar las soluciones y contiene una función de costo que evalúa si las soluciones identificadas mejoran la solución actual; en caso contrario, poda el árbol para no continuar buscando en esa rama. Los nodos pueden trabajar en paralelo con varias funciones a la vez, lo cual mejora su eficiencia; aunque en general requiere más recursos de memoria.

3.1. Niveles de abstracción para la construcción de algoritmos

La construcción de algoritmos se basa en la abstracción de las características del problema, a través de un proceso de análisis que permitirá seguir con el diseño de una solución fundamentada en modelos, los cuales ven su representación tangible en el proceso de implementación del algoritmo.

ANÁLISIS	• Consiste en reconocer cada una de las características del problema, lo cual se logra señalando los procesos y variables que lo rodean. Los procesos pueden identificarse como operaciones que se aplican a las variables del problema. Al analizar sus procesos o funciones, éstos deben relacionarse con sus variables. 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.
DISEÑO	• Una vez que se han analizado las causas del problema e 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.
IMPLEMETACIÓN	• Por último, ya que se tiene modelada la solución, ésta debe implementarse usando el lenguaje de programación más adecuado.

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.

Ciclos

Son estructuras que se caracterizan por iterar instrucciones en función de una condición que debe cumplirse en un momento bien definido.

Existen dos tipos de ciclos, puntualizados en el siguiente cuadro.

Mientras	Hasta que
Se caracteriza por realizar la verificación de la condición antes de ejecutar las instrucciones asociadas al ciclo.	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 MIENTRAS tiene el siguiente pseudocódigo:

```
mientras <condición> hacer
Instruccion1
Instruccion2
...
Instrucción n
fin mientras
```

El diagrama asociado a este tipo de ciclo es:

Ejemplo del uso de la instrucción *mientras*:

- n=0 (se inicializa el contador)
- suma=0 (se inicializa la variable suma)
- Mientras n=5 hacer (condición)
- Suma=suma+n
- n=n+1
- fin mientras

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

```
hacer
Instruccion1
Instruccion2
...
Instrucción n
Hasta que <condición>
```

Su diagrama se puede representar como:

Ahora, se utilizará el ejemplo anterior implementado con la función *hasta que*:

- n=0 (se inicializa el contador)
- suma=0 (se inicializa la variable suma)
- hacer
- Suma=suma+n
- n=n+1
- hasta que n=5 (condición)

Cabe mencionar que las instrucciones contenidas en la estructura *mientras* se siguen ejecutando cuando la condición resulte verdadera. En cambio, *hasta que* continuará iterando siempre que la evaluación de la condición resulte falsa.

Cuando el pseudocódigo se transforma al código fuente de un lenguaje de programación, se presenta el problema en la estructura, mientras no esté delimitada al final de ésta con un comando de algún lenguaje de programación, por lo que se tiene que cerrar con una llave, paréntesis o un *End*; en tanto, la segunda estructura está acotada por un comando tanto al inicio como al final de la misma.

Contadores

Este tipo de estructura también se caracteriza por iterar instrucciones en función de una condición que debe cumplirse en un momento conocido, y está representada por la instrucción para (*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 o decrementos en cada iteración, y suspende la ejecución de las instrucciones asociadas una vez que se alcanza el valor esperado.

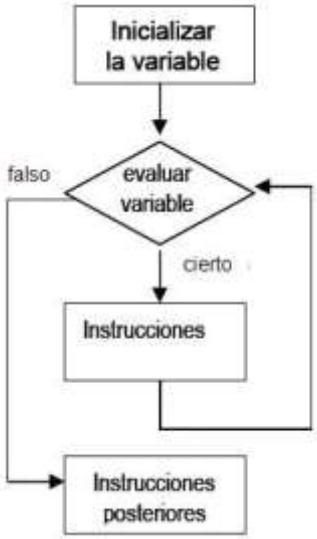
En algunos lenguajes, se puede definir el incremento que tendrá la variable; sin embargo, se recomienda que el incremento siempre sea con la unidad.

Pseudocódigo que representa la estructura:

```
Para <variable> = <valor inicial> hasta <valor
tope> [paso <incremento>] hacer
Instruccion1
Instruccion2
...
Instrucción n
Fin para <variable>
```

Se ha podido observar entre los símbolos [ ] la opción que existe para efectuar el incremento a la variable con un valor distinto de la unidad.

A continuación, se muestra el diagrama asociado a esta estructura:



Utilizaremos el ejemplo anterior implementado con la función *for*:

- suma=0 (inicializamos la variable suma)
- para n=0 hasta n=5, [n+1] hacer (indicamos que comenzaremos en n=0 y repetiremos hasta que n alcance el valor de 5, con un incremento del valor de n en 1)
- Suma=suma+n
- fin para n

Acumuladores

Los acumuladores son variables que tienen como propósito almacenar valores incrementales o decrementales a lo largo de la ejecución del algoritmo; y utilizan 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>
```

En los ejemplos anteriores podemos observar el uso de los acumuladores en la variable *n*.

Condicionales

Esta clase de estructura se utiliza para ejecutar selectivamente secciones de código de acuerdo con una condición definida. Y sólo tiene dos opciones: si la condición se cumple, se ejecuta una sección de código; si no, se ejecuta otra sección, aunque esta parte puede omitirse.

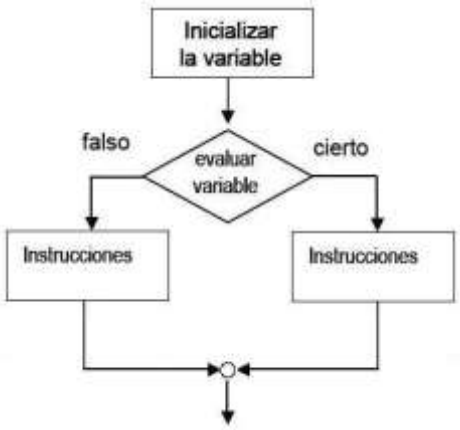
Es importante mencionar que se pueden anidar tantas condiciones como lo permita el lenguaje de programación en el que se implemente el programa.

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

```
si <condición> entonces
Instruccion1
Instrucción n
[si no
Instrucción 3
Instrucción n]
Fin si
```

Dentro del bloque de instrucciones que se definen en las opciones de la estructura, sí 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:



Ahora, se empleará la estructura *if* para solucionar el mismo ejemplo que se ha estado manejando.

- suma=0 (inicializamos la variable suma)
- n=0 (inicializamos el acumulador)
- si n=5 entonces (establecemos la condición)
- imprime suma
- de lo contrario (establecemos la alternativa de la condición)
- Suma=suma+n
- n=n+1
- fin si

En este caso, la condición se cumplirá hasta que *n* alcance el valor de 5; mientras no lo haga, realizará la suma de los valores hasta alcanzar la condición.

Rutinas recursivas

Son las que hacen llamadas a sí mismas en su definición, simplificando los valores originales de entrada. Se implementan en los casos en que el problema a 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 (SI) 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, cuando se invoca la rutina a sí misma con valores simplificados.

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

```
<valor_retorno> Nombre_Funcion
(<parámetroa> [,<parámetrob> ...])
si <caso_base> entonces
retorna <valor_retorno>
si no
Nombre_Funcion ( <parámetroa -1> [,
<parámetrob -1> ...] )
Finsi
```

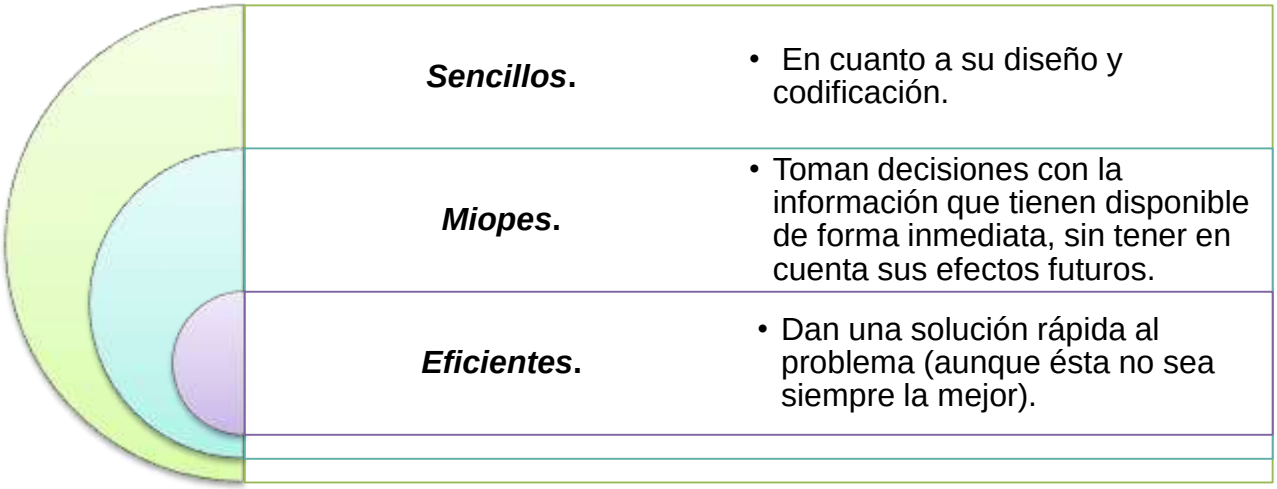
Como se observa en el ejemplo, en esta rutina es obligatoria la existencia de un valor de retorno, una estructura condicional yal menos un parámetro.

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

3.2. Técnicas de diseño de algoritmos

Algoritmos voraces

Suelen utilizarse en la solución de problemas de optimización y se distinguen porque son:



Tienen las propiedades siguientes:

- Tratan de resolver problemas de forma óptima.
- Disponen de un conjunto o lista de candidatos.

A medida que avanza el algoritmo, se acumulan dos conjuntos:

- Candidatos considerados y seleccionados.
- Candidatos considerados y rechazados.

Existe una función que comprueba si un cierto conjunto de candidatos constituye una solución del problema, ignorando si por el momento es óptima o no.

Otra función corrobora si un cierto conjunto de candidatos es factible, esto es, si es posible o no completar el conjunto añadiendo otros candidatos para obtener al menos una solución al problema. Una vez más, no nos importa si la solución es óptima o no. Normalmente se espera que al menos se obtenga una solución a partir de los candidatos disponibles inicialmente.

Hay otra función de selección que indica cuál es el más prometedor de los candidatos restantes no considerados aún.

Además, implícitamente está presente una función objetivo que da el valor a la solución que hemos hallado (valor que estamos tratando de optimizar).

Los algoritmos voraces suelen ser bastante simples. Se emplean sobre todo para resolver problemas de optimización. Por ejemplo, encontrar la secuencia óptima para procesar un conjunto de tareas por una computadora; hallar el camino mínimo de un grafo, etcétera.

Por lo regular, intervienen estos elementos:

- Un conjunto o lista de candidatos (tareas a procesar, vértices del grafo, etcétera).
- Un conjunto o lista de candidatos (tareas a procesar, vértices del grafo, etcétera).
- Una función que determina si un conjunto de candidatos es una solución al problema (aunque no tiene por qué ser la óptima).
- Una función que determina si un conjunto es completable, es decir, si añadiendo a este conjunto nuevos candidatos es posible alcanzar una solución al problema, suponiendo que ésta exista.
- Una función de selección que escoge el candidato aún no seleccionado que es más prometedor.
- Una función objetivo que da el valor/costo de una solución (tiempo total del proceso, longitud del camino, etcétera) y es la que se pretende maximizar o minimizar.

Divide y vencerás

Otra técnica común en el diseño de algoritmos es divide y vencerás, que consta de dos partes:

Dividir	***Vencer***
• Los problemas más pequeños se resuelven recursivamente (excepto, por supuesto, los casos base).	• La solución del problema original se forma, entonces, a partir de las soluciones de los subproblemas.

Las rutinas en las cuales el texto contiene al menos dos llamadas recursivas se denominan *algoritmos de divide y vencerás*; no así aquellas cuyo texto sólo comprende una.
La idea de la técnica divide y vencerás es dividir un problema en subproblemas del mismo tipo y, aproximadamente, del mismo tamaño; resolver los subproblemas recursivamente; y combinar la solución de los subproblemas para dar una solución al problema original.

La recursión finaliza cuando el problema es pequeño y la solución fácil de construir directamente.

Programación dinámica

Inventada por el matemático Richard Bellman en 1953, es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas. Una subestructura óptima significa que soluciones óptimas de subproblemas pueden ser usadas para encontrar las soluciones óptimas del problema en su conjunto.

En general, se pueden resolver problemas con subestructuras óptimas siguiendo estos pasos:

1. Dividir el problema en subproblemas más pequeños.
2. Resolver estos problemas de la mejor manera usando este proceso de tres pasos recursivamente.
3. Aplicar estas soluciones óptimas para construir una solución óptima al problema original.

Los subproblemas se resuelven, a su vez, dividiéndolos en subproblemas más pequeños, hasta alcanzar el caso fácil, donde la solución al problema es trivial.

***Vuelta atrás* (backtracking)**

El término *backtrack* fue acuñado por el matemático estadounidense D. H. Lehmer, en la década de 1950. Es una estrategia para encontrar soluciones a problemas que satisfacen restricciones.

Los problemas que deben satisfacer un determinado tipo de restricciones son completos, donde el orden de los elementos de la solución no importa. Y consisten en un conjunto o lista de variables en la que a cada una se le debe asignar un valor sujeto a las restricciones del problema.

La técnica va creando todas las combinaciones de elementos posibles para llegar a una solución. Su principal virtud es que en la mayoría de las implementaciones se pueden evitar combinaciones estableciendo funciones de acotación (o poda) y reduciendo el tiempo de ejecución.

La vuelta atrás está muy relacionada con la *búsqueda combinatoria*. La idea es encontrar la mejor combinación en un momento determinado, por eso se dice que este tipo de algoritmo es una “búsqueda en profundidad”. Si se halla una alternativa incorrecta, la búsqueda retrocede hasta el paso anterior y toma la siguiente alternativa. Cuando se han terminado las posibilidades, se vuelve a la elección anterior y se toma la siguiente opción. Si no hay más alternativas, la búsqueda falla.

Normalmente, se suele implementar este tipo de algoritmos como un procedimiento recursivo. Así, en cada llamada al procedimiento se toma una variable y se le asignan todos los valores posibles, llamando a su vez al procedimiento para cada uno de los nuevos estados.

La diferencia con la búsqueda en profundidad es que se suelen diseñar funciones de cota, de modo que no se generen algunos estados si no van a conducir a ninguna solución, o a una solución peor de la que ya se tiene. De esta forma se ahorra espacio en memoria y tiempo de ejecución.

Es una técnica de programación para hacer una búsqueda sistemática a través de todas las configuraciones posibles dentro de un espacio de búsqueda. Para lograrlo, los algoritmos de tipo *backtracking* construyen posibles soluciones candidatas de manera sistemática. En general, dada una solución candidata:

1. Verifican si *s* es solución. Si lo es, hacen algo con ella (depende del problema).
2. Construyen todas las posibles extensiones de *s* e invocan recursivamente al algoritmo con todas ellas.

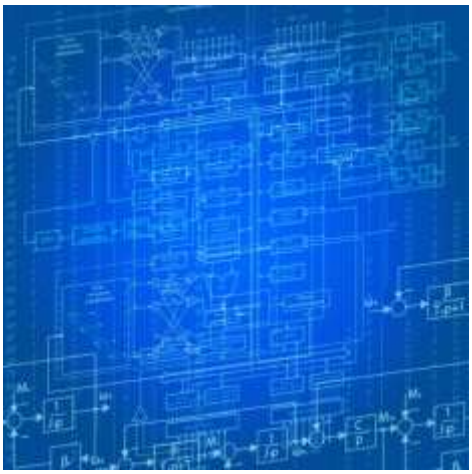
A veces, los algoritmos *backtracking* se aprovechan para encontrar una solución nada más, pero otras veces conviene que las revisen todas (por ejemplo, para ubicar la más corta).

Ramificación y poda

Esta técnica de diseño de algoritmos es similar a la de vuelta atrás y se emplea regularmente para solucionar problemas de optimización.

La técnica genera un árbol de expansión de nodos con soluciones, siguiendo distintas estrategias: recorrido de anchura (estrategia LIFO [*last input first output*]o última entrada primera salida), en profundidad (estrategia FIFO [*first input first output*] oprimera entrada primera salida), o empleando el cálculo de funciones de costo para seleccionar el nodo más prometedor.

También utiliza estrategias para las ramas del árbol que no conducen a la solución óptima: calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde éste. Si la cota muestra que cualquiera de estas soluciones no es mejor que la hallada hasta el momento, no continúa explorando esa rama del árbol, lo cual permite realizar el proceso de poda.



Se conoce como *nodo vivo del árbol* al que tiene posibilidades de ser ramificado, es decir, que no ha sido podado. Para determinar en cada momento qué nodo va a ser expandido se almacenan todos los nodos vivos en una estructura pila (LIFO) o cola (FIFO) que podamos recorrer. La estrategia de mínimo costo (LC [*low cost*]) utiliza una función de costo para decidir en cada momento qué nodo debe explorarse, con la esperanza de alcanzar pronto la solución más económica que la mejor encontrada hasta el momento.

Este proceso se da en tres etapas:

SELECCIÓN	Extrae un nodo de entre el conjunto de los nodos vivos.
RAMIFICACIÓN	Se construyen los posibles nodos hijos del nodo seleccionado en la etapa anterior.
PODA	Se eliminan algunos de los nodos creados en la etapa anterior. Los nodos no podados pasan a formar parte del conjunto de nodos vivos y se comienza de nuevo por el proceso de selección. El algoritmo finaliza cuando encuentra la solución, o bien cuando se agota el conjunto de nodos vivos**.**

Para cada nodo del árbol dispondremos de una función de costo que calcule el valor óptimo de la solución si se continúa por ese camino. No se puede realizar poda alguna hasta haber hallado alguna solución.

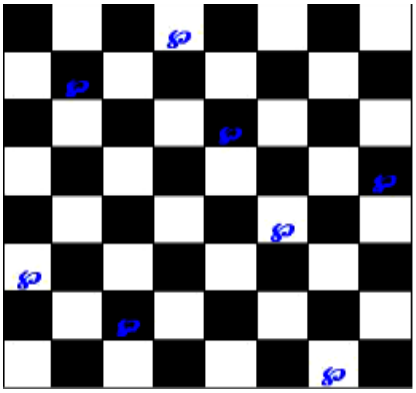
Disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución se traduce en eficiencia. La dificultad está en encontrar una buena función de costo para el problema, buena en el sentido que garantice la poda y su cálculo no sea muy costoso.

Es recomendable no realizar la poda de nodos sin antes conocer el costo de la mejor solución ubicada hasta el momento, para evitar expansiones de soluciones parciales a un costo mayor.

Estos algoritmos tienen la posibilidad de ejecutarse en paralelo. Debido a que disponen de un conjunto de nodos vivos sobre el que se efectúan las tres etapas del algoritmo antes mencionadas, se puede contar con más de un proceso trabajando sobre este conjunto, extrayendo nodos, expandiéndolos y realizando la poda.

Esto explica quelos requerimientos de memoria sean mayores que los de los algoritmos vuelta atrás. El proceso de construcción necesita que cada nodo sea autónomo en el sentido que ha de contener toda la información necesaria para realizar los procesos de bifurcación y poda, y reconstruir la solución encontrada hasta ese momento.

Un ejemplo de aplicación de los algoritmos de ramificación y poda está en el problema de las N-reinas, el cual consiste en colocar 8 reinas en un tablero de ajedrez cuyo tamaño es de 8 por 8 cuadros. Las reinas deben estar distribuidas dentro del tablero de modo que no se encuentren dos o más reinas en la misma línea horizontal, vertical o diagonal. Se han encontrado 92 soluciones posibles a este problema.



3.3. Alternativas de solución

El pseudocódigo es la técnica más usada para elaborar algoritmos. Se trata de una imitación de código. Al igual que el diagrama de flujo, va describiendo la secuencia lógica de pasos mediante enunciados que deben comenzar con un verbo que indique la acción a seguir, continuada de una breve descripción del paso en cuestión.

En caso de usar decisiones, se utilizan sentencias como:

```
si condición (relación boleana)
entonces instrucciones
si no instrucciones
finsi
```

Si es necesaria una bifurcación (cambio de flujo a otro punto del algoritmo), se emplean etiquetas como:

```
suma 2 y 5
ir a final
(instrucciones)
(instrucciones)
final (etiqueta)
```

Para conservar la sencillez, se debe usar un lenguaje llano y natural. Después que se codifique, cada frase será una línea de comando del programa.

Las órdenes más empleadas son *hacer-mientras*, *hacer-hasta*, *si-entonces-sino*, *repite-mientras.*

Por ejemplo:

```
Algoritmo: obtener la suma de los números del 1 al 100

Inicio
asigna a = 0
asigna suma = 0
mientras a <= 100

asigna suma = suma + a
fin-mientras
imprime “La suma es: “ suma
fin
```

El siguiente paso es la comprobación, y luego continúa la codificación a un programa escrito en un lenguaje de programación.

Diagrama de Nassi/Shneiderman (N/S)

El diagrama estructurado N/S, también conocido como diagrama de Chapin, es parecido a uno de flujo en el que se omiten las flechas de uniónу las cajas son contiguas. Las acciones sucesivas se pueden escribir en cajas sucesivas y, como en los diagramas de flujo, se anotan diferentes acciones en cada caja. Un algoritmo se representa en la siguiente forma:

Inicio
Accion1
Accion2
. . .
Fin

A continuación, se ejemplifica el uso de este tipo de diagrama.

Algoritmo: cálculo del salario neto a partir de las horas laboradas, el costo por hora y la tasa de impuesto del 16% sobre el salario.

Inicio
Leer Nombre,Hrs,Precio
Calcular Salario = Hrs * Precio
Calcular Imp = Salario*0.16
Calcular Neto = Salario + Imp
Escribir Nombre, Imp, SNeto
Fin

3.4. Diagramas de flujo

Los diagramas de flujo son la representación gráfica de los algoritmos. Elaborarlos implica diseñar un diagrama de bloque que contenga un bosquejo general del algoritmo, y con base en éste proceder a su ejecución con todos los detalles necesarios.

Reglas para construir diagramas de flujo:

1. Debe diagramarse de arriba hacia abajo y de izquierda a derecha. Es una buena costumbre en la diagramación que el conjunto de gráficos tenga un orden.
2. El diagrama sólo tendrá un punto de inicio y uno final. Aunque en el flujo lógico se tomen varios caminos, siempre debe existir una sola salida.
3. Usar notaciones sencillas dentro de los gráficos; y si se requieren notas adicionales, colocarlas en el gráfico de anotaciones a su lado.
4. Se deben inicializar todas las variables al principio del diagrama. Esto es muy recomendable, pues ayuda a recordar todas las variables, constantes y arreglos que van a ser utilizados en la ejecución del programa. Además, nunca sabemos cuándo otra persona modificará el diagrama y necesitará saber de estos datos.

5. Procurar no cargar demasiado una página con gráficos; si es necesario, utilizar más hojas, emplear conectores. Cuando los algoritmos son muy grandes, se pueden utilizar varias hojas para su graficación, con conectores de hoja para cada punto en donde se bifurque a otra hoja.
6. Todos los gráficos estarán conectados con flechas de flujo. Jamás debe dejarse un gráfico sin que tenga alguna salida, a excepción del que marque el final del diagrama.

Terminado el diagrama de flujo, se realiza la prueba de escritorio. Es decir, se le da un seguimiento manual al algoritmo, llevando el control de variables y resultados de impresión en forma tabular.

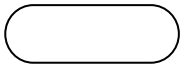
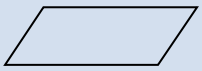
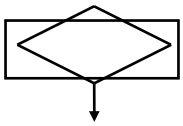
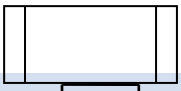
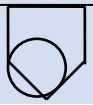
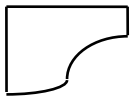
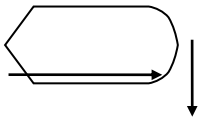
Ventajas:

1. Programas bien documentados.
2. Cada gráfico se codificará como una instrucción de un programa realizando una conversión sencilla y eficaz.
3. Facilita la depuración lógica de errores.
4. Se simplifica su análisis al facilitar la comprensión de las interrelaciones.

Desventajas:

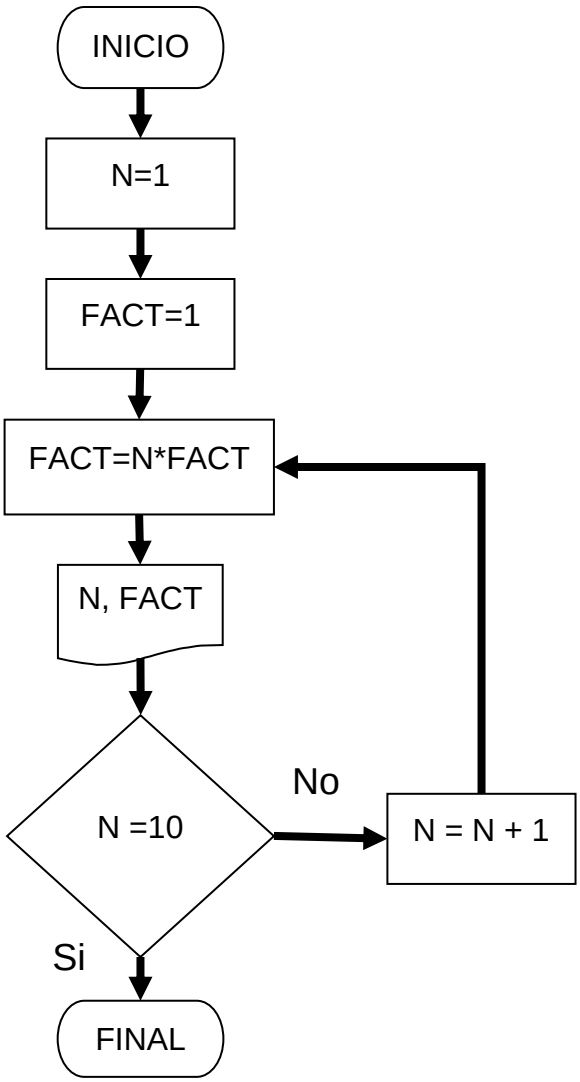
1. Su elaboración demanda varias pruebas en borrador.
2. Los programas muy grandes requieren diagramas laboriosos y complejos.
3. Falta de normatividad en su elaboración, lo que complica su desarrollo.

Algunos gráficos usados en los diagramas:

SÍMBOLO	DESCRIPCIÓN
	Terminal. Indica el inicio y el final del diagrama de flujo.
	***Entrada/Salida.*** Marca la entrada y salida de datos.
	***Proceso***. Señala la asignación de un valor en la memoria y/o la ejecución de una operación aritmética.
	***Decisión.*** Simboliza la realización de una comparación de valores.
	Proceso predefinido. Representa los subprogramas.
	***Conector de página.*** Indica la continuidad del diagrama dentro de la misma página.
	***Conector de hoja.*** Representa la continuidad del diagrama en otra página.
	***Impresión***. Indica la salida de información por impresora.
	Pantalla. Marca la salida de información en el monitor de la computadora.
	***Flujo***. Simbolizan la secuencia en que se realizan las operaciones.

A continuación, se presenta un ejemplo de diagrama de flujo.

Algoritmo: imprimir los factoriales para los números del 1 al 10.



Puede haber varias soluciones, pero una será la óptima.

RESUMEN

En esta unidad, se analizó un método por medio del cual se pueden construir algoritmos. Se revisaron las características de algunas estructuras básicas usadas típicamente en la implementación de estas soluciones, haciendo una abstracción de las características del problema basada en modelos y aterrizando en la implementación del algoritmo a través de la escritura del código fuente en un lenguaje de programación.

Las estructuras básicas de un algoritmo están presentes en el modelado de soluciones. En esta unidad, se abordaron estructuras como contadores, acumuladores, condicionales y rutinas recursivas. También se estudiaron técnicas de diseño de algoritmos para construir soluciones que satisfagan los requerimientos de los problemas, como algoritmos voraces, divide y vencerás, programación dinámica, vuelta atrás, ramificación y poda.

Además, se expuso cómo las soluciones parciales en un árbol en expansión de nodos, utilizando diversas estrategias (LIFO, FIFO y LC) para encontrar las soluciones, contienen una función de costo que evalúa si las soluciones halladas mejoran la solución actual.

Así, se ha mostrado un panorama general de la construcción de algoritmos, sus estructuras básicas y técnicas de diseño.

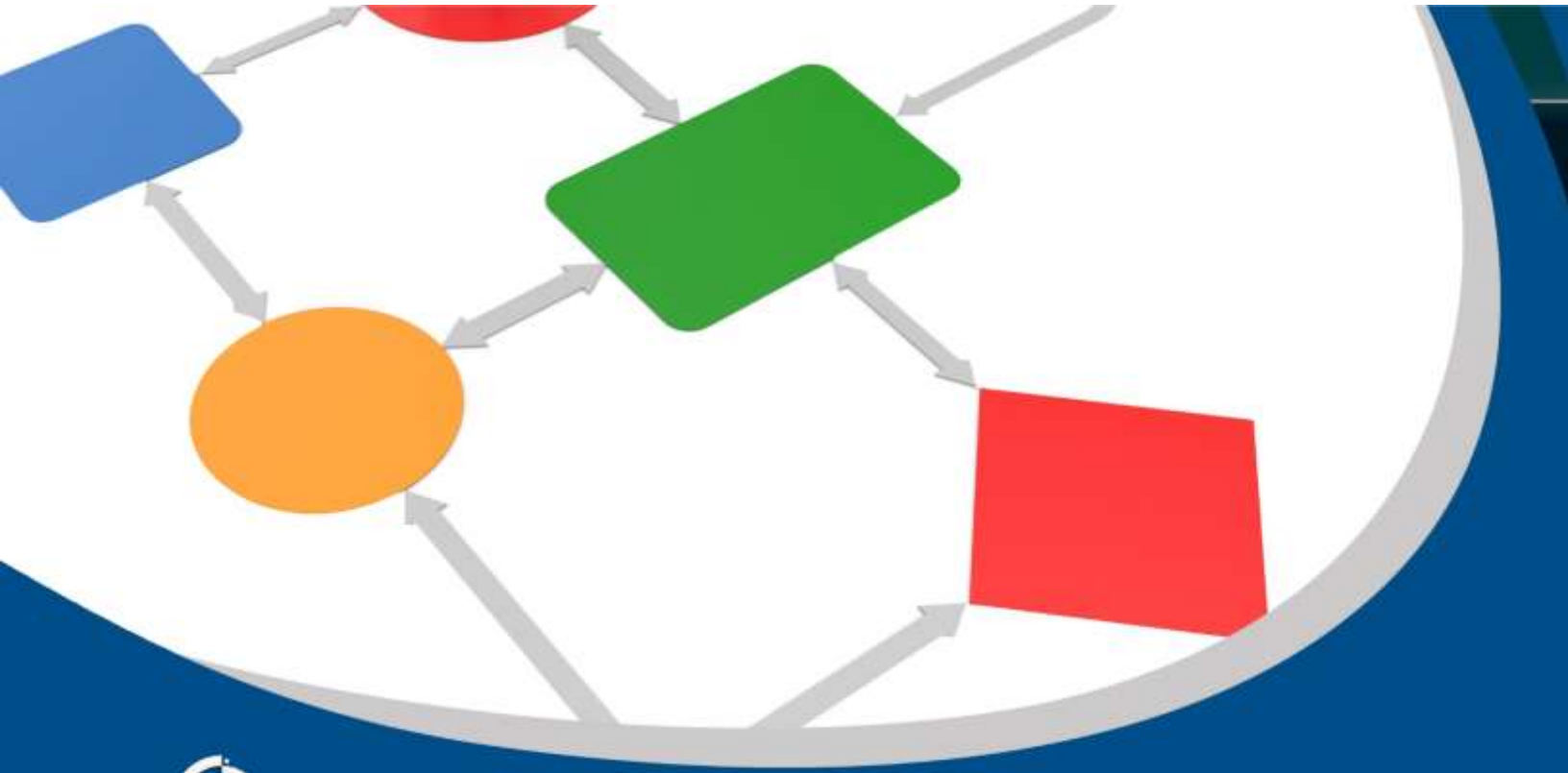
BIBLIOGRAFÍA DE LA UNIDAD

SUGERIDA

Autor	Capítulo	Páginas
Duy Ker-I Ko(2012)	2. Greedy Strategy	35-80

UNIDAD 4

Implantación de algoritmos



SUAyED UNAM
Sistema Universidad Abierta y Educación a Distancia

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Al finalizar la unidad, el alumno podrá llevar a cabo la realización de un programa a partir de un algoritmo para un problema determinado.

TEMARIO DETALLADO

(12 horas)

4. Implantación de algoritmos

4.1. El programa como una expresión computable del algoritmo
4.2. Programación estructurada
4.3. Modularidad
4.4. Funciones, rutinas y procedimientos
4.5. Enfoque de algoritmos

INTRODUCCIÓN

En esta unidad, se aborda el método para transformar un algoritmo a su expresión computable: el programa. Éste es un conjunto de instrucciones que realizan determinadas acciones y que están escritas en un lenguaje de programación. La labor de escribir programas se conoce como *programación*.

También se estudian las estructuras de control básicas a las que hace referencia el teorema de la estructura, que son piezas clave en la programación estructurada cuya principal característica es no realizar bifurcaciones lógicas a otro punto del programa (como se hacía en la programación libre), lo cual facilita su seguimiento y mantenimiento.

Asimismo se analizan los dos enfoques de diseño de sistemas: el refinamiento progresivo y el procesamiento regresivo, y se comparan sus ventajas y limitaciones.

4.1. El programa como una expresión computable del algoritmo

Como ya se ha mencionado, el algoritmo es una secuencia lógica y detallada de pasos para solucionar un problema. Una vez diseñada la solución, se implementa mediante un programa de computadora. El algoritmo debe transformarse, línea por línea, a la sintaxis utilizada por un lenguaje de programación (el que seleccione el programador).

Se revisa a continuación la manera como un algoritmo se convierte en un programa de computadora.

DEFINICIÓN DEL ALGORITMO

- Enunciado del problema para saber qué se espera que haga el programa.

ANÁLISIS DEL ALGORITMO

- Para resolver el problema, debemos estudiar las salidas que se esperan del programa para definir las entradas requeridas. También se bosquejarán los pasos a seguir por el algoritmo.

SELECCIÓN DE LA MEJOR ALTERNATIVA

- Si hay varias formas de solucionar el problema, se debe escoger la que produzca resultados en el menor tiempo y con el menor costo posible.

DISEÑO DE ALGORITMO

- Se diagraman los pasos del problema. También se puede utilizar el pseudocódigo como la descripción abstracta del problema.

PRUEBA DE ESCRITORIO

- Cargar datos muestra y seguir la lógica marcada por el diagrama o el pseudocódigo. Comprobar los resultados para verificar si hay errores.

CODIFICACIÓN

- Traducir cada gráfico del diagrama o línea del pseudocódigo a una instrucción de algún lenguaje de programación. El código fuente se guarda en archivo electrónico.

COMPILACIÓN

- El compilador verifica la sintaxis del código fuente en busca de errores, es decir, si se ha escrito mal algún comando o regla de puntuación del lenguaje. Se depura y vuelve a compilar hasta que ya no existan errores de este tipo. El compilador crea un código objeto, el cual lo enlaza con alguna librería de programas (edición de enlace) y obtiene un archivo ejecutable.

PRUEBA DEL PROGRAMA

- Se ingresan datos muestra para el análisis de los resultados. Si hay un error, se regresa al paso 6 para revisar el código fuente y depurarlo.

DOCUMENTACIÓN

- El programa libre de errores se documenta con los diagramas utilizados, listado de su código fuente, diccionario de datos en donde se listan las variables, constantes, arreglos, abreviaciones, etcétera.

Una vez que se produce el archivo ejecutable, el programa se hace independiente del lenguaje de programación que se empleó para generarlo, por lo que permite su portabilidad a otro sistema de cómputo. En resumen, el programa esla expresión computable del algoritmo ya implementado, y puede utilizarse repetidamente en el área en donde se produjo el problema.

4.2. Programación estructurada

Al construir un programa con un lenguaje de alto nivel, el control de su ejecución debe utilizar únicamente las tres estructuras de control básicas: secuencia, selección e iteración. A estos programas se les llama “estructurados”.

Teorema de la estructura

A finales de la década de 1960, surgió un nuevo teorema que indicaba que todo programa puede escribirse utilizando únicamente las tres estructuras de control siguientes.

SECUENCIA
Serie de instrucciones que se ejecutan sucesivamente.
SELECCIÓN
La instrucción condicional alternativa de la forma: *SI condición ENTONCES* *Instrucciones (si la evaluación de la condición resulta verdadera)* *SI NO* *Instrucciones (si la evaluación de la condición es falsa)* *FIN SI.*

ITERACIÓN
La estructura condicional *MIENTRAS,* que ejecuta la instrucción repetidamente siempre y cuando la condición se cumpla, o también la forma *HASTA QUE,* ejecuta la instrucción siempre que la condición sea falsa, o lo que es lo mismo, hasta que la condición se cumpla.

Estos tres tipos de estructuras lógicas de control pueden ser combinados para producir programas que manejen cualquier tarea de procesamiento de datos.

La programación estructurada se basa en el teorema de la estructura, el cual establece que cualquier programa contiene solamente las estructuras lógicas mencionadas anteriormente.

Una característica importante en un programa estructurado es que puede ser leído en secuencia, desde el comienzo hasta el final, sin perder la continuidad de la tarea que cumple. Esto es relevante, pues es mucho más simple comprender completamente el trabajo que realiza una función determinada si todas las instrucciones que influyen en su acción están físicamente cerca y encerradas por un bloque. La facilidad de lectura, de comienzo a fin, es una consecuencia de utilizar solamente tres estructuras de control y eliminar la instrucción de desvío de flujo de control (la antigua instrucción *goto etiqueta*).

Ventajas de la programación estructurada:

- Facilita el entendimiento de programas.
- Reduce el esfuerzo en las pruebas.
- Programas más sencillos y más rápidos.
- Mayor productividad del programador.
- Se facilita la utilización de otras técnicas para el mejoramiento de la productividad en programación.
- Los programas estructurados están mejor documentados.
- Un programa que es fácil de leer y está compuesto de segmentos bien definidos tiende a ser simple, rápido y menos expuesto a mantenimiento.

Estos beneficios derivan en parte del hecho que, aunque el programa tenga una extensión significativa, en documentación tiende siempre a estar al día.

El siguiente programa que imprime una secuencia de la serie de Fibonacci[2] de la forma 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 y 89, es un ejemplo de programación estructurada.

[2] En la serie de Fibonacci, el tercer número es el resultado de la suma de los dos números anteriores a éste.

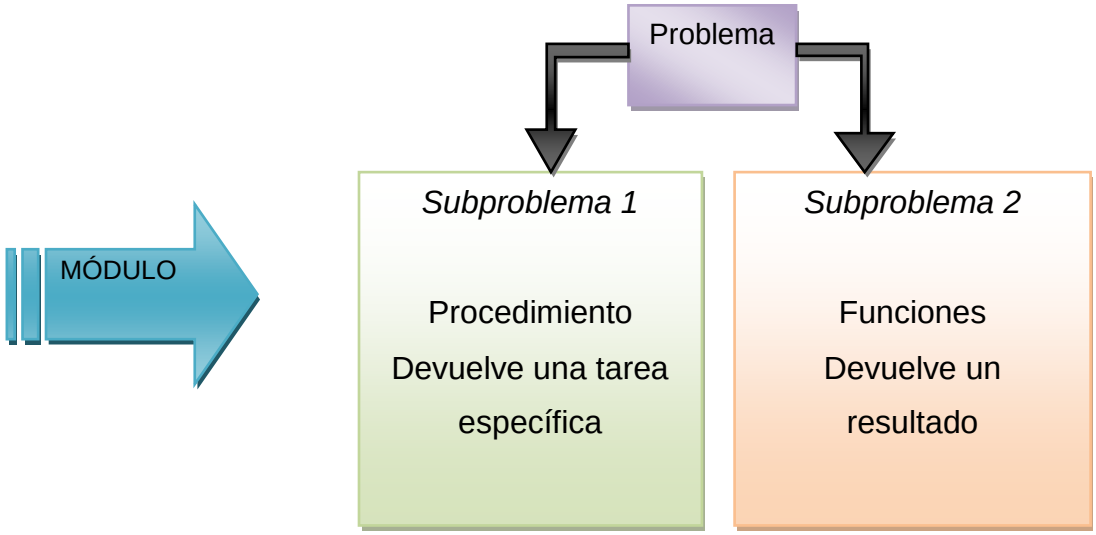
PSEUDOCÓDIGO	PROGRAMA FUENTE EN LENGUAJE C.

Inicio
entero x,y,z;
x=1;
y=1
imprimir (x,y);
mientras (x+y<100) hacer
z←x+y;
imprimir (z);
x←y;
y←z;
fin mientras
fin.
	#include <stdio.h>
	#include <conio.h>
	void main(void)
	{
	int x,y,z;
	x=1;
	y=1;
	printf("%i,%i",x,y);
	while (x+y<100) {
	z=x+y;
	printf("%i",z);
	x=y;
	y=z;
	}
	getch();
	}

Como se observa en la tabla anterior, las instrucciones del programa se realizan en secuencia, el programa contiene una estructura *mientras* que ejecuta el conjunto de instrucciones contenidas en ésta, siempre que se cumpla la condición que *x más y* sea menor que 100.

4.3. Modularidad

Un problema se puede dividir en subproblemas más sencillos o módulos. Dentro de los programas, se les conoce como *subprogramas* y presentan dos tipos: procedimientos y funciones. Ambos reciben datos del programa que los invoca, donde los primeros devuelven una tarea específica y las funciones un resultado:



En los nuevos lenguajes de programación, los procedimientos cada vez se utilizan menos; a diferencia de las funciones, de mayor aplicación. Un ejemplo de un lenguaje de programación construido únicamente por funciones es el lenguaje C.

Ahora, cuando un procedimiento o una función se invocan a sí mismos se le llama *recursividad*.

4.4. Funciones, rutinas y procedimientos

Función

Es un conjunto de pasos para realizar cálculos especificados y devolver siempre un resultado. Los pasos están almacenados con el nombre de *función*, la cual acepta ciertos valores o *argumentos* para realizar cálculos con éstos y proporcionar un resultado. Hay funciones que carecen de argumentos, pero sí arrojan un resultado.

Tanto el resultado de la función como los argumentos que recibe deben tener un tipo de dato previamente definido, por ejemplo, entero, carácter, cadena, fecha, booleano, etcétera.

Una función puede invocar a otra e inclusive tener la capacidad de invocarse a sí misma, como las funciones recursivas. La ventaja es que la función se puede implementar e invocar una y repetido número de veces.

Un ejemplo de una función sería la siguiente, que recibe un valor *n* y calcula su factorial:

```
entero función factorial (entero n)
inicio
si (n=0) entonces
factorial=1;
si no
        factorial = n* factorial(n-1);
fin si;
retorna factorial
fin función
```

Rutina

Es un algoritmo que realiza una tarea específica y que puede ser invocado desde otro algoritmo para ejecutar tareas intermedias. También como la función, recibe argumentos y retorna valores. De hecho, la rutina es un tipo muy específico de una función por lo que se puede considerar sinónimo de ésta.

Procedimiento

Es similar a una función, pero no regresa un resultado, sino que en su lugar realiza una o varias tareas, por ejemplo, centrar una cadena en la pantalla de la computadora, dibujar un marco, imprimir un mensaje etcétera.

Ejemplo:

```
Procedimiento mensaje Bienvenida
Inicio
Borrar pantalla
Imprimir “Bienvenido al sistema”
Imprimir “Teclee cualquier tecla para continuar…”
Salir
```

Como se observa, el procedimiento realiza las tareas de borrar la pantalla e imprimir un mensaje de bienvenida, mas no devuelve un valor como resultado (lo que sí ocurre en las funciones).

4.5. Enfoque de algoritmos

Existen dos enfoques que se refieren a la forma como se diseña un algoritmo: refinamiento progresivo y procesamiento regresivo.

Refinamiento progresivo

Es una técnica de análisis y diseño de algoritmos basada en la división del problema principal en problemas más simples.

Partiendo de problemas más sencillos, 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.

Esta técnica se conoce como *top-down* (arriba-abajo),y es aplicable a la optimización del desempeño y a la simplificación de un algoritmo.

Top-down

Conocida también como *diseño descendente*, consiste en establecer una serie de niveles de mayor a menor complejidad (arriba-abajo) que den solución al algoritmo.

Se efectúa una relación entre las etapas de la estructuración, de forma que una etapa jerárquica y su inmediato inferior se relacionen mediante entradas y salidas de datos.

Se integra de una serie de descomposiciones sucesivas del problema inicial, que recibe el refinamiento progresivo del repertorio de instrucciones que van a formar parte del programa.

Esta técnica tiene los siguientes objetivos:

- Simplificación del algoritmo y de los subalgoritmos de cada descomposición.
- Las diferentes partes del problema pueden ser detalladas de modo independiente e incluso por diferentes personas (división del trabajo).
- El programa final queda estructurado en forma de bloque o módulos, lo que hace más sencilla su lectura y mantenimiento (integración).
- Se alcanza el objetivo principal del diseño, ya que se parte de éste y se va descomponiendo el diseño en partes más pequeñas, pero siempre teniendo en mente dicho objetivo.

Un ejemplo de un diseño descendente está representado en este sistema de nómina. Como se puede observar, en este caso, el diseño descendente es jerárquico, el módulo 0 de nómina contendrá el menú principal que integrará al sistema, controlando desde éste los submenús del siguiente nivel.

- El módulo 1 de empleados comprenderá un submenú con las opciones de altas, bajas y los cambios a los registros de los empleados.
- En el módulo 2, se capturarán los movimientos quincenales de la nómina como los días trabajados, horas extra, faltas, incapacidades de los empleados, etcétera.
- En el módulo 3, se realizarán los cálculos de las percepciones, deducciones y el total de la nómina, individualizado por trabajador.
- El menú de reportes con el número 4 comprenderá los subprogramas para consultar en pantalla e imprimir los recibos de nómina, la nómina misma y un catálogo de empleados, aunque no es limitativo, puesto que se le pueden incluir más reportes o informes al sistema reportes correspondientes. Así, teniéndolo en mente, se fue descomponiendo en los distintos módulos y submódulos que conforman al sistema.

Procesamiento regresivo

Es otra técnica de análisis y diseño de algoritmos. 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 ciertos rasgos específicos de las entradas.

Esta técnica también es conocida como *bottom-up* (abajo-arriba), aunque suele pasar que no alcance la integración óptima y eficiente de las soluciones de los diversos problemas.

Bottom-up

Es el diseño ascendente referido a la identificación de aquellos subalgoritmos que necesitan computarizarse conforme vayan apareciendo, su análisis y su codificación, para satisfacer el problema inmediato.

Cuando la programación se realiza internamente y con enfoque ascendente, es difícil llegar a integrar los subalgoritmos a tal grado que el desempeño global sea fluido. Los problemas de integración entre los subalgoritmos no se solucionan hasta que la programación alcanza la fecha límite para la integración total del programa.

Aunque cada subalgoritmo parece ofrecer lo que se requiere, cuando se considera el programa final, éste presenta ciertas limitaciones por haber tomado un enfoque ascendente:

- Hay duplicación de esfuerzos al introducir los datos.
- Se incorporan al sistema muchos datos carentes de valor.
- El objetivo del algoritmo no fue completamente considerado y, en consecuencia, no se satisface plenamente.
- A diferencia del diseño descendente, en donde sí se alcanza la integración óptima de todos los módulos del sistema que lo conforman, en el diseño ascendente no se llega a este grado de integración, por lo que muchas tareas tendrán que llevarse a cabo fuera del sistema con el consiguiente retraso de tiempo, redundancia de información, mayor posibilidad de errores, etcétera.

El beneficio del diseño ascendente es que su desarrollo es mucho más económico que el descendente, pero habría que ponderar la bondad de esta ventaja comparada con la eficiencia en la obtención de los resultados que ofrezca el sistema ya terminado.

RESUMEN

En esta unidad, se estudió el programa, entendido como un conjunto de instrucciones que realizan acciones específicas escritas en un lenguaje de programación. De aquí la importancia de analizar el método para transformar un algoritmo a su expresión computable, el programa.

Se abordaron, además, las estructuras de control básicas referidas al teorema de la estructura, fundamentales en la programación estructurada que, a diferencia de la programación libre, no realiza bifurcaciones lógicas a otro punto, lo cual facilita su seguimiento y mantenimiento.

BIBLIOGRAFÍA DE LA UNIDAD

SUGERIDA

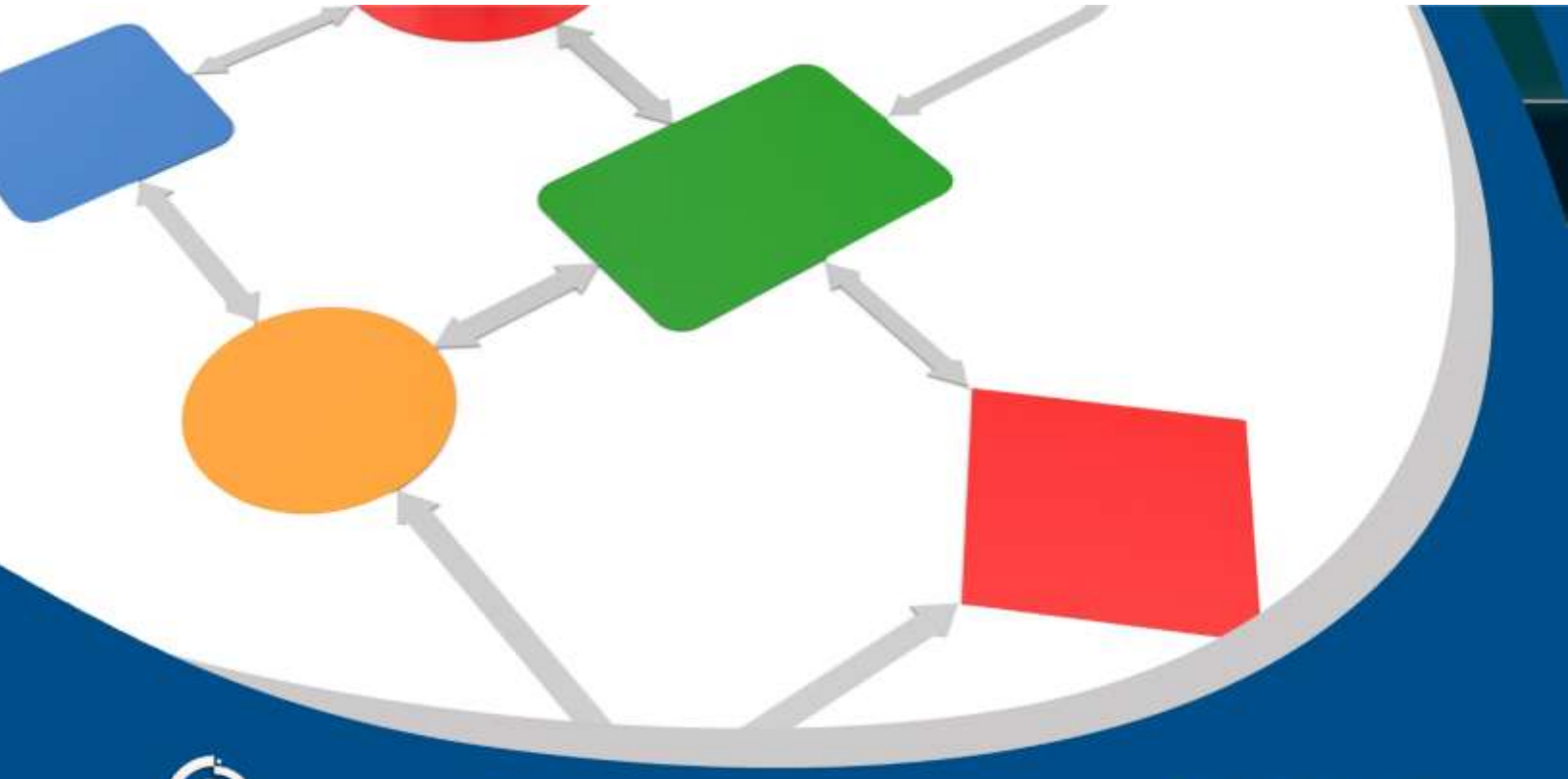
Autor	Capítulo	Páginas
Gaglioloy Lengrand (2010)	7	161-184

Gagliolo, Matteo y Legrand, Catherine. "Algorithm Survival Analysis". En: Thomas Bartz-Beielstein. (2010). *Experimental methods for the analysis of optimization algorithms*. Berlín: Springer.

Licenciatura: Informática

UNIDAD 5

Evaluación de algoritmos



SUAyED UNAM
Sistema Universidad Abierta y Educación a Distancia

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Podrá identificar el algoritmo que solucione más eficientemente al problema en cuestión, documentarlo en futuras revisiones y llevar a efecto el mantenimiento preventivo, correctivo y adaptativo para su óptima operación.

TEMARIO DETALLADO

(16 horas)

5. Evaluación de algoritmos

5.1.Refinamiento progresivo
5.2. Depuración y prueba
5.3. Documentación del programa
5.4. Mantenimiento de programas

INTRODUCCIÓN

La evaluación de algoritmos es un proceso de análisis de su desempeño en el tiempo de ejecución que tardan para encontrar una solución, y la cantidad de recursos empleados para ello. Entre las técnicas más confiables para esto se encuentran aquellas que miden la complejidad de algoritmos a través de funciones matemáticas.

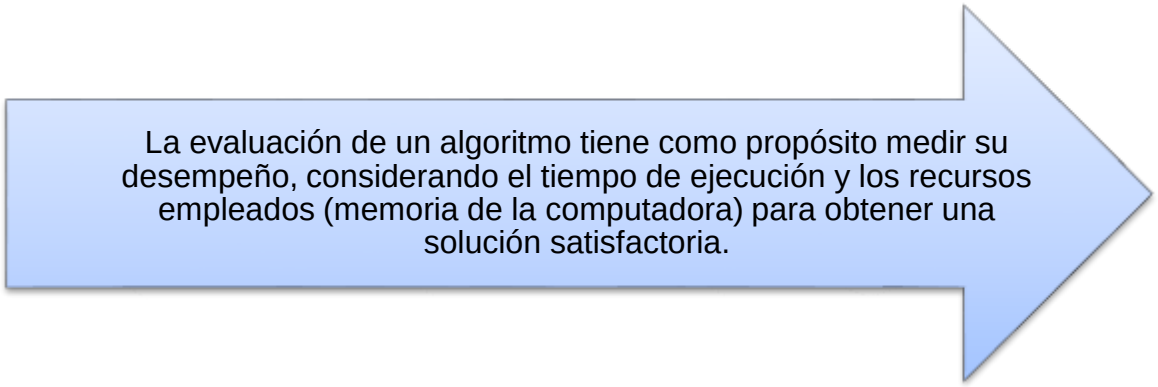
En esta unidad, se estudia la depuración y prueba de programas, con el fin de asegurar que estén libres de errores y cumplan eficazmente con el objetivo para el que fueron elaborados.

Es necesario documentar lo mejor posible los programas para que tanto analistas como programadores conozcan lo que hacen estos programas, y dejar una evidencia de todas sus especificaciones.

Además, los programas deben ser depurados para que cumplan de manera precisa su objetivo, por lo que se les dará un mantenimiento adecuado. En este orden, también se profundiza en los tipos de mantenimiento preventivo, correctivo y adaptativo.

5.1. Refinamiento progresivo

En la unidad anterior, se abordó el tema de refinamiento progresivo, que es la descomposición de un problema en *n* problemas para facilitar su solución, y al final integrarlos en una solución global. Lo que corresponde ahora es analizar la evaluación de los algoritmos con el fin de medir su eficiencia.



En muchas ocasiones, se le da mayor peso al tiempo que tarda un algoritmo en resolver un problema.

Para medir el tiempo de ejecución, el algoritmo se puede transformar a un programa de computadora. Aquí se involucran otros factores, como el lenguaje de programación elegido, sistema operativo empleado, habilidad del programador, etcétera.

Pero también hay otra forma, se puede medir el número de operaciones que realiza un algoritmo considerando el tamaño de las entradas al mismo (N). Entre más grande es la entrada, mayor será su tiempo de ejecución.

También se debe tomar en cuenta cómo está el conjunto de datos de entrada con el que trabajará el algoritmo. Como en los algoritmos de ordenación, el peor caso es que las entradas se encuentren totalmente desordenadas; el mejor, que estén totalmente ordenadas; y en el promedio, que aparezcan parcialmente ordenadas.

Como ejemplo, se exponen a continuación los algoritmos de ordenación por inserción y de ordenación por selección.

Ordenación por inserción

Se trata de ordenar un arreglo formado por *n* enteros. Para esto el algoritmo de inserción va intercambiando elementos del arreglo hasta que esté ordenado.

```
procedimiento Ordenación por Inserción ( var T [ 1
.. n ] )
para i := 2 hasta n hacer
x := T [ i ] ;
j := i - 1 ;
mientras j > 0 y T [ j ] > x hacer
T [ j + 1 ] := T[ j ] ;
j := j - 1
fin mientras ;
T [ j + 1 ] := x
fin para
fin procedimiento
```

Como se observa, *n* es una variable o constante global que indica el tamaño del arreglo.

Los resultados obtenidos dependen, en parte, de la inicialización del arreglo de datos, que puede ser creciente, decreciente o aleatoria. El peor caso ocurre cuando el arreglo está inicializado descendentemente. El mejor, si el arreglo está inicializado ascendentemente (el algoritmo recorre el arreglo hasta el final sin “apenas” realizar trabajo, pues ya está ordenado).

Se ha calculado empíricamente la complejidad para este algoritmo, y se ha obtenido una complejidad lineal cuando el arreglo está inicializado en orden ascendente; y una complejidad cuadrática O(n2) si el arreglo está inicializado en orden decreciente, y también cuando lo hace aleatoriamente.

Ordenación por selección

Se trata de ordenar un arreglo formado por *n* enteros. Para esto el algoritmo de selección va seleccionando los elementos menores al actual y los intercambia.

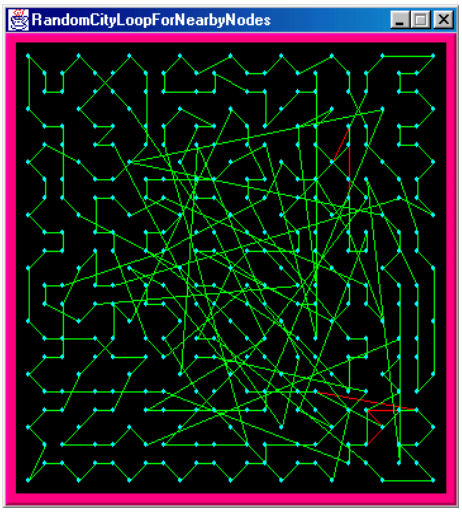
```
procedimiento Ordenación por Selección ( var T [ 1 .. n
] )
para i := 1 hasta n - 1 hacer
minj := i ;
minx := T [ i ] ;
para j := i + 1 hasta n hacer
si T [ j ] < minx entonces
minj := j ;
minx := T [ j ]
fin si
fin para ;
T [ minj ] := T [ i ] ;
T [ i ] := minx
fin para
fin procedimiento
```

Se observa que *n* es una variable o constante global que indica el tamaño del arreglo.

Al igual que en el caso anterior, los resultados obtenidos dependen de la inicialización del arreglo de datos, que puede ser creciente, decreciente o aleatoria:

- El peor caso ocurre cuando el arreglo está inicializado descendentemente.
- El mejor caso se da tanto para la inicialización ascendente como para la aleatoria.

En comparación con la ordenación por inserción, para este algoritmo de selección los tiempos fluctúan mucho menos entre las diferentes inicializaciones del arreglo. Esto se debe a que en éste se realiza prácticamente el mismo número de operaciones en cualquier inicialización del arreglo.



Se ha calculado empíricamente la complejidad para este algoritmo y se ha obtenido que, para cualquier inicialización del arreglo de datos, el algoritmo tiene una complejidad cuadrática O (n2).

5.2. Depuración y prueba

Depuración

Una anécdota sobre el origen de este término (del inglés *debugging*, 'eliminación de bichos') cuenta que en la época de la primera generación de computadoras constituidas por bulbos encontraron una polilla entre los circuitos que era la responsable de la falla del equipo. De allí nació la expresión para indicar que el equipo o los programas presentan algún problema.

La *depuración*, entonces, es el proceso de identificación y corrección de errores de programación. Para depurar el código fuente, el programador se vale de herramientas de *software* que le facilitan la localización y eliminación de errores. Los compiladores son un ejemplo de estas herramientas.

Se dice que un programa está depurado si está libre de errores. Cuando se depura un programa se hace un seguimiento de su funcionamiento y se van analizando los valores de sus distintas variables, así como los resultados obtenidos de los cálculos del programa.

Una vez depurado el programa, se solucionan los posibles errores encontrados y se procede a depurar otra vez. Estas acciones se repiten hasta que el programa no contiene ningún error, tanto en tiempo de programación como en ejecución.

Los errores más sencillos de detectar son de sintaxis, presentes cuando alguna instrucción está mal escrita o se omitió puntuación necesaria para el programa.

Existen también errores lógicos; en este caso, aunque el programa no contenga fallas de sintaxis, no realiza el objetivo por el que fue creado. Pueden presentarse incorrecciones en los valores de las variables, ejecuciones de programa que no terminan, imprecisiones en los cálculos, etcétera.

Estos últimos son los más difíciles de detectar, por lo que se debe realizar un seguimiento puntual del programa.

Prueba de programas

El propósito de las pruebas es asegurar que el programa produce los resultados definidos en las especificaciones funcionales. El programador a cargo utilizará los datos de prueba para comprobar que el programa genera los resultados correctos.

O sea, que se produzca la acción correcta en el caso de datos correctos, o el mensaje de error; y una acción correcta en el caso de datos incorrectos.

Concluida la programación, el analista volverá a usar los datos de prueba para verificar que el programa o sistema dalos resultados correctos. En esta ocasión, concentrará su atención también en la interacción correcta entre los diferentes programas y el funcionamiento completo del sistema.

Verificará lo siguiente:

- Todos los registros incluidos en los datos de prueba.
- Todos los cálculos efectuados por el programa.
- Todos los campos del registro cuyo valor determine una acción a seguir dentro de la lógica del programa.
- Todos los campos que el programa actualice.
- Los casos en que haya comparación contra otro archivo.
- Todas las condiciones especiales del programa.
- Se cotejará la lógica del programa.

5.3. Documentación del programa

La documentación de programas es una extensión de la documentación del sistema. El programador convierte las especificaciones de programas en lenguaje de computadora y debe trabajar conjuntamente con las especificaciones de programas, y comprobar que el programa cumpla con las mismas.

Cualquier modificación que surja como resultado de la programación deberá ser expuesta y aceptada antes de aplicar el cambio.

Nombre del programa (código). Indicará el código que identifica el programa y el título del mismo
Descripción*.* Señalará la función que realiza el programa.
Frecuencia de procesamiento*.* Diaria, semanal, quincenal, mensual, etcétera.
Fecha de vigencia. Fecha a partir de la cual se comienza a ejecutar en producción la versión modificada o desarrollada del programa.
Archivos de entrada.
Lista de archivos de salida*.* Indicará el nombre, copia y descripción de los archivos.
Lista de informes y/o totales de control. Se mostrará el nombre de los informes y se incluirá ejemplo de los informes y/o totales de control producidos por el programa, utilizando los datos de prueba.
Datos de prueba*.* Se incluirá una copia de los datos usados para prueba.
Mensajes al operador. Pantallas de definición de todos los mensajes al operador por consola y las posibles contestaciones, con una breve explicación de cada una de ellas.

Datos de control. Para ejecutar el programa (parámetros).
Transacciones.
Nombre del programador*.* Deberá indicar el nombre del programador que escribió el programa o efectuó el cambio, según sea el caso.
Fecha*.* Indicará la fecha cuando se escribió el programa o efectuó el cambio, según sea el caso.
Diccionario de datos*.* Si aplica, se incluirá detalle de las diferentes tablas y códigos usados con los valores, explicaciones yempleo en el programa.
Lista de programas. Deberá incluir copia de la última compilación del programa con todas las opciones.

5.4. Mantenimiento de programas

Los usuarios de los programas solicitarán los cambios necesarios al área de sistemas con el fin de que éstos continúen operando correctamente. Para ello, se le debe dar mantenimiento periódicamente a los programas.

El mantenimiento presenta las modalidades descritas a continuación.

PREVENTIVO	• Cuando los programas no presentan errores, pero hay necesidad de regenerar los índices de los registros, realizar respaldos, verificar la integridad de los programas, actualizar porcentajes y tablas de datos, etcétera.
CORRECTIVO	• Los programas presentan algún error en algún reporte, por lo que es necesario revisar la codificación para depurarlos y compilarlos. Se deben realizar las pruebas al sistema, imprimiendo los reportes que generan y verificar si los cálculos que éstos ofrecen son correctos.
ADAPTATIVO	• Los programas no tienen errores, mas se requiere alguna actualización por una nueva versión del programa, una nueva plataforma de sistema operativo o un nuevo equipo de cómputo con ciertas características. Es decir, hay que adaptar los programas a la nueva tecnología tanto de *software* como de *hardware*.

En cualquier caso, el usuario debe realizar la solicitud formal por escrito puntualizando el tipo de mantenimiento que requiere, y remitirla al área de sistema para su revisión y valoración. En tanto, el personal del área de sistemas hará una orden de trabajo para proceder a realizar el servicio pedido.

RESUMEN

En esta unidad, se revisó la evaluación de algoritmos como un proceso de análisis de desempeño del tiempo de ejecución para encontrar una solución, y la cantidad de recursos empleados para ello. Se estudió también la importancia de la depuración y prueba de programas con el fin de asegurar que estén libres de errores y cumplan eficazmente con el objetivo para el que fueron elaborados.

De igual manera, se subrayó la relevancia de la documentación de los programas para que tanto analistas como programadores conozcan su dinámica y el fin para los que fueron creados, y tengan un archivo con sus especificaciones. Además, se expuso el tema del mantenimiento, específicamente preventivo, correctivo y adaptativo, fundamentales para el buen funcionamiento y operación de un algoritmo.

BIBLIOGRAFÍA DE LA UNIDAD

SUGERIDA

Autor	Capítulo	Páginas
Diniz (2008)	6. Data-selective filtering	231-287

Du, Ding-Zhu; Ker-I, Ko & Xiaodong, Hu. (2012). "Restriction". En: *Design and analysis of approximation algorithms*. NY: Springer.

Gagliolo, Matteo y Legrand, Catherine. "Algorithm Survival Analysis". En: Thomas Bartz-Beielstein. (2010). *Experimental methods for the analysis of optimization algorithms*. Berlín: Springer

REFERENCIA BIBLIOGRÁFICA

BÁSICA

Bataller, Jordi y Rafael, Magdalena. (2004). *Programación en C*. Madrid: Alfaomega / UPV.

Cairó Batistutti, Osvaldo. (2002). *Metodología de la programación*: algoritmos, *diagramas de flujo y programas*. México: Alfaomega.

Ceballos, Francisco J. (2004). *Microsoft visual C++. Aplicaciones para Win 32*. (2ª ed.) México: Alfaomega / Ra-Ma.
----------. (2004a). *Enciclopedia del lenguaje C*. México: Alfaomega / Ra-Ma.
----------. (2004b). *Enciclopedia del lenguaje C++*. México: Alfaomega / Ra-Ma.
----------. (2004c). *Java 2, curso de programación*. México: Alfaomega / Ra-Ma.
----------. (2004d). *El lenguaje de programación C#*. México: Alfaomega / Ra-Ma.

Flores Rueda, Roberto. (2005). *Algoritmos, estructuras de datos y programación orientada objetos*. Bogotá: ECOE.

García, Luis; Cuadrado, Juan; Amescua, Antonio de y Velasco, Manuel. (2004). *Construcción lógica de programas, teorías y problemas resueltos.* México: Alfaomega / Ra-Ma.

López, Leobardo. (2004). *Programación estructurada: un enfoque algorítmico*. (2ª ed.) México: Alfaomega.

Peñalosa, Ernesto. (2004). *Fundamentos de programación C/C++.*(4ª ed.) México: Alfaomega / Ra-Ma.

COMPLEMENTARIA

Cairó Battistutti, Osvlado. (2006). *Fundamentos de programación: piensa en C*. México: Pearson Educación.

García Pérez, J. Baltasar y Laza Fidalgo, Rosalía. (2008). *Metodología y tecnología de la programación*. Madrid: Pearson Prentice Hall.

Rodríguez, Carlos G. (2003). *Ejercicios de programación creativos y recreativos en C++*. México: Thompson.

Van Gelder, Baase. (2003). *Algoritmos computacionales*. (3ª ed.) México: Thompson.

BIBLIOGRAFÍA ELECTRÓNICA

(Nota: todos los enlaces, consultados o recuperados, funcionan al 18/09/13 [dd/mm/aa])

	Libros	
Fuente	Capítulos (s) Unidad (es) que soporta	Liga
Ahn, Chang Wook. (2006). *Advances in evolutionary algorithms: theory, design and practice*. Berlín, Heidelberg: Springer.	Capítulo 2 (U,3)	http://link.springer.com/book/10.1007/3-540-31759-7/page/1
Axelson-Fisk, Marina. (2010). *Implementation of a Comparative Gene Finder. En: Comparative gene finding: models, algorithms and implementation*. Londres: Springer.	Texto completo (U,3)	http://link.springer.com/chapter/10.1007/978-1-84996-104-2_7
Du, Ding-Zhu; Ker-I ,Ko & Xiaodong, Hu. (2012). "Restriction". En: *Design and analysis of approximation algorithms*. NY: Springer.	Texto completo (U, 3 y 4)	http://link.springer.com/book/10.1007/978-1-4614-1701-9/page/1
Fuchs, Fabian; Völker, Markus y Wagner, Dorothea. (2012). Simulation-Based Analysis of Topology Control Algorithms for Wireless Ad Hoc Networks. En: *Design and Analysis of Algorithms. Lecture Notes in Computer Science.* 7659, 188-202.	Texto completo (U, 2)	http://link.springer.com/chapter/10.1007/978-3-642-34862-4_14#page-1

Gaydecki, Patrick. (2004). *Foundations of digital signal processing: theory, algorithms and hardware design*. Londres: Institution of Electrical Engineers.	Capítulo 1 (U, 1)	http://digital-library.theiet.org/content/books/cs/pbcs015e
Ghosh, Sumit. (2004). *Algorithm design for networked information technology systems*. NY: Springer Verlag.	Capítulo 1, 2 y 3 (U,1)	http://ehis.ebscohost.com/ehost/ebookviewer/ebook/nlebk_108086_AN?sid=83214e9a-9ba4-4df6-9f30-b696cfd62487@sessionmgr198&vid=1&format=EB&lpid=lp_III
Gagliolo, Matteo y Legrand, Catherine. "Algorithm Survival Analysis". En: Thomas Bartz-Beielstein. (2010). *Experimental methods for the analysis of optimization algorithms*. Berlín: Springer.	Capítulo 7 (U, 4)	http://link.springer.com/chapter/10.1007/978-3-642-02538-9_7
Hopcroft, John E.; Motwani, Rajeevy y Ullman, Jeffrey D. (2009). *Introducción a la teoría de autómatas, lenguajes y computación*. (3a ed.) México: Pearson.	Capítulo 8 (U,1)	http://unam.libri.mx/libro.php?libroId=59#
Lee, Kang Seok. (2009). "Standard Harmony Search Algorithm for Structural Design Optimization". En: Zong Woo Geem. (Ed.) *Harmony search algorithms for structural design optimization*. Berlin: Springer.	Texto completo (U,2)	http://link.springer.com/chapter/10.1007/978-3-642-03450-3_1
Lee, RCT y otros. (2007). *Introducción al diseño y análisis de algoritmos: un enfoque estratégico*. México: McGraw-Hill.	Capítulo 1 (U,4)	http://unam.libri.mx/libro.php?libroId=125
Landau, Yoan D. (2011). *Adaptive control: algorithms, analysis and applications*. (2ª ed.) Londres: Springer	Capítulo 2 (U,2)	http://link.springer.com/book/10.1007/978-0-85729-664-1/page/1

Magoulès, F. (2010). *Fundamentals of grid computing: theory, algorithms and technologies*. Boca Raton, Florida: CRC Press.	Capítulo 9 (U,2 y 3)	http://www.netLibrary.com/urlapi.asp?action=summary&v=1&bookid=338997
Markovsky. Ivan. (2012). "Algorithms". En: Ivan Markovsky. *Low rank approximation: algorithms, implementation, applications*. Londres: Springer.	Texto completo (U,4)	http://link.springer.com/chapter/10.1007/978-1-4471-2227-2_3
Norton, Peter. (2006). *Introducción a la computación*. (3ª ed.) México: McGraw-Hill.	Capítulo 13 (U, 1 y 2)	http://unam.libri.mx/libro.php?libroId=123
Skiana, Steven S. (2008). *The algorith design manaual*. (2ª ed.) Londres: Springer / Verlag.	Capítulo 2 (U,2)	http://link.springer.com/chapter/10.1007/978-1-84800-070-4_2#page-1
Tempo, Roberto; Dabbene, Fabrizio y Calafiore, Giuseppe. (2005). Applications of Randomized Algorithms. En: *Randomized Algorithms for Analysis and Control of Uncertain Systems*. (2nd ed.) Londres: Springer.	Capítulo 20 (U,1)	http://link.springer.com/chapter/10.1007/978-1-4471-4610-0_19
Verma, Rakesh M. y Reyner, Steve. (1989). An Analysis of a Good Algorithm for the Subtree Problem, Corrected. En: *SIAM J. Comput,* 8(5), 906–908.	Texto completo (U,2)	http://epubs.siam.org/doi/abs/10.1137/0218062
Verma, Rakesh M. (1997). General Techniques for Analyzing Recursive Algorithms with Applications. *SIAM Journal Comput*. 26(2), 568–581.	Texto completo (U,2)	http://epubs.siam.org/doi/abs/10.1137/S0097539792240583

Plan 2012

2016

actualizado