UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

DIVISIÓN SISTEMA UNIVERSIDAD ABIERTA Y
EDUCACIÓN A DISTANCIA

L I C E N C I A T U R A en

INFORMÁTICA

APUNTES DIGITALES
PLAN 2011

ANALISIS, DISEÑO E IMPLANTACIÓN DE ALGORITMOS

Plan: 2011

Clave: 1164		Créditos: 8
Licenciatura: INFORMÁTICA		Semestre: 1º
Área: Informática (Desarrollo de sistemas)		Horas. Asesoría: 4
Requisitos: Ninguno		Horas. por semana: 4
Tipo de asignatura:	Obligatoria (x)	Optativa ( )

AUTOR (ES):
GILBERTO MANZANO PEÑALOZA
ADAPTADO A DISTANCIA:
ACTUALIZACION AL PLAN DE ESTUDIOS 2012:
GILBERTO MANZANO PEÑALOZA

Temario oficial

(horas 64)

	Horas
1. FUNDAMENTOS DE ALGORITMOS	12
2. ANÁLISIS DE ALGORITMOS	12
3. DISEÑO DE ALGORITMOS PARA LA SOLUCIÓN DE PROBLEMAS	12
4. IMPLANTACIÓN DE ALGORITMOS	12
5. EVALUACIÓN DE ALGORITMOS	16
TOTAL	64 HORAS

INTRODUCCIÓN GENERAL AL MATERIAL DE ESTUDIO.

Las modalidades abierta y a distancia (SUAYED) son alternativas que pretenden responder a la demanda creciente de educación superior, sobre todo, de quienes no pueden estudiar en un sistema presencial. Actualmente, “con la incorporación de las nuevas tecnologías de información y comunicación a los sistemas abierto y a distancia, se empieza a fortalecer y consolidar el paradigma educativo de éstas, centrado en el estudiante y su aprendizaje autónomo, para que tenga lugar el diálogo educativo que establece de manera semipresencial (modalidad abierta) o vía Internet (modalidad a distancia) con su asesor y condiscípulos, apoyándose en materiales preparados ex profeso”[1].

Un rasgo fundamental de la educación abierta y a distancia es que no exige presencia diaria. El estudiante SUAYED aprende y organiza sus actividades escolares de acuerdo con su ritmo y necesidades; y suele hacerlo en momentos adicionales a su jornada laboral, por lo que requiere flexibilidad de espacios y tiempos. En consecuencia, debe contar con las habilidades siguientes.

[1] Sandra Rocha, *Documento de Trabajo. Modalidad Abierta y a Distancia en el SUA-FCA*, 2006.

- Saber estudiar, organizando sus metas educativas de manera realista según su disponibilidad de tiempo, y estableciendo una secuencia de objetivos parciales a corto, mediano y largo plazos.
- Mantener la motivación y superar las dificultades inherentes a la licenciatura.
- Asumir su nuevo papel de estudiante y compaginarlo con otros roles familiares o laborales.
- Afrontar los cambios que puedan producirse como consecuencia de las modificaciones de sus actitudes y valores, en la medida que se adentre en las situaciones y oportunidades propias de su nueva situación de estudiante.
- Desarrollar estrategias de aprendizaje independientes para que pueda controlar sus avances.
- Ser autodidacta. Aunque apoyado en asesorías, su aprendizaje es individual y requiere dedicación y estudio. Acompañado en todo momento por su asesor, debe organizar y construir su aprendizaje.
- Administrar el tiempo y distribuirlo adecuadamente entre las tareas cotidianas y el estudio.
- Tener disciplina, perseverancia y orden.
- Ser capaz de tomar decisiones y establecer metas y objetivos.
- Mostrar interés real por la disciplina que se estudia, estar motivado para alcanzar las metas y mantener una actitud dinámica y crítica, pero abierta y flexible.
- Aplicar diversas técnicas de estudio. Atender la retroalimentación del asesor; cultivar al máximo el hábito de lectura; elaborar resúmenes, mapas conceptuales, cuestionarios, cuadros sinópticos, etcétera; presentar trabajos escritos de calidad en contenido, análisis y

reflexión; hacer guías de estudio; preparar exámenes; y aprovechar los diversos recursos de la modalidad.

- Además de lo anterior, un estudiante de la modalidad a distancia debe dominar las herramientas tecnológicas. Conocer sus bases y metodología; tener habilidad en la búsqueda de información en bibliotecas virtuales; y manejar el sistema operativo Windows, paquetería, correo electrónico, foros de discusión, chats, blogs, wikis, etcétera.

También se cuenta con materiales didácticos como éste elaborados para el SUAYED, que son la base del estudio independiente. En específico, este documento electrónico ha sido preparado por docentes de la Facultad para cada una de las asignaturas, con bibliografía adicional que te permitirá consultar las fuentes de información originales. El recurso comprende referencias básicas sobre los temas y subtemas de cada unidad de la materia, y te introduce en su aprendizaje, de lo concreto a lo abstracto y de lo sencillo a lo complejo, por medio de ejemplos, ejercicios y casos, u otras actividades que te posibilitarán aplicarlos y vincularlos con la realidad laboral. Es decir, te induce al “saber teórico” y al “saber hacer” de la asignatura, y te encauza a encontrar respuestas a preguntas reflexivas que te formules acerca de los contenidos, su relación con otras disciplinas, utilidad y aplicación en el trabajo. Finalmente, el material te da información suficiente para autoevaluarte sobre el conocimiento básico de la asignatura, motivarte a profundizarlo, ampliarlo con otras fuentes bibliográficas y prepararte adecuadamente para tus exámenes. Su estructura presenta los siguientes apartados.

1. *Información general de la asignatura.* Incluye elementos introductorios como portada, identificación del material, colaboradores, datos oficiales de la asignatura, orientaciones para el estudio, contenido y programa oficial de la asignatura, esquema general de contenido, introducción general a la asignatura y objetivo general.

2. *Desarrollo de cada unidad didáctica.* Cada unidad está conformada por los siguientes elementos.
   - Introducción a la unidad.
   - Objetivo particular de la unidad.
   - Contenidos.
   - Actividades de aprendizaje y/o evaluación. Tienen como propósito contribuir en el proceso enseñanza-aprendizaje facilitando el afianzamiento de los contenidos esenciales. Una función importante de estas actividades es la retroalimentación: el asesor no se limita a valorar el trabajo realizado, sino que además añade comentarios, explicaciones y orientación.
   - Ejercicios y cuestionarios complementarios o de reforzamiento. Su finalidad es consolidar el aprendizaje del estudiante.
   - Ejercicios de autoevaluación. Al término de cada unidad hay ejercicios de autoevaluación cuya utilidad, al igual que las actividades de aprendizaje, es afianzar los contenidos principales. También le permiten al estudiante calificarse él mismo cotejando su resultado con las respuestas que vienen al final, y así podrá valorar si ya aprendió lo suficiente para presentar el examen correspondiente. Para que la

autoevaluación cumpla su objeto, es importante no adelantarse a revisar las respuestas antes de realizar la autoevaluación; y no reducir su resolución a una mera actividad mental, sino que debe registrarse por escrito, labor que facilita aún más el aprendizaje. Por último, la diferencia entre las actividades de autoevaluación y las de aprendizaje es que éstas, como son corregidas por el asesor, fomentan la creatividad, reflexión y valoración crítica, ya que suponen mayor elaboración y conllevan respuestas abiertas.

3. *Resumen por unidad*.
4. *Glosario de términos.*
5. *Fuentes de consulta básica y complementaria*. Mesografía, bibliografía, hemerografía y sitios web, considerados tanto en el programa oficial de la asignatura como los sugeridos por los profesores.

Esperamos que este material cumpla con su cometido, te apoye y oriente en el avance de tu aprendizaje.

Recomendaciones (orientación para el estudio independiente)

- Lee cuidadosamente la introducción a la asignatura, en ella se explica la importancia del curso.
- Revisa detenidamente los objetivos de aprendizaje (general y específico por unidad), en donde se te indican los conocimientos y habilidades que deberás adquirir al finalizar el curso.
- Estudia cada tema siguiendo los contenidos y lecturas sugeridos por tu asesor, y desarrolla las actividades de aprendizaje. Así

podrás aplicar la teoría y ejercitarás tu capacidad crítica, reflexiva y analítica.

- Al iniciar la lectura de los temas, identifica las ideas, conceptos, argumentos, hechos y conclusiones, esto facilitará la comprensión de los contenidos y la realización de las actividades de aprendizaje.
- Lee de manera atenta los textos y mantén una actitud activa y de diálogo respecto a su contenido. Elabora una síntesis que te ayude a fijar los conceptos esenciales de lo que vas aprendiendo.
- Debido a que la educación abierta y a distancia está sustentada en un principio de autoenseñanza (autodisciplina), es recomendable diseñar desde el inicio un plan de trabajo para puntualizar tiempos, ritmos, horarios, alcance y avance de cada asignatura, y recursos.
- Escribe tus dudas, comentarios u observaciones para aclararlas en la asesoría presencial o a distancia (foro, chat, correo electrónico, etcétera).
- Consulta al asesor sobre cualquier interrogante por mínima que sea.
- Revisa detenidamente el plan de trabajo elaborado por tu asesor y sigue las indicaciones del mismo.

Otras sugerencias de apoyo

- Trata de compartir tus experiencias y comentarios sobre la asignatura con tus compañeros, a fin de formar grupos de estudio presenciales o a distancia (comunidades virtuales de

aprendizaje, a través de foros de discusión y correo electrónico, etcétera), y puedan apoyarse entre sí.

- Programa un horario propicio para estudiar, en el que te encuentres menos cansado, ello facilitará tu aprendizaje.
- Dispón de periodos extensos para al estudio, con tiempos breves de descanso por lo menos entre cada hora si lo consideras necesario.
- Busca espacios adecuados donde puedas concentrarte y aprovechar al máximo el tiempo de estudio.

INTRODUCCIÓN GENERAL A LA ASIGNATURA

Los algoritmos se pueden definir como la secuencia lógica y detallada de pasos para solucionar un problema, por lo que su estudio es útil para dar una solución computable a los problemas que se presentan en las organizaciones.

El campo de los algoritmos es amplio y dinámico. Los algoritmos intervienen directamente en la vida de las organizaciones, solucionando problemas mediante el empleo de programas de computadora para su aplicación en las distintas áreas de la empresa, de ahí que sean objeto

de estudio de la asignatura Análisis, diseño e implantación de algoritmos.

Unidad 1. Se definen los conceptos necesarios para comprender los algoritmos y sus características, así como los autómatas y los lenguajes formales utilizados; se aborda el autómata finito determinista, conocido como la Máquina de Turing, y algunos ejemplos de su aplicación.

Unidad 2. El análisis del problema, los problemas computables y no computables, la recursividad y los algoritmos de ordenación y búsqueda; se estudian los problemas que se pueden resolver mediante la máquina de Turing, por lo que se les denomina problemas computables o decidibles, y aquéllos que no se puedan solucionar por esta forma o tarde bastante su proceso por la complejidad del algoritmo, que se denominan problemas no computables. Se aborda la recursividad, que es la capacidad de una función de invocarse a sí misma, es decir que alguno de los pasos de la función contiene una llamada a la misma función con el pase de valores, los cuales se irán modificando cada vez, hasta alcanzar un caso base que detenga al algoritmo para posteriormente retornar el resultado a la función que la invocó. En la mayor parte de las aplicaciones empresariales se utilizan los algoritmos de ordenación y búsqueda, aquí radica la importancia de su estudio. Se analizarán algoritmos de ordenación tales como: burbuja, inserción, selección y rápido ordenamiento (*quick sort*), y algoritmos de búsqueda como la secuencial, binaria o dicotómica, y la técnica hash.

Unidad 3. Aborda 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, la programación dinámica, divide y vencerás, vuelta atrás, y la ramificación y poda.

Unidad 4. Se da a conocer la manera de implementar los algoritmos mediante programas de cómputo en los que se utiliza la programación estructurada, que consiste en utilizar estructuras de control como: *si condición entonces sino, mientras condición hacer, hacer mientras condición, hacer hasta condición, y para x desde limite1 hasta limite2 hacer*. También se aborda el estudio de los enfoques de diseño de algoritmos como el diseño descendente *(top down*) y el diseño 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.

Unidad 5. Se trata el refinamiento progresivo de los algoritmos mediante la depuración y prueba de los programas. Se estudia la documentación de los programas, así como los diferentes tipos de mantenimiento: preventivo, correctivo y adaptativo.

OBJETIVO GENERAL DE LA ASIGNATURA

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

ESTRUCTURA CONCEPTUAL

Análisis, diseño e implementación de algoritmos

1.- Fundamentos de algoritmos
- ✓ Definición de Algoritmo
- ✓ Autómatas y lenguajes formales
- ✓ Maquina de Turing

2.- Análisis de algoritmos
- ✓Análisis del problema
- ✓Compatibilidad
- ✓Algoritmos recursivos
- ✓Algoritmos de Búsqueda y ordenación

3.- Diseño de algoritmos para la solución de problemas.
- ✓Niveles de abstracción para la construcción de algoritmos
- ✓Técnicas de diseño de algoritmos

4.- Implantación de algoritmos
- ✓El programa como una expresión computable del algoritmo
- ✓Programación estructurada
- ✓Modularidad
- ✓Enfoque de algoritmos

5.- Evaluación de algoritmos
- ✓Refinamiento progresivo
- ✓Depuración y prueba
- ✓Documentación del programa
- ✓Mantenimiento de programas

UNIDAD 1

FUNDAMENTOS DE ALGORITMOS

OBJETIVO ESPECÍFICO

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

INTRODUCCIÓN

La palabra algoritmo viene de *Al-Khowarizmi,* sobrenombre del célebre matemático Mohamed Ben Musa. Hoy en día, el algoritmo es una forma ordenada de describir los pasos para resolver problemas. Es una forma abstracta de reducir un problema a un conjunto de pasos que le den solución.

Hay algoritmos muy sencillos y de gran creatividad, aunque también están los que 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 él.

Será de gran utilidad involucrarse en su funcionamiento y terminología para entender que, bajo el contexto de autómatas, los conceptos de *alfabeto*, *frase*, *cadena vacía, lenguaje*, *gramática*, etcétera, cobran particular relevancia.

Se definirá y estudiará en particular a la Máquina de Turing, que es un ejemplo de los autómatas finitos deterministas, que realizan sólo una actividad en una situación dada.

Es importante que analices con detalle los ejemplos desarrollados en esta unidad, sobre el diseño y funcionamiento de una Máquina de Turing.

El estudio de los algoritmos y los autómatas es básico y medular para que ejercites un pensamiento lógico y abstracto sobre la forma de abordar los problemas de tu área de desempeño, que es la Informática

LO QUE SÉ

De acuerdo con los conocimientos o idea de lo que es un algoritmo, plantea tu propia definición.

Realiza la actividad en Word y guádala en tu computadora.

TEMARIO DETALLADO
(12 HORAS)

1.1. Fundamentos de algoritmos

- 1.1.1 Definición de algoritmo.
- 1.1.2 Propiedades de los algoritmos
- 1.1.3 Autómatas y lenguajes formales.
- 1.1.4 Máquina de Turing.

1.1. Fundamentos de algoritmos

Definición de algoritmo.

Un algoritmo es un conjunto detallado y lógico de pasos, para alcanzar un objetivo o resolver un problema.

Como ejemplo, el instructivo para armar un modelo de un avión a escala. Si una persona sigue en forma estricta los pasos indicados en el instructivo, obtendrá el avión a escala. Lo mismo obtendría otra persona que se dedicara a armar el mismo modelo.

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 muchas cosas y es muy factible que haga en forma automática algunos de los pasos del instructivo, sin detenerse a pensar en cómo llevarlos a cabo. Pero para una computadora resultaría imposible, ya que la máquina requiere de instrucciones muy detalladas para poder ejecutarlas.

Ejemplificando lo anterior, considérese que a una persona se le pide intercambiar los números 24 y 9. El sujeto, con cierta lógica,

responderá inmediatamente "9 y 24". Ahora, veamos cómo lo haría el procesador de una computadora: se tendría que indicar de qué tipo son los datos que se van 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, asignarle a *num2* el 9 y, posteriormente, indicarle que a la variable *aux* se le asigne el valor contenido en la variable *num1*, a *num1* se le asigne el valor contenido en la variable *num2* y a esta última, se le asigne el valor de la variable *aux,* para posteriormente imprimir los valores de las variables *num1* y *num2,* que exhibirán los números 9 y 24; como observaste, se requieren muchos más pasos para indicarle a una computadora que realice la misma tarea que un ser humano, y la máquina es incapaz de realizar muchas tareas aún.

Propiedades de los algoritmos

Los algoritmos deben de tener características o propiedades para considerarse, valga la redundacia, algoritmos, estas son:

FINITO
El algoritmo debe tener, dentro de la secuencia de pasos para realizar la tarea, una situación o condición que lo detenga, porque 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 indicara que preparen un pastel; suponemos que las personas saben cómo preparar un pastel, y siguiendo las indicaciones de la receta del pastel llegan a un paso en el que se indica que se

agregue azúcar al gusto. Cada persona agregaría azúcar de acuerdo a sus preferencias, pero entonces el resultado ya no sería el mismo, ya que los dos pasteles serían diferentes en sus características. Con este ejemplo concluimos que no se trata de un algoritmo, puesto que existe una ambigüedad en el paso descrito.

OBTENER EL MISMO RESULTADO
Bajo cualquier circunstancia, si se siguen en forma estricta los pasos del algoritmo, siempre se debe llegar a un mismo resultado, como por ejemplo: 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, entonces los pasos en cuestión no pueden considerarse como un algoritmo.

Autómatas y lenguajes formales.[2]

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

[2] Véase mi *Tutorial para la asignatura Análisis, diseño e implantación de algoritmos*, 1ª edición, Fondo editorial FCA, México, 2003.

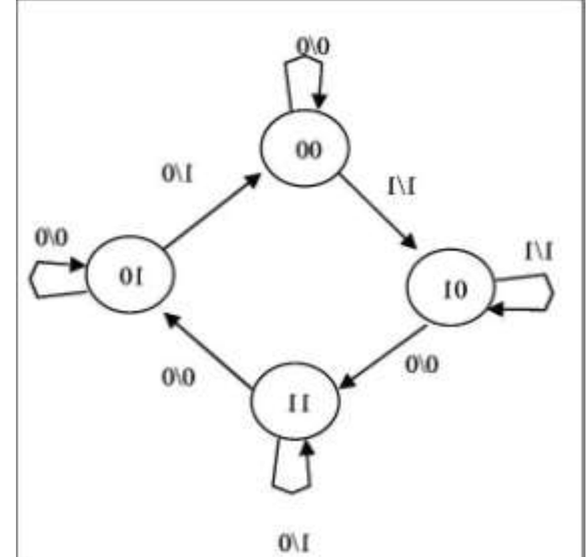
su condición en un instante dado. El autómata comienza en el estado inicial con un conjunto de símbolos; su paso de un estado a otro se efectúa a través de la función de transición, la cual, partiendo del estado actual y un conjunto de símbolos de entrada, lo lleva al nuevo estado correspondiente.

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.
Los inventores más célebres son Pierre Jacquet Droz, autor de "El dibujante" y "Los músicos", y Jacques Vaucanson, autor de "El 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 el denominado “Máquina de Turing”, elaborado por el matemático inglés Alan Mathison Turing.

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

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

En él se pueden visualizar los estados como círculos que en su interior registran su significado, y flechas que representan la transición entre estados y la notación de Entrada/Salida, que provoca la transición entre estados.



En el ejemplo se muestran cuatro diferentes estados de un autómata y se define lo siguiente:

Partiendo del estado "00", si se recibe una entrada "0", la salida es "0" y el autómata conserva el estado actual; pero si la entrada es "1", la salida será "1" y el autómata pasa al estado "01".

Este comportamiento es homogéneo para todos los estados del autómata. Vale la pena resaltar que el autómata que se muestra aquí tiene un alfabeto binario (0 y 1).

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

La siguiente tabla es la correspondiente al diagrama que se presentó en la figura anterior

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

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

$$y = A' x$$

ALFABETO
Un alfabeto se puede definir como el conjunto de todos los símbolos válidos o posibles para una aplicación. Por tanto, en el campo de los autómatas, un alfabeto está formado por todos los caracteres que utiliza para definir sus entradas, salidas y estados. En algunos casos, el alfabeto puede ser infinito o tan simple como el alfabeto binario que se utilizó en el ejemplo del punto anterior, donde sólo se usan los símbolos 1 y 0 para representar cualquier expresión de entrada, salida y estado.

FRASE

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

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

CADENA VACIA

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

Este tipo de cadenas no siempre 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

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

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

GRAMATICAS FORMALES

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

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

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

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

<regla> = sentencias y frases. *

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:

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

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

<raíz> = <opcion>.
<opción> = ARCHIVO
| EDICION
| OPCIONES
| AYUDA

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

<raiz> = Hola mundo | Hola todos

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

Un ejemplo real aplicado a una frase simple de uso común como "Me puede mostrar su licencia", con la opción de anteponer el título

Señorita, Señor o Señora, se puede estructurar de la manera siguiente en una gramática BNF:

<petición> = <comando> | <titulo> <comando> .

<título> = Señor | Señora | Señorita.

<comando> = Me puede mostrar su licencia.

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

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

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

<petición> = <título>? <comando>.

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

LENGUAJE FORMAL

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

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

GRAMÁTICAS SENSIBLES AL CONTEXTO

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

GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO

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

GRAMÁTICAS REGULARES

Son aquéllas que 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 puede ser reconocido como una entrada válida en una máquina de Turing.

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

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

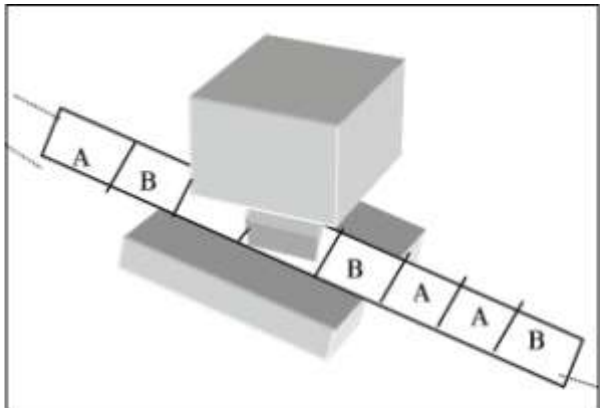
Este tipo de lenguajes se conoce también como lenguajes recursivos o lenguajes totalmente decidibles.

Máquina de Turing.

Un algoritmo es un conjunto de pasos lógicos y secuenciales para solucionar un problema. Este concepto fue implementado en 1936 por Alan Turing, matemático inglés, en la llamada Máquina de Turing (MT).

La Máquina de Turing está formada por tres elementos: una cinta, una cabeza de lectura-escritura y un programa. La cinta tiene la propiedad de ser infinita (no acotada por sus extremos) y estar dividida en segmentos del mismo tamaño, los cuales pueden almacenar cualquier símbolo o estar vacíos. La cinta puede interpretarse como el dispositivo de almacenamiento.

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



El último elemento, el programa, es un conjunto de instrucciones que controla los movimientos de la cabeza de lectura-escritura, indicándole

hacia dónde debe moverse y si debe escribir o leer en la celda donde se encuentre.

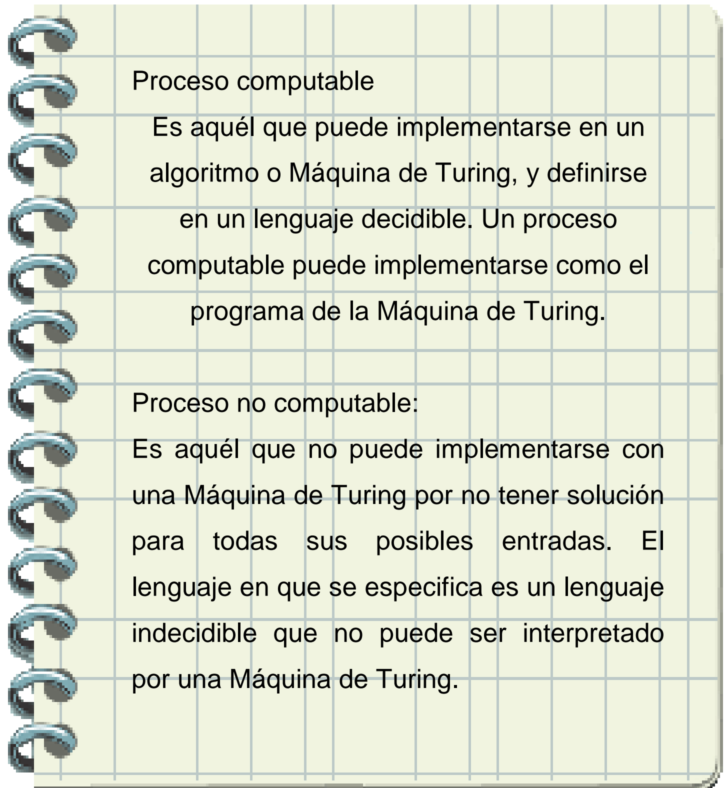
Actualmente, la Máquina de Turing es una de las principales abstracciones utilizadas en la teoría moderna de la computación, ya que auxilia en la definición de lo que una computadora puede o no puede hacer.

Máquina de Turing como función

La Máquina de Turing es una función cuyo dominio se encuentra en la cinta infinita, y es en esta misma donde se plasma su co-dominio, esto es: todos los posibles valores de entrada se encuentran en la cinta y todos los resultados de su operación se plasman también en ella.

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

Hay problemas que pueden resolverse mediante una Máquina de Turing y otros que no. A los primeros se les denomina problemas computables y a los segundos problemas no computables o problemas indecidibles. De ello derivan respectivamente, los procesos computables y los procesos no computables, a saber:



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

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: ∂ (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.* ∂ (q1, b) = (q1, 0, 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 MT se comienza con la cabeza de lectura / escritura sobre el 0, tenemos la siguiente secuencia:

(q1, 0b) ├ (q1, 1b) ├ (q1, 10b) ├ (q1, 11b) ├ (q1, 110b) ├

Nota: el caracter subrayado indica que la cabeza lectora/grabadora de la MT está posicionada sobre ese caracter.

Otro ejemplo:

Diseñar una máquina de Turing que acepte el lenguaje L={$a^n b^m$| n 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* sería primero transformada en *caabbd*, después en *ccabdd,* y por último, en *cccddd*.

Solución:

Consideremos la MT 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 MT se comienza con la cabeza lectora / escritora sobre la primera de la izquierda, se tiene la siguiente secuencia de movimientos:

(q1, $\underline{a}$aabbb) ├ (q2, c$\underline{a}$abbb) ├ (q2, ca$\underline{a}$bbb) ├ (q2, caa$\underline{b}$bb) ├ (q2, caab$\underline{b}$b) ├
(q2, caabb$\underline{b}$) ├ (q2, caabbb $\underline{ß}$) ├ (q5, caabb$\underline{b}$ ß) ├ (q3, caab$\underline{b}$dß) ├ (q3, caa$\underline{b}$bd) ├
(q3, ca$\underline{a}$bbd) ├ (q3, c$\underline{a}$abbd) ├ (q3, $\underline{c}$aabbd) ├ (q1, c$\underline{a}$abbd) ├ (q2, cc$\underline{a}$bbd) ├
(q2, cca$\underline{b}$bd) ├ (q2, ccab$\underline{b}$d) ├ (q2, ccabb$\underline{d}$) ├ (q5, ccab$\underline{b}$d) ├ (q3, cca$\underline{b}$dd) ├
(q3, cc$\underline{a}$bdd) ├ (q3, c$\underline{c}$abdd) ├ (q1, cc$\underline{a}$bdd) ├ (q2, ccc$\underline{b}$dd) ├ (q2, cccb$\underline{d}$d) ├
(q5, ccc$\underline{b}$dd) ├ (q3, cc$\underline{c}$ddd) ├ (q1, ccc$\underline{d}$dd) ├ (q4, ccc$\underline{d}$dd)
ALTO.

Con lo anterior queda ejemplificado el diseño de una MT, así como su desarrollo.

RESUMEN DE LA UNIDAD

En esta primera unidad se presentaron conceptos y principios básicos de los algoritmos, sus características y terminología básica, para aplicación en la resolución de problemas, que es la razón de ser de un algoritmo. Con el apoyo de ejemplos se trató de generar una mejor comprensión de los puntos tratados, ya que los algoritmos pueden ser muy sencillos o muy complejos.

Se estudiaron los autómatas, que son una aplicación de los algoritmos, los cuales, basados en una condición de una situación dada, llevarán a cabo algunas acciones que ya se encuentran programadas. Se definió y estudió en particular a la Máquina de Turing, que es un ejemplo de los autómatas finitos deterministas que realizan sólo una actividad en una situación dada.

GLOSARIO DE LA UNIDAD

Algoritmo

Es un conjunto detallado y lógico de pasos, para alcanzar un objetivo o resolver un problema.

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.

Alfabeto

Es el conjunto de todos los símbolos válidos o posibles para una aplicación, está formado por todos los caracteres que utiliza para definir sus entradas, salidas y estados.

Frase

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

Cadena vacía

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

Lenguaje

Es un conjunto de cadenas que obedecen a un alfabeto fijado.

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.

Lenguaje formal

Está constituido por un alfabeto, un vocabulario y un conjunto de reglas de producción definidas por las gramáticas.

Máquina de Turing

Modelo utilizado en la teoría moderna de la computación para definir si un problema puede o no ser solucionado por una computadora.

ACTIVIDADES DE APRENDIZAJE

Realiza tus actividades en un procesador de textos, guárdalas en tu computadora y una vez concluidas, presiona el botón Examinar. Localiza el archivo, ya seleccionado, presiona Subir este archivo para guardarlo en la plataforma.

ACTIVIDAD 1

Investiga acerca de la aplicación de las características de un algoritmo y redáctalo en una ficha.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

ACTIVIDAD 2

Elabora un algoritmo del ejemplo presentado en el tema:

- Dale nombre a tres variables *num1*, *num2* y *aux.*
- Asígnale a la variable *num1* el número 24.
- Asígnale a *num2* el 9.
- Posteriormente, indícale que a la variable *aux* se le asigne el valor contenido en la variable *num1.*
- A *num1,* asignar el valor contenido en la variable *num2*.
- Asignar a *num 2* el valor de la variable *aux.*
- Posteriormente, imprimir los valores de las variables *num1* y *num2,* que exhibirán los números 9 y 24.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

ACTIVIDAD 3

Lee la unidad 1 "¿Para qué sirven los autómatas?" **(ANEXO 1)** del libro de Introducción a la Teoría de Autómatas, Lenguajes y Computación de John E. Hopcroft y con base en lo leido da un ejemplo de situaciones en las que se pueden aplicar las siguientes demostraciones:

- Demostraciones deductivas
- Demostración de la conversión contradictoria
- Demostración por reducción al absurdo
- Contraejemplos
- Demostraciones inductivas
- Inducciones estructurales

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón Examinar. Localiza el archivo, ya seleccionado, presiona Subir este archivo para guardarlo en la plataforma.

ACTIVIDAD 4

Diseña una MT para determinar si la cantidad de paréntesis de apertura y de cierre está o no balanceada.

Ejemplo: Para la cadena de paréntesis ((()), la MT determinará que no están balanceados. Envía el diseño en un documento a tu asesor.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**.

Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

CUESTIONARIO DE REFORZAMIENTO

Contesta el siguiente cuestionario.

Realiza tu actividad en un documento en Word, guárdala en tu computadora y una vez concluida, presiona el botón Examinar. Localiza tu archivo donde lo guardaste, selecciónalo y presiona Subir este archivo para guardarlo en la plataforma.

1. ¿Qué es un algoritmo?
2. ¿Cuáles son las características de un algoritmo?
3. ¿Qué es un autómata?
4. Explica por qué un termostato puede ser considerado un autómata.
5. ¿Qué es un diagrama de estado?
6. ¿Qué es una tabla de estado?
7. En el campo de autómatas, ¿qué es un alfabeto?
8. Define lo que es una cadena vacía.
9. ¿Cuál es la definición de lenguaje?
10. ¿Qué es y para qué sirve una gramática?
11. Da un ejemplo de una regla de producción BNF.
12. ¿Qué elementos constituyen un lenguaje formal?
13. Describe brevemente tres tipos de gramáticas.
14. Define lo que es una Máquina de Turing.
15. ¿Qué es un proceso computable?

EXAMEN DE AUTOEVALUACIÓN 1

Responde si son verdaderas (V) o falsas (F) las siguientes aseveraciones. Una vez que concluyas, obtendrás tu calificación de manera automática.

	Verdadera	Falsa
1. Es muy factible obviar pasos que sean repetitivos en las operaciones que realiza una computadora.	( )	( )
2. La posibilidad de aplicar diversos criterios es una característica de los algoritmos	( )	( )

EXAMEN DE AUTOEVALUACIÓN 2

Marca en el cuadro, el concepto correspondiente a cada definición.

☐	**1.** Conjunto de todos los símbolos validos o posibles para una aplicación.	1 Frase 2 Gramática 3 Alfabeto 4 Lenguaje 5 Cadena vacía
☐	**2.** Es la asociación de un conjunto de símbolos definidos en un alfabeto (cadena), que tiene la propiedad de tener sentido, significado y lógica.	

☐ **3.** Conjunto de cadenas que obedecen a un alfabeto fijado.	
☐ **4.** La longitud del conjunto de caracteres que utiliza es igual a cero.	
☐ **5.** 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.	

EXAMEN DE AUTOEVALUACIÓN 3

Responde si son verdaderas (V) o falsas (F) las siguientes aseveraciones. Una vez que concluyas, obtendrás tu calificación de manera automática.

	Verdadera	Falsa
1. Los procesos computables y no computables pueden implementarse en un algoritmo o maquina de tuning.	( )	( )
2. A un proceso no computable se le puede generar un lenguaje decible para que sea leído en una Maquina de Tuning.	( )	( )
3. La cita de una maquina de tuning es necesariamente finita, ya que está definida por el tamaño del programa.	( )	( )

4. El programa es un conjunto de instrucciones que controla los movimientos de la cabeza de lectura/escrita.	( )	( )
5. La cinta es un dispositivo solo de lectura y no de almacenamiento.	( )	( )

EXAMEN DE AUTOEVALUACIÓN 4

Elige la respuesta correcta para las siguientes preguntas, una vez que concluyas, obtendrás de manera automática tu calificación.

1. Es una característica de un algoritmo:
   - a) Acepta criterios en su desarrollo
   - b) Se pueden omitir pasos al seguir los algoritmos
   - c) En ocasiones, no obtiene un resultado
   - d) Contiene una condición que detiene su ejecución

2. Inventor de “El pato con aparato digestivo”
   - a) Pierre Jacquet Drozz
   - b) Falcon
   - c) Josheph Marie Jaquard
   - d) Jacques Vacanson

3. Autómata que está formado por una cinta, una cabeza de lectura – escritura y un programa:

- a) maquina de turing
- b) el dibujante
- c) los músicos
- d) el telar automatico

4. Problema que no puede implementarse en una maquina de turing por no tener solución para todas sus posibles entradas:

- a) Computable
- b) Indecible
- c) Decidible
- d) disfuncional

5.-Tipo de gramáticas que tienen la característica de que el lado derecho de la regla de producción siempre debe ser igual o mayor que el lado izquierdo:

- a) Gramáticas independientes del contexto
- b) gramáticas sensible al contexto
- c) Gramáticas regulares
- d) Ninguna de las anteriores

LO QUE APRENDÍ

Retoma la definición de algoritmo que has anotado en el apartado *“Lo que sé”* y compleméntala con lo visto en la unidad, y con algunas referencias bibliográficas o mesográficas. No olvides citarlas.

MESOGRAFÍA

Sitios

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

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

http://www.zator.com/Cpp/E0_1_1.htm

http://www.rastersoft.com/articulo/turing.html

http://perseo.dif.um.es/~roque/talf/Material/apuntes.pdf

BIBLIOGRAFÍA BÁSICA

- DE GIUSTI, A. ***Algoritmos, datos y programas con aplicaciones en Pascal, Delphi y Visual Da Vinci***, Buenos Aires, Pearson Education, 2001, 472 pp.
- HOPCROFT, J. MOTWANI, R. y ULLMAN, J. ***Introducción a la teoría de autómatas, lenguajes y computación***, 2ª edición, Madrid, Pearson Addison Wesley, 2002, 584 pp.
- JOYANES, L. ***Estructuras de datos, algoritmos, abstracción y objetos***, México, McGraw Hill, 1998.
- LOZANO, L. ***Diagramación y programación estructurada y libre***, 3ª edición, México, McGraw-Hill, 1986, 384 pp.

- MANZANO, G. ***Tutorial para la asignatura Análisis, diseño e implantación de algoritmos***, Fondo editorial FCA, México, 2003.
- LEE, R. TSENG, S. CHANG, R. y TSAI, Y. ***Introducción al diseño y análisis de algoritmos un enfoque estratégico***, México, McGraw Hill, 2007, 752 pp.
- SEDGEWICK, R. ***Algoritmos en C++***, México, Pearson Education, 1995.
- VAN GELDER, B. ***Algoritmos computacionales Introducción al análisis y diseño***, 3ª ed. México, Thomson, 2002.

BIBLIOGRAFÍA COMPLEMENTARIA

- HERNÁNDEZ, Roberto, ***Estructuras de datos y algoritmos***, MÉXICO, PRENTICE HALL, 2000, 296 PP.
- JOYANES Aguilar Luis, Programación En C++, ***Algoritmos, estructuras de datos y objetos***, MÉXICO, MC.GRAW-HILL, 2000.
- VAN Gelder, Baase, ***Algoritmos Computacionales***, 3ª. ED., MÉXICO, THOMSON, 2003.
- OSVALDO Cairo Battistutti Aniei, ***Fundamentos de programación piensa en C***, 392 pag. (2006)

UNIDAD 2

ANÁLISIS DE ALGORITMOS

OBJETIVO ESPECÍFICO

Al finalizar la unidad, el alumno podrá analizar un problema determinado y buscar una solución a partir de un algoritmo

INTRODUCCIÓN

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

Se abordará la computabilidad como la 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.

Se utilizará la construcción de modelos para abstraer una expresión a sus características más sobresalientes, que sirvan al objetivo del modelo mismo.

Asimismo, se tratarán los problemas decidibles, los cuales pueden resolverse por un conjunto finito de pasos con una variedad de entradas.

Otro punto a abarcar será la recursividad, que es la propiedad de una función de invocarse repetidamente a sí misma hasta encontrar un caso base que le asigne un resultado a la función y retorne esta solución hasta la función que la invocó. La recursión 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 elevarla al número de veces que la función recursiva se llame a sí misma.

Por último, se estudiarán los diferentes métodos de ordenación y búsqueda, los cuales se utilizan con bastante recurrencia en la solución de problemas de negocios, por lo que se hace indispensable su comprensión. Ordenar es organizar un conjunto de datos en una cierta forma que facilite la tarea del usuario de la información, a la vez que facilita su búsqueda y el acceso a un elemento determinado.

LO QUE SÉ

Investiga los elementos de un problema y relaciónalos con los algoritmos, y anota tus conclusiones. Más adelante retomarás éste apartado.

Pulsa el botón **Iniciar o editar mi entrada de diario**, escribe lo que será el inicio de tu protocolo de investigación. Si deseas borrar algo de lo que hasta el momento llevas escrito, pulsa el botón **Revertir**. Cuando decidas concluir tu trabajo del día, pulsa el botón **Guardar cambios**. Pulsa el botón **Comenzar**. Una vez que concluyas, pulsa el botón **Enviar todo y terminar**.

TEMARIO DETALLADO

(12 HORAS)

2.1 Análisis de algoritmos

- 2.1.1 Análisis del problema.
- 2.1.2 Computabilidad.
- 2.1.3 Algoritmos cotidianos.
- 2.1.4 Algoritmos recursivos.
- 2.1.5 Algoritmos de búsqueda y ordenación

2.1. Análisis de Algoritmos

Análisis del problema.

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

Diversos problemas requieren algoritmos diferentes. Un problema puede llegar a tener más de un algoritmo que lo solucione, pero la dificultad se centra en saber cuál algoritmo está mejor implementado, es decir, que dependiendo del tipo de datos a procesar, tenga un tiempo de ejecución óptimo.

Para poder determinar el rendimiento de un algoritmo se deben considerar dos aspectos:

- La cantidad de datos de entrada a procesar y
- El tiempo necesario de procesamiento

El tiempo de ejecución depende del tipo de datos de entrada, que pueden clasificarse en tres casos:

Caso óptimo
Datos de entrada con las mejores condiciones, por ejemplo: que el conjunto de datos se encuentre completamente ordenado

Caso medio
Conjunto estándar de datos de entrada, ejemplo: que el 50% de los datos se encuentre ordenado y el 50% restante no lo esté.

Peor caso
Datos de entrada más desfavorable, por ejemplo: que los datos se encuentren completamente desordenados.

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.

Existen ciertos inconvenientes para no determinar con exactitud el rendimiento de los algoritmos, a saber:

- Algunos algoritmos son muy sensibles a los datos de entrada, modificando cada vez su rendimiento, causando que entre ellos no sean comparables en absoluto.
- Algoritmos bastante complejos, de los cuáles no sea posible obtener resultados matemáticos específicos.

No obstante lo anterior, en la mayoría de los casos sí es posible calcular el tiempo de ejecución de un algoritmo, para así poder

seleccionar el algoritmo que tenga el mejor rendimiento para un problema en particular.

Computabilidad.

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

La computabilidad es la propiedad que tienen ciertos problemas de poder resolverse a través de un algoritmo, como por ejemplo una Máquina de Turing.

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

Representación de un fenómeno descrito

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

La percepción que se tiene de un fenómeno implica conocimiento; cuando se logra su representación, se dice que dicho conocimiento se convierte en un conocimiento transmisible.

Esta representación puede realizarse utilizando diferentes técnicas de abstracción, desde una pintura hasta una función matemática; sin embargo, la interpretación que puede darse a los diferentes tipos de

representación varía de acuerdo a dos elementos: la regulación de la técnica utilizada y el conocimiento del receptor.

De esta manera, un receptor, con 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 nivel de conocimientos matemáticos análogo al nivel de otro receptor siempre dará la misma interpretación a una expresión matemática. Esto se debe a que en el primer caso intervienen factores personales de interpretación, que hacen válidas las diferencias, mientras que en el segundo caso se tiene un cúmulo de reglas que no permiten variedad de interpretaciones sobre una misma expresión. En este tema nos enfocaremos en la representación de fenómenos del segundo caso.

Modelo

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

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

El problema de la decisión

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

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

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

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.

Por ejemplo tenemos el siguiente algoritmo:

Algoritmo para cambiar una llanta ponchada:

Paso 1: Poner el freno de mano del automovil

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

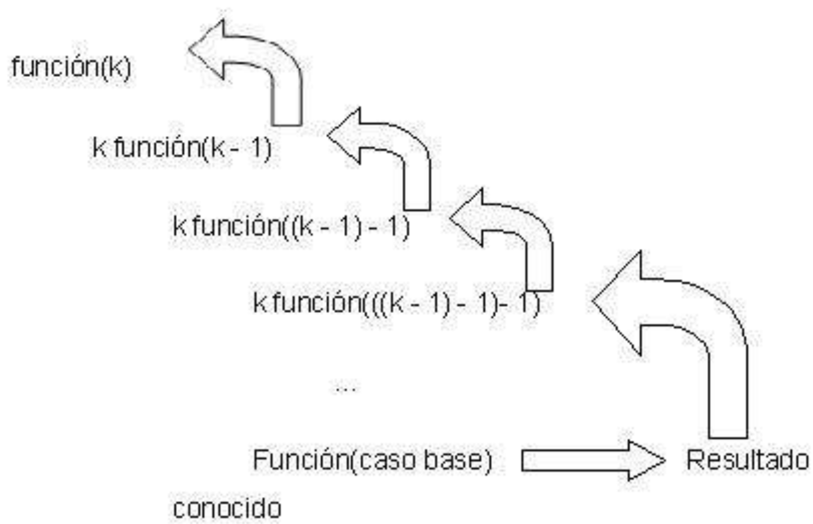
Como se aprecia, la gente común realiza algoritmos cotidianos para realizar sus actividades cotidianas.

Algoritmos recursivos.

Las funciones recursivas son aquéllas que hacen llamadas a sí mismas en su definición, simplificando los valores originales de entrada. Se pueden implementar cuando el problema que se desea resolver puede simplificarse en versiones más pequeñas del mismo problema, hasta llegar a casos simples de fácil resolución.

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

El funcionamiento de una función recursiva puede verse como:



Introducción a la inducción

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

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

Primer principio	Segundo principio
Si el paso base y el paso inductivo asociados a una función recursiva pueden ser	Se basa en afirmaciones de la forma *x* *P(x)*. Esta forma de inducción no requiere del paso

probados, la función será válida para todos los casos que caigan dentro del dominio de la misma. Asimismo, si un elemento arbitrario del dominio cumple con las propiedades definidas en las cláusulas inductivas, su sucesor o predecesor, generado según la cláusula inductiva, también cumplirá con dichas propiedades.	básico, ya que asume que *P(x)* es válido para todo elemento del dominio .

Definición de funciones recursivas

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

BÁSICA
Especifica la semilla del dominio a partir de la cual se generarán todos los elementos del contradominio (resultado de la función), utilizando las reglas definidas en la cláusula inductiva. Este conjunto de elementos se denomina caso base de la función que se está definiendo.
INDUCTIVA
Establece la manera en que los elementos del dominio pueden ser combinados para generar los elementos del contradominio. Esta cláusula afirma que, a partir de los elementos del dominio, se puede generar un contradominio con propiedades análogas.

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

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

- Paso básico

Demostrar que *P*(*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 es válido.

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

- Cálculo de complejidad de una función recursiva

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

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

Algoritmos de búsqueda y ordenación

Al utilizar matrices o bases de datos, las tareas que más comúnmente se utilizan 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.

Algotirmos de búsqueda

SECUENCIAL
Este método de búsqueda, también conocido como lineal, es el más sencillo y 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.

BINARIA O DICOTÓMICA
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. Si el elemento es menor, se busca en el primer subarreglo, y si es mayor, se busca 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 sub-arreglo:

{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 sub-arreglo:

{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 sub-arreglo a dividir está vacio {}, el elemento no se encuentra en el arreglo y la búsqueda termina.

Tablas Hash

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

Las tablas *hash* se implementan sobre arreglos que almacenan grandes cantidades de información, sin embargo, como utilizan

posiciones pseudo-aleatorias, 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
y

0 <= 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 elegir 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 que se requiera hacer una cantidad considerable de búsquedas, y es importante el factor tiempo.

Los métodos de ordenación más conocidos son: burbuja, selección, inserción, y rápido ordenamiento (*quick sort*). Vamos a analizar a cada uno de los métodos de ordenación mencionados.

Burbuja

El método de ordenación por burbuja es el más sencillo, pero el menos eficiente. Se basa en la comparación de elementos adyacentes e intercambio de los mismos si estos 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.

Los pasos a seguir para ordenar un vector por este método son:

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 valor 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. Procedemos 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 con color gris claro representan la posición actual y las celdas más oscuras representan la posición siguiente, de acuerdo a 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 podamos seleccionar 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 básicos:

- 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 identifica 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

Este método 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.

Ejemplo:

Supongamos 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 posición 1, porque ya están ordenadas correctamente.

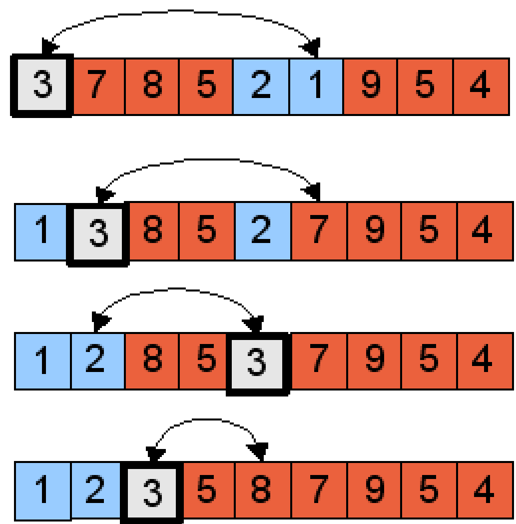
La siguiente tabla muestra las diversas secuencias de la lista de números conforme se van sucediendo las comparaciones y los intercambios:

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, esto es, 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,* que es el arreglo a ordenar, y *N,* que es el número de elementos que se encuentran 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 procederá a colocar en el lugar que le corresponda. Por conveniencia se seleccionará "a" como el primer elemento del arreglo. Se procede a comparar el elemento "a" 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, a la izquierda de "a" 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 realiza el mismo procedimiento. Se toma el subarreglo derecho (los mayores de "a") y se realiza 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 ascende

Shell

A diferencia del algoritmo de ordenación por inserción, este algoritmo intercambia elementos distantes.

La velocidad del algoritmo dependerá de una secuencia de valores (llamados 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

Recabar la información necesaria para indicar una acción para la solución de un problema, en forma adecuada, es fundamental, por lo que se toma como tema inicial en esta unidad. Esta información nos 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 la computabilidad es muy importante, ya que permite dar solución a problemas a través del algoritmo de la Máquina de Turing, permite interpretar un fenómeno a través de un cúmulo de reglas establecidas. También se aborda el concepto de la recursividad, que es cuando una función se invoca repetidamente a sí misma, hasta encontrar un resultado base, y éste retorne a la función que la invocó. A través de la inducción se genera una solución recursiva que implica la abstracción. Como se planteó, esto 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, por lo que

se hace indispensable su comprensión. Ordenar los datos para su mejor manipulación facilita la tarea del usuario de la información, a la vez que facilita su búsqueda y el acceso a un elemento determinado.

GLOSARIO DE LA UNIDAD

Teoría de la computabilidad

Es la parte de la computación que estudia los problemas de decisión que pueden ser resueltos con un algoritmo o, equivalentemente, con una Máquina de Turing.

Modelo

Un Modelo es una representación gráfica o esquemática de una realidad. Sirve para organizar y comunicar de forma clara los elementos que involucran un todo.

Recursión

Es la forma en la cual se especifica un proceso basado en su propia definición. Siendo un poco más precisos, y para evitar el aparente círculo sin fin en esta definición, las instancias complejas de un proceso se definen en términos de instancias más simples, estando las finales más simples definidas de forma explícita.

Inducción matemática

En la inducción matemática se va de lo particular a lo general y, no obstante, se obtiene una conclusión necesaria. Típicamente, el razonamiento inductivo se contrapone al razonamiento deductivo, que va de lo general a lo particular, y sus conclusiones son necesarias

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.

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.

ACTIVIDADES DE APRENDIZAJE

ACTIVIDAD 1

Investiga 5 ejemplos de problemas no decidibles, coméntalos con tus compañeros.

Pulsa el botón **Colocar un nuevo tema de discusión aquí**.
Escribe en el apartado **Asunto** el título de tu aportación, redacta tu comentario en el área de texto y da clic en el botón **Enviar al foro**.

ACTIVIDAD 2

Investiga las diferencias que existen entre la solución iterativa y la solución recursiva, coméntalas en el foro de la asignatura.

Pulsa el botón **Colocar un nuevo tema de discusión aquí**.
Escribe en el apartado **Asunto** el título de su aportación, redacta tu comentario en el área de texto y da clic en el botón **Enviar al foro**.

ACTIVIDAD 3

Realiza un cuadro comparativo con las características de los métodos de ordenación: burbuja, inserción, selección, quick sort y shell.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

ACTIVIDAD 4

Elabora un ejemplo en el que una función hash h(k) convierta un universo de claves (pequeño) en números que se almacenen en una tabla hash que vincule las claves con su valor correspondiente. Utiliza cualquiera de las técnicas de hashing: multiplicativo o por división.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

CUESTIONARIO DE REFORZAMIENTO

Contesta las siguientes preguntas:

Realiza tu actividad en un documento en Word, guárdala en tu computadora y una vez concluida, presiona el botón Examinar. Localiza tu archivo donde lo guardaste, selecciónalo y presiona Subir este archivo para guardarlo en la plataforma.

1. ¿Qué elementos se deben tomar en consideración para determinar el rendimiento de un algoritmo?
2. ¿Qué factores podrían influir en forma negativa para determinar con exactitud el rendimiento de los algoritmos?
3. Define lo que es un modelo.
4. ¿Qué son los problemas decidibles?
5. Explica con tus propias palabras el término recursividad.
6. ¿Qué entiendes por inducción?
7. Describe el método para calcular la complejidad de una función recursiva.
8. ¿Cuál es el método de ordenación menos eficiente y cuál el más eficiente?
9. Explica el concepto “divide y vencerás” que utiliza el método de ordenación *Quick Sort*.
10. ¿Qué diferencia existe entre una tabla *Hash* y una función *hash*? Explica cada una de éstas.

LO QUE APRENDÍ

Retoma el apartado "*Lo que sé*" y complementa la información con lo aprendido y lo investigado, no olvides citar tus referencias.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora

EXAMEN DE AUTOEVALUACIÓN 1

Responde si son verdaderas (V) o falsas (F) las siguientes aseveraciones. Una vez que concluyas, obtendrás tu calificación de manera automática.

	Verdadera	Falsa
1. un problema puede llegar a tener más de un algoritmo que lo solucione.	( )	( )
2. La cantidad de datos de entrada y las operaciones determinan el tiempo de ejecución.	( )	( )
3. Mediante el empleo de formulas matemáticas es posible conocer el rendimiento de un algoritmo.	( )	( )
4. El tiempo de ejecución depende del tipo de datos de salida.	( )	( )
5. Un algoritmo se selecciona en función de su tamaño.	( )	( )

EXAMEN DE AUOEVALUACIÓN 2

☐	**1.** Es la propiedad que tienen ciertos problemas de poder resolverse a través de un algoritmo.	1 Problema de decisión
☐	**2.** Es aquel cuya respuesta puede mapearse al conjunto de valores {0,1}.	2 Computabilidad
☐	**3.** Es el lenguaje que se implementa para resolver un problema con un número finito de pasos por algoritmo.	3 Indecidible
☐	**4.** Tipo de problema cuyo lenguaje no puede ser reconocido por una MT.	4 Recursivo
☐	**5.** Problemas que pueden resolverse con una MT.	5 Computables

EXAMEN DE AUTOEVALUACIÓN 3

Escribe sobre la línea la opción que mejor complete la sentencia.

☐	**1.** Intercambia elementos que están muy distantes.	1 Burbuja
☐	**2.** Emplea la técnica de “divide y vencerás” para separar el problema en subproblemas mas pequeños	2 Selección
☐	**3.** Se basa en selección el elemento más pequeño del arreglo y colocarlo en la posición más baja del mismo.	3 Shell
☐	**4.** Es el método más sencillo, pero el menos eficiente.	4 Inserción
☐	**5.** Método que consiste en tomar un elemento y colocarlo en la posición ordenada correspondiente.	5 Quicksort

MESOGRAFÍA

SITIOS DE INTERÉS

- http://es.scribd.com/doc/27478655/ALGORITMOS-COTIDIANOS, 25/Marzo/2011; responsable de la página: Scribd; descripción del sitio: Definición y ejemplos de algoritmos cotidianos.
- http://www.mailxmail.com/curso-aprende-programar/metodos-ordenamiento-busqueda, 25/Marzo/2011; responsable de la página: Mailxmail, S.L.; descripción del sitio: Métodos de búsqueda y ordenación.
- https://www.itescam.edu.mx/principal/sylabus/fpdb/recursos/r59236.PDF, 25/Marzo/2011; responsable de la página: Instituto Tecnológico Superior de Calkini en el estado de Campeche; descripción del sitio: Análisis y diseño de algoritmos recursivos.

BIBLIOGRAFÍA BÁSICA

- JOYANES, L. ***Estructuras de datos, algoritmos, abstracción y objetos***, México, McGraw Hill, 1998.
- LOZANO, L. ***Diagramación y programación estructurada y libre***, 3ª edición, México, McGraw-Hill, 1986, 384 pp.
- MANZANO, G. ***Tutorial para la asignatura Análisis, diseño e implantación de algoritmos***, Fondo editorial FCA, México, 2003.
- LEE, R. TSENG, S. CHANG, R. y TSAI, Y. ***Introducción al diseño y análisis de algoritmos un enfoque estratégico***, México, McGraw Hill, 2007, 752 pp.
- SEDGEWICK, R. ***Algoritmos en C++***, México, Pearson Education, 1995.
- VAN GELDER, B. ***Algoritmos computacionales Introducción al análisis y diseño***, 3ª ed. México, Thomson, 2002.

BIBLIOGRAFÍA COMPLEMENTARIA

- HERNÁNDEZ, Roberto, ***Estructuras de datos y algoritmos***, MÉXICO, PRENTICE HALL, 2000, 296 PP.
- JOYANES Aguilar Luis, Programación En C++, ***Algoritmos, estructuras de datos y objetos***, MÉXICO, MC.GRAW-HILL, 2000.

UNIDAD 3

DISEÑO DE ALGORITMOS PARA LA SOLUCIÓN DE PROBLEMAS

OBJETIVO ESPECÍFICO

Al terminar la unidad, el alumno podrá plantear, desarrollar y seleccionar un algoritmo determinado para solucionar un problema específico.

INTRODUCCIÓN

En este tema se describirá un método por medio del cual se pueden construir algoritmos para la solución de problemas, además de las características de algunas estructuras básicas usadas típicamente en la implementación de estas soluciones 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 del problema, el diseño de una solución basada en modelos y, por último,

la implementación del algoritmo a través de la escritura del código fuente, con la sintaxis de algún lenguaje de programación.

Todo algoritmo tiene estructuras básicas que están presentes en el modelado de soluciones. En el estudio del tema se abordarán las siguientes: ciclos, contadores, acumuladores, condicionales y las rutinas recursivas.

También se abordarán las diferentes técnicas de diseño de algoritmos para construir soluciones que satisfagan los requerimientos de los problemas, entre las que destacan:

ALGORITMOS VORACES
Son utilizados para la solución de problemas de optimización, son fáciles de diseñar y eficientes al encontrar una solución rápida al problema
DIVIDE Y VENCERÁS
Dividen el problema en forma recursiva, solucionan cada subproblema y la suma de estas soluciones es la solución del problema general
PROGRAMACIÓN DINÁMICA
Definen subproblemas superpuestos y subestructuras óptimas, buscan soluciones óptimas del problema en su conjunto.
VUELTA ATRÁS
(*Backtracking*). Encuentran soluciones a problemas que satisfacen restricciones, van creando todas las posibles combinaciones de elementos para obtener una solución.

RAMIFICACIÓN Y PODA
Encuentra soluciones parciales en un árbol en expansión de nodos, utiliza diversas estrategias (LIFO, FIFO y LC) para encontrar las soluciones, contiene una función de costo que evalúa si las soluciones halladas mejoran a la solución actual; en caso contrario, poda el árbol para ya 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. Al final, tendrás un panorama general de la construcción de algoritmos, sus estructuras básicas y las técnicas de diseño de algoritmos para encontrar soluciones a los diversos problemas que se presentan en las organizaciones.

LO QUE SÉ

Investiga algunos de los distintos niveles de abstracción para la construcción de algoritmos y coméntalas con tu asesor.

TEMARIO DETALLADO

(12 HORAS)

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

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 basada 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 los procesos o funciones del problema, 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 y se ha identificado el punto exacto donde radica y sobre el cual se debe actuar para llegar a una solución, comienza el proceso de modelado de una solución factible, es decir, el diseño. En

esta etapa se debe estudiar el modelo del problema, elaborar hipótesis acerca de posibles soluciones y comenzar a realizar pruebas con éstas.

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

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

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

Existen dos tipos de ciclos:

Mientras	Hasta que
El primero 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

Instrucción1

Instrucción2

...
```

Instrucción n

fin mientras

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

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

hacer

Instrucción1

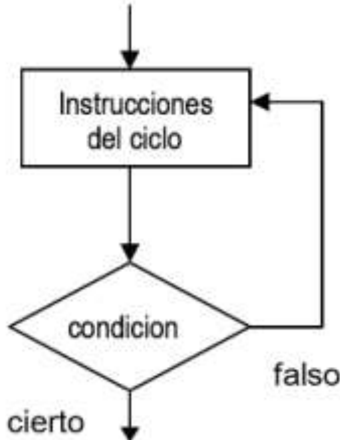
Instrucción2

...

Instrucción n

Hasta que
<condición>

Su diagrama se puede representar como:



Cabe mencionar que las instrucciones contenidas en la estructura *Mientras* se siguen ejecutando mientras la condición resulte verdadera y que a diferencia de la estructura. *Hasta que ésta* continuará iterando siempre y cuando 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 que la segunda estructura está acotada por un comando tanto al inicio como al final de la misma.

Contadores

Este otro tipo de estructura también se caracteriza por iterar instrucciones en función de una condición que debe cumplirse en un

momento conocido, y está representado por la instrucción 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.

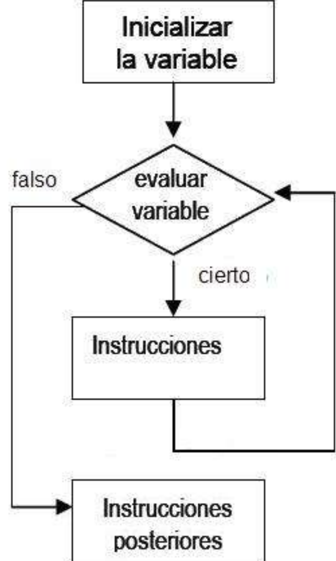
En algunos lenguajes se puede definir el incremento que tendrá la variable, sin embargo, se recomienda que el incremento siempre sea con la unidad.

El pseudocódigo que representa la estructura es el siguiente:

```
Para <variable> =
<valor inicial> hasta
<valor tope> [paso
<incremento>] hacer

    Instrucción1

    Instrucción2

    ...

    Instrucción n

Fin para <variable>
```

Aquí se puede 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:



Acumuladores

Los acumuladores son variables que tienen por objeto almacenar valores incrementales o decrementales a lo largo de la ejecución del algoritmo.

Este tipo de variables utiliza la asignación recursiva de valores para no perder su valor anterior.

Su misión es arrastrar un valor que se va modificando con la aplicación de diversas operaciones y cuyos valores intermedios, así como el final, son importantes para el resultado global del algoritmo. Este tipo de variables generalmente almacena el valor de la solución arrojada por el algoritmo.

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

```
<variable> =
<variable> +
<incremento>
```

Condicionales

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

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

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

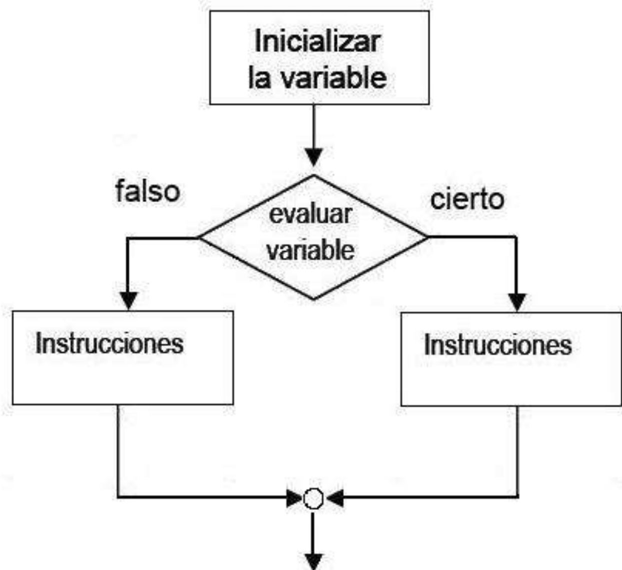
```
si <condición>
  entonces

 Instrucción1

 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:



Rutinas recursivas

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

Este tipo de rutinas se puede implementar en los casos en que el problema que se desea resolver puede simplificarse en versiones más

pequeñas del mismo problema, hasta llegar a casos simples de fácil resolución.

Las rutinas recursivas regularmente contienen una cláusula condicional (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, que es cuando se invoca la rutina a sí misma con valores simplificados.

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

```
<valor_retorno>
Nombre_Función
(<parametroa> [,<parametrob>
...])

si <caso_base> entonces

retorna <valor_retorno>

si no

Nombre_Función (
<parametroa -1> [,
<parametrob -1> ...] )

finsi
```

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

El diagrama asociado a este tipo de rutinas ya se ha ejemplificado en la figura funciones recursivas, que se encuentra ubicada en la Unidad 2, tema 3.

3.2 Técnicas de diseño de algoritmos.

Objetivo

Identificar las características de las diferentes técnicas de diseño, implementadas en el modelado algoritmos.

Desarrollo

Algoritmos Voraces

Los algoritmos voraces típicamente se utilizan en la solución de problemas de optimización, y se caracterizan por ser:

- Sencillos: de diseñar y codificar.
- Miopes: toman decisiones con la información que tienen disponible de forma inmediata, sin tener en cuenta sus efectos futuros.
- Eficientes: dan una solución rápida al problema (aunque ésta no sea siempre la mejor).

Los algoritmos voraces se caracterizan por las propiedades siguientes:

- *Tratan de resolver problemas de forma óptima*
- *Disponen de un conjunto (o lista) de candidatos*

A medida que avanza el algoritmo vamos acumulando dos conjuntos:

- *Candidatos considerados y seleccionados*
- *Candidatos considerados y rechazados*

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

Otra que comprueba 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 pueda obtener una solución a partir de los candidatos disponibles inicialmente.

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

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, como por ejemplo:

Encontrar la secuencia óptima para procesar un conjunto de tareas por una computadora, hallar el camino mínimo de un grafo, etcétera.

Habitualmente, los elementos que intervienen son:

• 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, la longitud del camino, etc.*) y que es la que se pretende maximizar o minimizar.

Divide y vencerás

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

Dividir: los problemas más pequeños se resuelven recursivamente (excepto, por supuesto, los casos base).

Vencer: 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 las rutinas cuyo texto sólo contiene una llamada recursiva.

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, finalmente, combinar la solución de los subproblemas para dar una solución al problema original.

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

Programación dinámica

La programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas. El matemático Richard Bellman inventó la programación dinámica en 1953.

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 tres pasos:

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

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

Vuelta atrás *(Backtracking)*

Vuelta atrás (*Backtracking*) es una estrategia para encontrar soluciones a problemas que satisfacen restricciones. El término *backtrack* fue acuñado por primera vez por el matemático estadounidense D. H. Lehmer, en la década de los 50.

Los problemas que deben satisfacer un determinado tipo de restricciones son problemas completos, donde el orden de los elementos de la solución no importa. Estos problemas 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 posibles combinaciones de elementos para obtener una solución. Su principal virtud es que en la mayoría de las implementaciones se pueden evitar combinaciones, estableciendo funciones de acotación (o poda), reduciendo el tiempo de ejecución

La vuelta atrás está muy relacionada con la *búsqueda combinatoria*. Esencialmente, la idea es encontrar la mejor combinación posible en un momento determinado, por eso se dice que este tipo de algoritmo es una búsqueda en profundidad.

Durante la búsqueda, si se encuentra 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 forma 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 lograr esto, los algoritmos de tipo *backtracking* construyen posibles soluciones candidatas de manera sistemática. En general, dado 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 de tipo *backtracking* se usan para encontrar una solución, pero otras veces interesa que las revisen todas (por ejemplo, para encontrar 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- Última Entrada Primera Salida), en profundidad (estrategia FIFO -First Input First Output- Primera Entrada Primera Salida) o utilizando 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 ése. Si la cota muestra que cualquiera de estas soluciones no es mejor que la solución 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 a aquel nodo con 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 lo más rápidamente posible una solución más económica que la mejor encontrada hasta el momento.

En este proceso se realizan 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 estime el valor óptimo de la solución, si se continúa por ese camino. No se puede realizar poda alguna hasta haber hallado alguna solución.

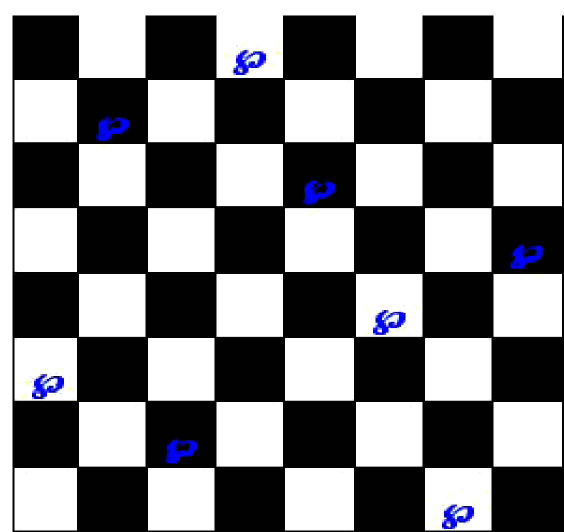
Disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución se traduce en eficiencia. La dificultad está en encontrar una buena función de costo para el problema, buena en el sentido que garantice la poda y que su cálculo no sea muy costoso.

Es recomendable no realizar la poda de nodos sin antes conocer el costo de la mejor solución hallada 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 tener más de un proceso trabajando sobre este conjunto, extrayendo nodos, expandiéndolos y realizando la poda.

Debido a lo anterior, los requerimientos de memoria son 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 para reconstruir la solución encontrada hasta ese momento.

Un ejemplo de aplicación de los algoritmos de ramificación y poda lo encontramos 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; pseudo significa imitación, de modo que pseudocódigo es una imitación de código, al igual que el diagrama de flujo, éste va describiendo la secuencia lógica de pasos mediante enunciados que deben comenzar con un verbo que indique la acción a seguir, seguida de una breve descripción del paso en cuestión.

El caso de usar decisiones se utilizan sentencias como:

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

y si es necesario una bifurcación (cambio de flujo a otro punto del algoritmo) se utilizan etiquetas tales como:

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

Por conservar la sencillez, se debe de usar un lenguaje llano y natural. Cada frase será después que se codifique, una línea de comando del programa.
Las órdenes más usadas son hacer-mientras, hacer-hasta, si-entonces-sino, repite-mientras, un ejemplo sería:

```
Algoritmo: Obtener la suma de los números
del 1 al 100.
Inicio
asigna a = 0
asigna suma = 0
mientras a <= 100
        asigna a = a + 1
asigna suma = suma + a
fin-mientras
imprime “La suma es: “ suma
fin
```

El siguiente paso es la comprobación y luego 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 como un diagrama de flujo en el que se omiten las flechas de unión y las cajas son contiguas. Las acciones sucesivas se pueden escribir en cajas sucesivas y como en los diagramas de flujo, se pueden escribir diferentes acciones en cada caja. Un algoritmo se representa en la siguiente forma:

Inicio
Accion1
Accion2
. . .
Fin

Como ejemplo tenemos el siguiente ejemplo:

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.

Elaborarlo implica diseñar un diagrama de bloque que contenga un bosquejo general del algoritmo y basándose en este proceder a elaborar el diagrama de flujo con todos los detalles necesarios.

Reglas de elaboración le diagramas de flujo:

1.-Debe de 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 requiere notas adicionales colocarlas en el gráfico de anotaciones a su lado.

4.-Se deben de inicializar todas las variables al principio del diagrama. Esto es muy recomendable, ya que nos ayuda a recordar todas las variables, constantes y arreglos que van a ser utilizados en la ejecución del programa. Además, nunca sabemos cuando otra persona, modifique posteriormente el diagrama y necesite 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, en donde se utilizarán conectores de hoja para cada punto en donde se bifurque a otra hoja.

6.-Todos los gráficos deben de estar conectados con flechas de flujo. Jamás debe de dejarse un gráfico sin que tenga alguna salida, a excepción del gráfico que marque el final del diagrama.

Terminado el diagrama de flujo, se realiza la prueba de escritorio, que no es otra cosa, que darle 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 requiere de 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.

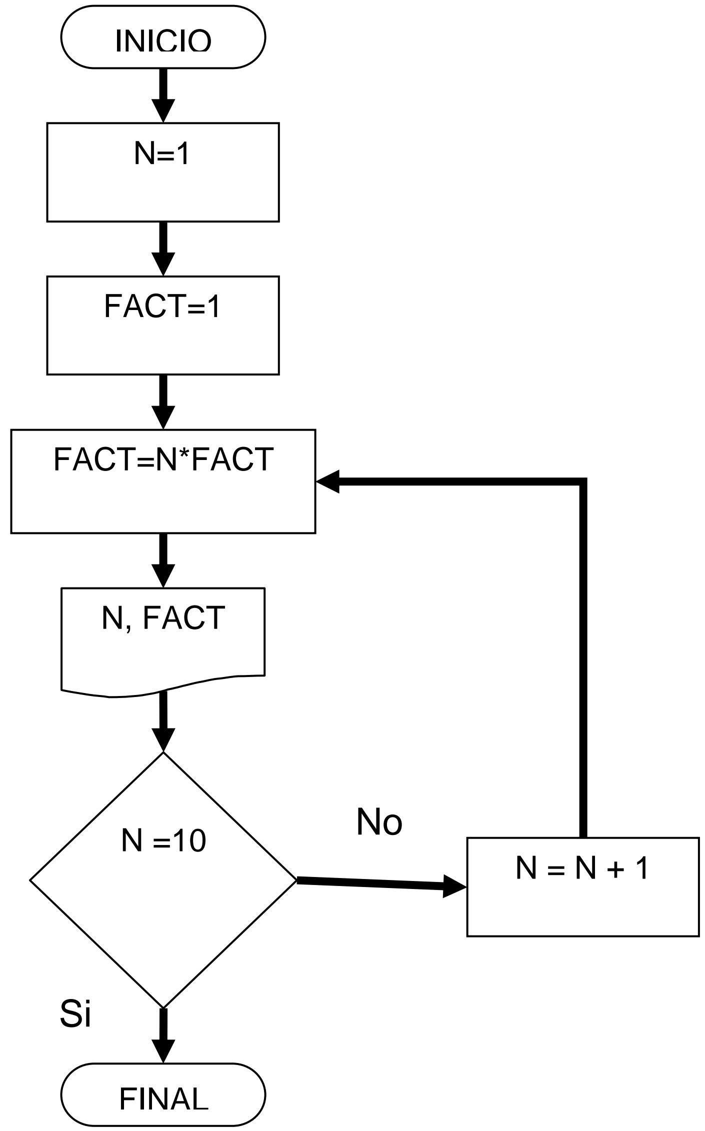
Algunos de los gráficos usados en los diagramas son los siguientes:

SÍMBOLO	DESCRIPCIÓN
	Terminal. Indica el inicio y el final del diagrama de flujo.
	Entrada/Salida. Indica la entrada y salida de datos.
	Proceso. Indica la asignación de un valor en la memoria y/o la ejecución de una operación aritmética.
	Decisión. Indica la realización de una comparación de valores.

Símbolo	Descripción
	Proceso Predefinido. Representa los subprogramas.
	Conector de pagina. Representa la continuidad del diagrama dentro de la misma pagina.
	Conector de Hoja. Representa la continuidad del diagrama en otra página.
	Impresión. Indica la salida de información por impresora.
	Pantalla. Indica la salida de información en el monitor de la computadora.
	Flujo. Indican 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.



Cabe mencionar, que puede haber varias soluciones y alguna ser la más óptima.

RESUMEN

En esta unidad se analizó un método por medio del cual se pueden construir algoritmos, revisando 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, 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 revisaron estructuras como contadores, acumuladores, condicionales y las rutinas recursivas. También se revisan 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, las soluciones parciales en un árbol en expansión de nodos, utilizando diversas estrategias (LIFO, FIFO y LC) para encontrar las soluciones, contiene una función de costo que evalúa si las soluciones halladas mejoran la solución actual.

Generando con esto un panorama general de la construcción de algoritmos, sus estructuras básicas y las técnicas de diseño de algoritmos.

GLOSARIO DE LA UNIDAD

Ciclos

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

Condicionales

Este tipo de estructura se utiliza para ejecutar selectivamente secciones de código, de acuerdo con una condición definida.

Contadores

Este otro tipo de estructura también se caracteriza por iterar instrucciones en función de una condición que debe cumplirse en un momento conocido.

Rutinas recursivas

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

Programación Dinámica

La programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo, mediante la utilización de subproblemas superpuestos y subestructuras óptimas. El matemático Richard Bellman inventó la programación dinámica en 1953.

Pseudocódigo

Es una serie de normas léxicas y gramaticales parecidas a la mayoría de los lenguajes de programación, pero sin llegar a la rigidez de sintaxis de éstos ni a la fluidez del lenguaje coloquial.

ACTIVIDADES DE APRENDIZAJE

ACTIVIDAD 1

Elabora un mapa conceptual del contenido del tema.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

ACTIVIDAD 2

Diseña un algoritmo para dar solución a un problema que tú propongas, en donde se utilice alguna de las estructuras de control: Mientras, Hasta que, Si entonces Sino y el contador Para.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

ACTIVIDAD 1

Elabora un cuadro comparativo de las diferentes técnicas de diseño de algoritmos. Envíalo en un documento a tu asesor.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma

ACTIVIDAD 2

Diseña un algoritmo voraz para solucionar el problema de dar cambio de dinero por la venta de diversos artículos en una tiendita. Envíalo en un documento a tu asesor.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

ACTIVIDAD 3

Investiga el juego de las torres de Hanoi y diseña las funciones recursivas necesarias para su ejecución. Envíalo en un documento a tu asesor.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

CUESTIONARIO DE REFORZAMIENTO

Contesta las siguientes preguntas:

Realiza tu actividad en un documento en Word, guárdala en tu computadora y una vez concluida, presiona el botón Examinar. Localiza tu archivo donde lo guardaste, selecciónalo y presiona Subir este archivo para guardarlo en la plataforma.

1. ¿Cuáles son las estructuras de ciclos?
2. ¿Qué diferencias existen entre las estructuras Mientras y Hasta que?
3. Dentro de una estructura *for* se puede utilizar una instrucción para cambiar el valor de la variable que utiliza la estructura para controlar las iteraciones. Indica la razón por la cual no debería cambiarse el valor a esta variable dentro de la misma estructura.
4. ¿Para qué tipo de problemas se utilizan los algoritmos voraces?
5. ¿Qué funciones utiliza un algoritmo voraz?
6. Explica el concepto de recursividad en la técnica Divide y vencerás.
7. En la programación dinámica, ¿qué se entiende por subestructura óptima?
8. ¿Cuál estrategia de diseño está relacionada con la búsqueda combinatoria?
9. ¿Qué tareas realizan los algoritmos *backtracking* cuando encuentran una solución candidata?

10. En un tablero de ajedrez de 8 x 8 casillas, la pieza denominada reina puede avanzar una o varias casillas en forma horizontal, vertical o diagonal. Si en su camino encuentra una pieza adversaria, la ataca. Entonces:

- ¿Cómo colocarías 8 reinas sobre el tablero sin que alguna reina ataque a la otra?
- ¿Cuál estrategia de diseño de algoritmos recomendarías para solucionar el problema de las ocho reinas?

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluida, presiona el botón Examinar. Localiza el archivo, ya seleccionado, presiona Subir este archivo para guardarlo en la plataforma

EXAMEN DE AUTOEVALUACIÓN 1

Responde si son verdaderas (V) o falsas (F) las siguientes aseveraciones. Una vez que concluyas, obtendrás tu calificación de manera automática.

	Verdadera	Falsa
1. Las condiciones son aquellas que hacen llamadas a sí mismas en su definición, simplificando los valores originales de entrada.	( )	( )
2. Los acumuladores se pueden implementar en los casos en que el problema que se desea resolver puede simplificarse en versiones más pequeñas del mismo.	( )	( )
3. Los contadores se caracterizan por iterar instrucciones en función de una condición que debe cumplirse en un momento conocido.	( )	( )
4. Los ciclos son estructuras que se caracterizan por iterar instrucciones en función de una condición que debe cumplirse en un momento bien definido.	( )	( )
5. Las rutinas recursivas regularmente contienen una clausula condicional.	( )	( )

EXAMEN DE AUTOEVALUACIÓN 2

Escribe la palabra que mejor complete la sentencia o responda la pregunta.

1.- Divide el problema en subproblemas que resuelve recursivamente para, finalmente, reunir las soluciones individuales. ________________________

2.- Resuelve el problema en suconjuntos, a través de subestructuras optimas.________________

3.- Técnica en que esencialmente encuentra la mejor combinación en un momento determinado (búsqueda en profundidad)

4.- Se utilizan en solución de problemas de optimización, aunque son un poco eficientes.__________________

Programación dinámica / Algoritmos voraces / Vuelta atrás / Divide y vencerás

EXAMEN DE AUTOEVALUACIÓN 3

Elige la opción que conteste correctamente cada pregunta.

1. Es característico de la estructura hasta que:
    - a) Evalúa una condición al principio de la estructura
    - b) Ejecuta las instrucciones y luego evalúa la condición
    - c) Si la condición evaluada resulta falsa, se sale de la estructura
    - d) La estructura al final no está delimitada por un comando.

2. Una variable del tipo acumulador es aquella que:
    - a) Se incrementa en cada iteración con la unidad
    - b) No sufre un cremento alguno, solo es de control
    - c) Aumenta su valor con el valor propio más el del incremento
    - c) No tiene algo que ver con la solución arrojada por el algoritmo.

3. Cuando se sabe con exactitud el número de iteraciones que debe realizar una estructura, se utiliza:
    - a) Para
    - b) Mientras
    - c) Hasta que
    - d) Si entonces si no

4. Estructura en la que se puede prescindir del conjunto de instrucciones de la condición falsa:

a) Para

b) Mientras

c) Hasta que

d) Si entonces si no

5. Técnica de diseño de algoritmos que contienen una función de factibilidad, una función de selección y una función objetivo

a) Algoritmos voraces

b) Divide y vencerás

c) Programación dinámica

d) Vuelta atrás

6. Si una rutina contiene dos llamadas recursivas, se denominan algoritmos de:

a) Algoritmos voraces

b) Divide y vencerás

c) Programación dinámica

d) Vuelta atrás

MESOGRAFÍA

BIBLIOGRAFÍA BÁSICA

- JOYANES, L. ***Estructuras de datos, algoritmos, abstracción y objetos***, México, McGraw Hill, 1998.
- LOZANO, L. ***Diagramación y programación estructurada y libre***, 3ª edición, México, McGraw-Hill, 1986, 384 pp.
- MANZANO, G. ***Tutorial para la asignatura Análisis, diseño e implantación de algoritmos***, Fondo editorial FCA, México, 2003.
- LEE, R. TSENG, S. CHANG, R. y TSAI, Y. ***Introducción al diseño y análisis de algoritmos un enfoque estratégico***, México, McGraw Hill, 2007, 752 pp.
- SEDGEWICK, R. ***Algoritmos en C++***, México, Pearson Education, 1995.
- VAN GELDER, B. ***Algoritmos computacionales Introducción al análisis y diseño***, 3ª ed. México, Thomson, 2002.

BIBLIOGRAFÍA COMPLEMENTARIA

- HERNÁNDEZ, Roberto, ***Estructuras de datos y algoritmos***, MÉXICO, PRENTICE HALL, 2000, 296 PP.

- JOYANES Aguilar Luis, Programación En C++, ***Algoritmos, estructuras de datos y objetos***, MÉXICO, MC.GRAW-HILL, 2000.
- VAN Gelder, Baase, ***Algoritmos Computacionales***, 3ª. ED., MÉXICO, THOMSON, 2003.
- OSVALDO Cairo Battistutti Aniei, ***Fundamentos de programación piensa en C***, 392 pag. (2006)

SITIOS ELECTRÓNICOS

http://www.lcc.uma.es/~av/Libro/indice.html, 25/Marzo/2011, Universidad de Málaga, Libro de las técnicas de diseño de algoritmos.

http://mis-algoritmos.com/aprenda-a-crear-diagramas-de-flujo, 25/Marzo/2011, Victor de la Rocha, Descripción de cómo crear diagramas de flujo.

http://prof.usb.ve/mvillasa/compcient/estructuras.pdf, 25/Marzo/2011, Universidad Simon Bolivar, Estructuras de control

UNIDAD 4

IMPLANTACIÓN DE ALGORITMOS

OBJETIVO ESPECÍFICO

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

INTRODUCCIÓN

En este tema se aborda el método para transformar un algoritmo a su expresión computable: el programa. Un programa 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, comparando sus ventajas y limitaciones.

LO QUE SÉ

Busca tres definiciones de programación. Escribe lo que será el inicio de tu protocolo de investigación Elabora tu actividad en Word y guárdala en tu computadora.

TEMARIO DETALLADO

(12 HORAS)

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.

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 debe implementar mediante la utilización de un programa de computadora.

El algoritmo debe transformarse, línea por línea, a la sintaxis utilizada por un lenguaje de programación (el lenguaje que seleccione el programador), por lo que revisaremos desde el principio la manera en que un algoritmo se convierte en un programa de computadora:

DEFINICIÓN DEL ALGORITMO
Es el 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 deben bosquejar los pasos a seguir por el algoritmo.

SELECCIÓN DE LA MEJOR ALTERNATIVA
Si hay varias formas de solucionar nuestro problema, se debe escoger la alternativa 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 lo guardamos en archivo electrónico.

COMPLILACIÓN
El compilador verifica la sintaxis del código fuente en busca de errores, es decir que algún comando o regla de puntuación del lenguaje la escribimos mal. Se depura y se 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, volveríamos al paso 6 para revisar el código fuente y depurarlo.

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

Una vez que se produce el archivo ejecutable, el programa se hace independiente del lenguaje de programación que se utilizó para generarlo, por lo que permite su portabilidad a otro sistema de cómputo.

El programa es, entonces, la expresión computable del algoritmo ya implementado, y puede utilizarse repetidamente en el área en donde se generó 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 los años sesenta 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 está basada 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 el programa.

Esto es importante debido a que es mucho más fácil 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*).

La programación estructurada tiene las siguientes ventajas:

* 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 las 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 [3], de la forma: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 y 89, es un ejemplo de la programación estructurada:

[3] La serie de Fibonacci se define como la serie en que el tercer número es el resultado de la suma de los dos números anteriores a éste.

<table>
<tr><th>PSEUDOCÓDIGO</th><th>PROGRAMA FUENTE
EN LENGUAJE C.</th></tr>
<tr><td>

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.

</td><td>

#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();
}

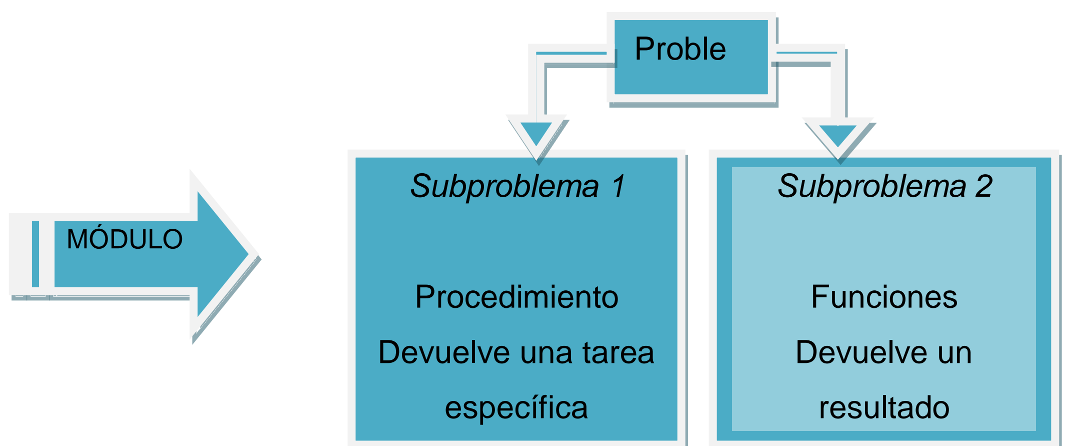
</td></tr>
</table>

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

4.3 Modularidad.

Un problema se puede dividir en subproblemas más sencillos. Estos subproblemas se conocen como módulos.

Dentro de los programas se les conoce como subprogramas, y de estos hay dos tipos: los procedimientos y las funciones. Ambos reciben datos del programa que los invoca, donde los primeros devuelven una tarea específica y las funciones un resultado:



Los procedimientos, en los nuevos lenguajes de programación, cada vez se utilizan menos, por lo que la mayoría de lenguajes de programación utilizan en mayor medida las funciones. Un ejemplo de un lenguaje de programación construido únicamente por funciones, es el lenguaje C.

Cuando un procedimiento o una función se invocan a sí mismos, se le llama recursividad (tema ya tratado con anterioridad.

4.4 Funciones, Rutinas y Procedimientos

Función

Una función es un conjunto de pasos para realizar cálculos específicados y devolver siempre un resultado, los pasos están almacenados bajo un nombre de función, la cual acepta ciertos valores conocidos como argumentos para realizar cálculos con estos y devolver un resultado. Hay funciones que carecen de argumentos, pero que de cualquier forma devuelven un resultado.

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

Una función puede invocar a otra función e inclusive tener la capacidad de invocarse a sí misma como en el caso de 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 cálcula 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

Una rutina es un algoritmo que realiza una tarea específica y que puede ser invocado desde otro algoritmo para realizar 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 esta.

Procedimiento

Un procedimiento es similar a una función, la única diferencia es que este no regresa un resultado sino que en su lugar realiza una o varias tareas, como puede ser el de centrar una cadena en la pantalla de la computadora, dibujar un marco, imprimir un mensaje etcétera, veámos el siguiente ejemplo de un procedimiento:

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

Como se aprecia, el procedimiento realiza las tareas de borrar la pantalla e imprimir un mensaje de bienvenida pero sin devolver un valor como resultado como en el caso de las funciones.

4.5 Enfoque de algoritmos.

Existen dos enfoques que se refieren a la forma en que se diseña un algoritmo, los cuales son refinamiento progresivo y procesamiento regresivo, veamos a qué se refiere cada uno de estos:

Refinamiento progresivo

Es una técnica de análisis y diseño de algoritmos que se basa en la división del problema principal en problemas más simples. Partiendo de problemas más simples se logra dar una solución más efectiva, ya que el número de variables y casos asociados a un problema simple es más fácil de manejar que el problema completo.

Esta técnica se conoce como *Top-Down,* y es aplicable a la optimización del desempeño y a la simplificación de un algoritmo.

T*op Down* (arriba - abajo)

La técnica *top down*, o diseño descendente como también se le conoce, consiste en establecer una serie de niveles de mayor a menor complejidad (arriba-abajo), que den solución al algoritmo. Consiste en efectuar 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. Este diseño consiste en una serie de descomposiciones sucesivas del problema inicial, que recibe el refinamiento progresivo del repertorio de instrucciones que van a formar parte del programa.

La utilización de 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, 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.

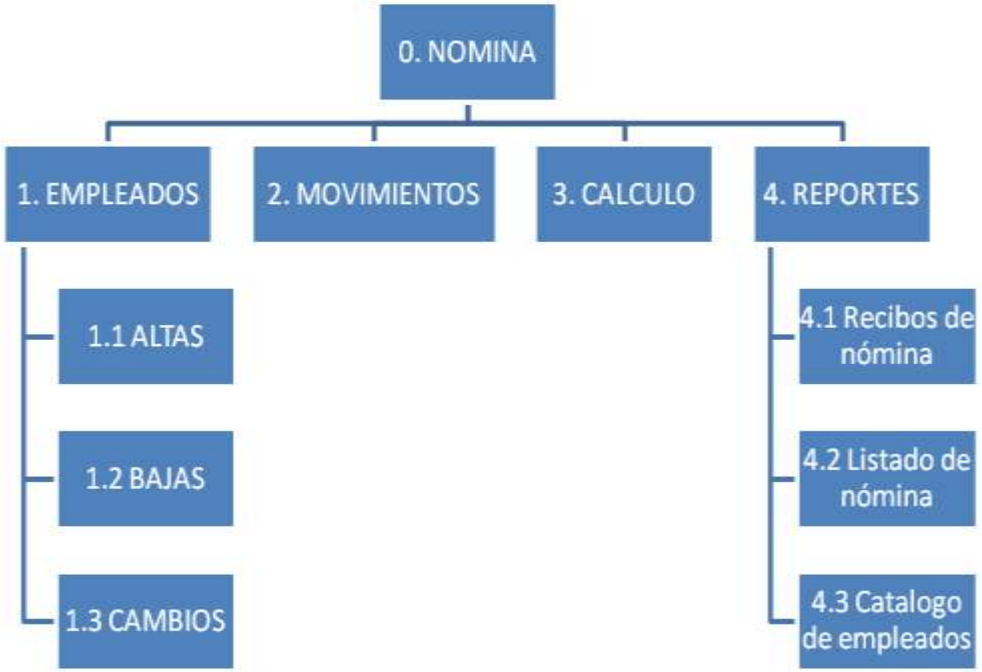
- El módulo 1 de empleados contendrá un submenú con las opciones de altas, bajas y los cambios a los registros de los empleados.

- En el módulo 2 se capturarán los movimientos quincenales de la nómina como los días trabajados, horas extra, faltas, incapacidades, etcétera, de los empleados.
- En el módulo 3 se realizarán los cálculos de las percepciones, deducciones y el total de la nómina, individualizado por trabajador y, por último, el menú de reportes con el número 4, que contendrá 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.

El objetivo es gestionar los movimientos de la nómina por trabajador e imprimir los reportes correspondientes. Teniéndolo en mente se fue descomponiendo en los distintos módulos y submódulos que conforman al sistema.

Procesamiento regresivo

Ésta es otra técnica de análisis y diseño de algoritmos, que parte de la existencia de múltiples problemas y se enfoca en la asociación e identificación de características comunes entre ellos, para diseñar un modelo que represente la solución para todos los casos, de acuerdo con ciertas características específicas de las entradas. Esta técnica también es conocida como *Bottom-Up,* aunque suele pasar que no alcance la integración óptima y eficiente de las soluciones de los diversos problemas.



Bottom Up (abajo-arriba)

Es el diseño ascendente que se refiere 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 haciendo un enfoque ascendente, es difícil llegar a integrar los subalgoritmos al grado tal 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 contempla el programa final, éste adolece de ciertas limitaciones por haber tomado un enfoque ascendente:

- Hay duplicación de esfuerzos al introducir los datos.
- Se introducen 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 alcanza 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.

La ventaja del diseño ascendente es que su desarrollo es mucho más económico que el diseño 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 DELA UNIDAD

En esta unidad se estudió al programa, definiéndolo como un conjunto de instrucciones que realizan acciones especificas, escritas en un lenguaje de programación. De ahí la importancia de haber abordado en esta unidad el método para transformar un algoritmo a su expresión computable, el programa.

A partir de ahí se estudiaron las estructuras de control básicas, referidas al Teorema de la Estructura, que son fundamentales en la programación estructurada que, a diferencia de la programación libre, no realiza bifurcaciones lógicas a otro punto, lo cuál facilita su seguimiento y mantenimiento.

GLOSARIO DE LA UNIDAD

Iteración

En programación es la repetición de una serie de instrucciones en un programa de computadora. Puede usarse como un término genérico (como sinónimo de repetición), así como para describir una forma específica de repetición con un estado mutable.

Compilación

El compilador verifica la sintaxis del código fuente en busca de errores, es decir que algún comando o regla de puntuación del lenguaje la escribimos mal.

Modularidad

Es la característica por la cual un programa de computadora está compuesto de partes separadas a las que llamamos módulos.

ACTIVIDADES DE APRENDIZAJE

ACTIVIDAD 1

Realiza un diagrama de flujo de la manera en que un algoritmo se convierte en un programa de computadora

Realiza tu actividad en un procesador de textos, guardala en tu computadora y una vez que concluyas, presiona el botón **Examinar**. Localiza el archivo ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

ACTIVIDAD 2

Investiga qué otras estructuras de control existen que se deriven de las básicas explicadas en este apunte.

Pulsa el botón **Colocar un nuevo tema de discusión aquí**.
Escribe en el apartado **Asunto** el título de tu aportación, redacta tu comentario en el área de texto y da clic en el botón **Enviar al foro.**

ACTIVIDAD 3

Desarrolla un diagrama *top down* y uno de *bottom up* para un sistema de inventarios.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma

ACTIVIDAD 4

Investiga en una empresa que tipo de enfoque para desarrollar sus sistemas de información. Sube al foro tus comentarios.

Pulsa el botón **Colocar un nuevo tema de discusión aquí.**

Escribe en el apartado **Asunto** el titulo de tu aportación, redacta tu comentario en el área de texto y da clic en el botón **Enviar al foro.**

CUESTIONARIO DE REFORZAMIENTO

Contesta las siguientes preguntas:

Realiza tu actividad en un documento en Word, guárdala en tu computadora y una vez concluida, presiona el botón Examinar. Localiza tu archivo donde lo guardaste, selecciónalo y presiona Subir este archivo para guardarlo en la plataforma.

1. ¿Qué entiendes por una prueba de escritorio?
2. ¿Qué es un compilador?
3. ¿Qué es un diccionario de datos?
4. Define la expresión "el programa como la expresión computable del algoritmo".
5. ¿Cuáles son las estructuras de control básicas?
6. ¿Qué es lo que establece el Teorema de la Estructura?
7. Enuncia 5 ventajas de la programación estructurada.
8. Define la Modularidad.
9. ¿Qué entiendes por refinamiento progresivo?
10. ¿Qué es el procesamiento regresivo?

LO QUE APRENDÍ

Retoma el apartado “*Lo que sé*” y complementa la definición que le has dado a programación, y anexa sus derivaciones.

Pulsa el botón **Iniciar o editar mi entrada de diario**, escribe lo que será el inicio de tu protocolo de investigación. Si deseas borrar algo de lo que hasta el momento llevas escrito, pulsa el botón **Revertir**. Cuando decidas concluir tu trabajo del día, pulsa el botón **Guardar cambios**.

EXAMEN DE AUTOEVALUACIÓN 1

Relaciona las columnas colocando el número en el espacio que le corresponde

Cargar datos muestra y seguir la lógica marcada por el diagrama o el pseudocódigo	1 Compilación 2 Codificación 3 Prueba de escritorio 4 Prueba del programa 5 Diseño del algoritmo
Se diagraman los pasos del problema	
Verifica la sintaxis del código fuente en busca de errores.	
Se ingresan datos muestra para el análisis de los resultados	
Traduce cada grafico del diagrama o línea del pseudocódigo a una instrucción de algún lenguaje de programa	

EXAMEN DE AUTOEVALUACIÓN 2

Relaciona las columnas colocando el número en el espacio que le corresponde:

	Estructuras de control básicas	1. Productividad del programador
	Ventajas de la programación estructurada	2. Teorema de la estructura
	Una característica importante en un programa estructurado	3. Iteración, selección, secuencia.
	La programación estructurada está basada en …	4. Ser leído en secuencia.
	Serie de instrucciones que se ejecutan sucesivamente	5. Secuencia

EXAMEN DE AUTOEVALUACIÓN 3

Responde si son verdaderas (V) o falsas (F) las siguientes aseveraciones. Una vez que concluyas, obtendrás tu calificación de manera automática.

	Verdadera	Falsa
1. Los procedimientos devuelven un resultado.	( )	( )
2. Las funciones devuelven una tarea específica.	( )	( )
3. Los procedimientos son los más usados en los lenguajes de programación.	( )	( )
4. Las funciones son los más usados en los lenguajes de programación.	( )	( )
5. El refinamiento progresivo se enfoca en la asociación e identificación de características comunes entre ellos para diseñar un modelo que represente la solución para todos los casos.	( )	( )
6. La técnica top down, o diseño descendiente como también se le conoce, consiste en establecer una serie de niveles de mayor a menor complejidad.	( )	( )
7. El procesamiento regresivo es una técnica de análisis y diseño de algoritmos que se basa en la división del problema principal en problemas más simples.	( )	( )

8. La ventaja del diseño ascendente es que su desarrollo es mucho más económico que el diseño descendiente.	( )	( )
9.- El diseño ascendente identifica subalgoritmos que necesitan computarizarse conforme vayan apareciendo, su análisis y su codificación, para satisfacer el problema inmediato.	( )	( )

EXAMEN DE AUTOEVALUACIÓN 4

Responde si son verdaderas (V) o falsas (F) las siguientes aseveraciones. Una vez que concluyas, obtendrás tu calificación de manera automática.

	Verdadera	Falsa
1. La compilación es un programa para convertir un código fuente a un programa ejecutable.	( )	( )
2. Si en las pruebas del programa se detectan errores, solo se tienen que volver a compilar el programa.	( )	( )
3. Las estructuras MIENTRAS y HASTA QUE, son estructuras condicionales iterativas.	( )	( )
4. El teorema de la estructura solo hace referencia a las estructuras de control de secuencia, selección e iteración.	( )	( )

5.- Un programa estructurado contiene instrucciones de desvió del flujo de control.	( )	( )
6.- Solo hay un tipo de modulo, y es la función	( )	( )
7.- Un procedimiento devuelve una tarea y una función, un resultado.	( )	( )
8.- El refinamiento progresivo contiene al procedimiento mas costoso, pero el más eficiente para integrar los módulos de un sistema.	( )	( )
9.- El Bottom Up tiene la limitación de duplicar esfuerzos al introducir datos, ya que se introducen al sistema muchos datos carentes de valor.	( )	( )

MESOGRAFÍA

SITIOS DE INTERÉS

http://sistemas.itlp.edu.mx/tutoriales/pascal/u1_1_4.html, 25/Marzo/2011, Departamento de Sistemas y Computación del Instituto Tecnológico de La Paz, tutorial de la programación estructurada.

http://www.mailxmail.com/curso-aprende-programar/modularidad-procedimientos-funciones, 25/Marzo/2011, Mailxmail, S.L., Capítulo sobre modularidad, procedimientos y funciones.

http://www.desarrolloweb.com/articulos/2183.php, 25/Marzo/2011, Desarrollo web.com, Artículo sobre las técnicas de diseño Top Down y Bottom Up.

BIBLIOGRAFÍA BÁSICA

- DE GIUSTI, A. ***Algoritmos, datos y programas con aplicaciones en Pascal, Delphi y Visual Da Vinci***, Buenos Aires, Pearson Education, 2001, 472 pp.
- JOYANES, L. ***Estructuras de datos, algoritmos, abstracción y objetos***, México, McGraw Hill, 1998.
- LOZANO, L. ***Diagramación y programación estructurada y libre***, 3ª edición, México, McGraw-Hill, 1986, 384 pp.
- MANZANO, G. ***Tutorial para la asignatura Análisis, diseño e implantación de algoritmos***, Fondo editorial FCA, México, 2003.

- LEE, R. TSENG, S. CHANG, R. y TSAI, Y. ***Introducción al diseño y análisis de algoritmos un enfoque estratégico***, México, McGraw Hill, 2007, 752 pp.
- SEDGEWICK, R. ***Algoritmos en C++***, México, Pearson Education, 1995.
- **BIBLIOGRAFÍA COMPLEMENTARIA**
- HERNÁNDEZ, Roberto, ***Estructuras de datos y algoritmos***, MÉXICO, PRENTICE HALL, 2000, 296 PP.
- JOYANES Aguilar Luis, Programación En C++, ***Algoritmos, estructuras de datos y objetos***, MÉXICO, MC.GRAW-HILL, 2000.
- OSVALDO Cairo Battistutti Aniei, ***Fundamentos de programación piensa en C***, 392 pag. (2006)

UNIDAD 5

EVALUACIÓN DE ALGORITMOS

OBJETIVO ESPECÍFICO

Al finalizar la unidad, el alumno 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.

INTRODUCCIÓN

La evaluación de algoritmos es un proceso de análisis de desempeño del tiempo de ejecución que éste tarda en encontrar una solución, y la cantidad de recursos empleados para ello.

Entre las técnicas más confiables se encuentran aquéllas que miden la complejidad de algoritmos a través de funciones matemáticas.

Se abordará la depuración y prueba de programas, con el fin de asegurar que estén libres de errores y que 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 los

programas y dejar una evidencia de todas las especificaciones del programa.

Los programas deben ser depurados para cumplir en forma eficaz con su objetivo, para ello se les debe dar el adecuado mantenimiento. Con este propósito se analizarán los diferentes tipos de mantenimiento: preventivo, correctivo y adaptativo.

LO QUE SÉ

Elabora un cuadro sinóptico con las principales características de un algoritmo, haciendo énfasis la evaluación.

Realiza tu actividad en un procesador de textos, guárdala en tu computadora.

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.

5.1 Refinamiento progresivo.

En el tema anterior ya se había tocado el tema de refinamiento progresivo, que es la descomposición de un problema en *n* problemas para facilitar su solución, y al final integrar éstas en una solución global.

Lo que nos concierne en este tema es la evaluación de los algoritmos con el fin de medir su eficiencia.

La evaluación de un algoritmo tiene como propósito medir su desempeño, considerando el tiempo de ejecución y los recursos empleados (memoria de la computadora) para obtener una solución satisfactoria.

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, y es aquí en donde entran otros factores como el lenguaje de programación elegido, el sistema operativo empleado, la habilidad del programador, etcétera.

Pero existe 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, en donde el peor caso es que las entradas se encuentren totalmente desordenadas, el mejor de los casos es que se encuentren totalmente ordenadas y en el caso promedio están parcialmente ordenadas. Veamos como ejemplo a 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
```

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. Este arreglo puede estar inicializado de forma creciente, decreciente o aleatoria.

El peor caso ocurre cuando el arreglo está inicializado descendentemente.

El mejor caso ocurre cuando el arreglo está inicializado ascendentemente (en este caso 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) cuando el arreglo está inicializado en orden decreciente, y también cuando está inicializado 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
```

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. Este arreglo puede estar inicializado de forma creciente, decreciente o aleatoria.

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

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

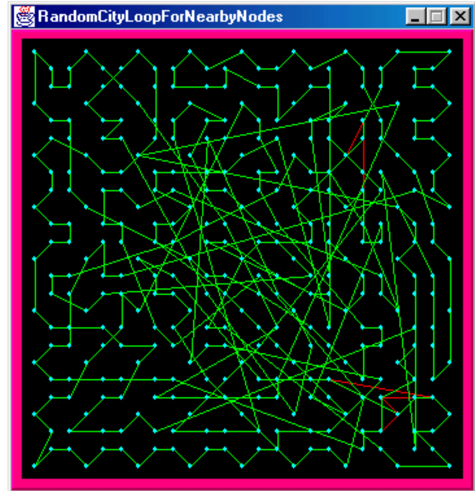
Se ha calculado empíricamente la complejidad para este algoritmo y se ha obtenido que, para cualquier inicialización del arreglo de datos, el algoritmo tenga una complejidad cuadrática O (n2).

5.2 Depuración y prueba.

Depuración

Es el proceso de identificación y corrección de errores de programación.

El término en inglés es *debugging*, que significa eliminación de bichos. Una anécdota sobre el origen de este término es que, en la época de la primer generación de computadoras constituidas por bulbos, encontraron una polilla entre los circuitos que era la responsable de la falla del equipo, y de ese hecho nació el término para indicar que el equipo o los programas presentan algún problema.



Para depurar el código fuente, el programador se vale de herramientas de software que le facilitan la localización y depuración de errores. Los compiladores son un ejemplo de estas herramientas.

Se dice que un programa está depurado cuando 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 tipo de error, tanto en tiempo de programación como en tiempo de ejecución.

Los errores más sencillos de detectar son los errores de sintaxis, que se presentan cuando alguna instrucción está mal escrita o se omitió alguna puntuación necesaria para el programa. Existen también los errores lógicos, en los que, aunque el programa no contenga errores de sintaxis, no realiza el objetivo por el que fue desarrollado. Pueden presentarse errores en los valores de las variables, ejecuciones de programa que no terminan, errores 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 produce 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.

Una vez terminada la programación, el analista a cargo del sistema volverá a usar los datos de prueba para verificar que el programa o sistema produce los resultados correctos.

En esta ocasión, el analista concentrará su atención también en la interacción correcta entre los diferentes programas y el funcionamiento completo del sistema. Se verificarán:

- Todos los registros que se incluyen 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 cambio que surja como resultado de la programación, deberá ser expuesto y aceptado antes de aplicar el cambio.

Nombre del programa (código). Indicará código que identifica el programa y el título del mismo
Descripción. Indicará 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 y copia, y descripción de los archivos.

Lista de informes y/o totales de control. Se indicará 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 que efectuó el cambio, según sea el caso.

Fecha. Indicará la fecha en que se escribió el programa o que se efectuó el cambio, según sea el caso.

Diccionario de datos. En caso que aplique, se incluirá detalle de las diferentes tablas y códigos usados con los valores, explicaciones y su uso en el programa.

Lista de programas. Deberá incluir copia de la última compilación del programa con todas las opciones.

5.4 Mantenimiento de programas.

Objetivo

Identificar la importancia del mantenimiento de programas y las características del preventivo, correctivo y adaptativo.

Desarrollo

Los usuarios de los programas solicitarán los cambios necesarios al área de sistemas, con el fin de que los programas sigan operando correctamente. Para ello, periódicamente se le debe dar el mantenimiento que requieren los programas, el cual puede ser de tres tipos:

PREVENTIVO
Los programas no presentan error alguno, 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 depurarlo y compilarlo. Se deben realizar las pruebas al sistema, imprimiendo los reportes que generan y verificar si los cálculos que éstos presentan son correctos.

ADAPTATIVO
Los programas no tienen error alguno, pero 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, llenando por escrito el tipo de mantenimiento que requiere y remitiéndolo al área de sistema para su revisión y valoración.

El personal del área de sistemas hará una orden de trabajo para proceder a realizar el servicio solicitado.

RESUMEN DE LA UNIDAD

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 abordó la depuración y prueba de programas con el fin de asegurar que estén libres de errores y que cumplan eficazmente con el objetivo para el que fueron elaborados.

También se revisó la documentación de los programas para que, tanto analistas como programadores conozcan la función y el fin para los que fueron creados, y así tener un archivo con las especificaciones del programa. Además, se tocó el tema del mantenimiento, específicamente el preventivo, correctivo y adaptativo, ya que son fundamentales para el buen funcionamiento y operación de un algoritmo.

GLOSARIO DE LA UNIDAD

Depuración

Es el proceso de identificación y corrección de errores de programación.

Documentación de programas

El programa libre de errores se documenta, incluyendo los diagramas utilizados, el listado de su código fuente, el diccionario de datos en donde se listarán las variables, constantes, arreglos, abreviaciones utilizadas, etcétera. El objetivo de la documentación es familiarizar a analistas y programadores con lo que hace cada programa en particular.

ACTIVIDADES DE APRENDIZAJE

ACTIVIDAD 1 Elaborar un cuadro comparativo de evaluación de métodos de ordenación, para determinar su eficiencia con base en la complejidad de sus algoritmos. Realiza tu actividad en un procesador de textos, guardala en tu computadora y una vez que concluyas, presiona el botón **Examinar**. Localiza el archivo ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

ACTIVIDAD 2 Elabora un mini manual para documentar programas. Realiza tu actividad en un procesador de textos, guárdala en tu computadora y una vez concluyas, presiona el botón **Examinar**. Localiza el archivo, ya seleccionado, presiona **Subir este archivo** para guardarlo en la plataforma.

ACTIVIDAD 3

Investiga en una empresa el procedimiento para llevar a cabo el mantenimiento de programas.

Pulsa el botón **Colocar un nuevo tema de discusión aquí**.

Escribe en el apartado **Asunto** el título de tu aportación, redacta tu comentario en el área de texto y da clic en el botón **Enviar al foro**.

CUESTIONARIO DE REFORZAMIENTO

Contesta las siguientes preguntas:

Realiza tu actividad en un documento en Word, guárdala en tu computadora y una vez concluida, presiona el botón Examinar. Localiza tu archivo donde lo guardaste, selecciónalo y presiona Subir este archivo para guardarlo en la plataforma.

1. ¿Qué significa la evaluación de algoritmos?
2. En la forma en que se encuentran los datos de entrada a un algoritmo ¿qué significa el peor caso, el mejor caso y el caso promedio?
3. ¿Qué se entiende por depuración de programas?
4. ¿Cuáles son los errores de sintaxis y los errores lógicos?
5. Define la prueba de programas.
6. Enlista 5 elementos que se verifican en la prueba de programas.
7. Enlista 5 elementos que se deben incluir en la documentación de un programa.
8. ¿Para qué sirve el mantenimiento de programas?
9. Qué entiendes por mantenimiento preventivo?
10. ¿Qué es el mantenimiento correctivo?

EXAMEN DE AUTOEVALUACIÓN 1

Relaciona las columnas colocando el número en el espacio que le corresponde

Es el proceso de identificación y corrección de errores de programación	1 Sintaxis
Tiene como propósito asegurar que el programa produce los resultados definidos en las especificaciones funcionales	2 Depuración
Los errores más sencillos de detectar son los errores de …	3 Prueba de programas
Los errores más difíciles de detectar son los errores de …	4 Cálculo

EXAMEN DE AUTOEVALUACIÓN 2

Relaciona las columnas colocando el número en el espacio que le corresponde

Es una extensión de la documentación del sistema	1 Lista de Programas
El programador debe trabajar conjuntamente con ella	2 Documentación de programas
Incluye copia de la ultima compilación del programa con todas las opciones	3 Especificación de programas

EXAMEN DE AUTOEVALUACIÓN 3

Relaciona las columnas colocando el número en el espacio que le corresponde

Los programas no tienen error alguno, pero se requiere alguna actualización por una nueva versión del programa, esto se refiere al mantenimiento:	Preventivo
La revisión, la codificación para depurar y compilar un programa se realiza en el mantenimiento:	Adaptativo
La regeneración de índices, de registros, realizar respaldos, verificar la integridad de los programas, actualizar porcentajes y tablas de datos, son actividades del mantenimiento:	Correctivo

EXAMEN DE AUTOEVALUACIÓN 4

Responde si son verdaderas (V) o falsas (F) las siguientes aseveraciones. Una vez que concluyas, obtendrás tu calificación de manera automática.

	Verdadera	Falsa
1. El algoritmo de selección va intercambiando elementos del arreglo hasta que este ordenado.	( )	( )
2. El algoritmo de inserción va seleccionado los elementos menores al actual y los intercambia.	( )	( )
3. El propósito e la evaluación de un algoritmo es medir su desempeño.	( )	( )
4. En la evaluación del algoritmo solamente debe considerarse el tiempo de proceso.	( )	( )
5. Para medir la complejidad de un algoritmo no es necesario utilizar funciones matemáticas.	( )	( )
6. El termino debugging significa eliminación de bichos.	( )	( )
7. Un error lógico es cuando un programa tiene errores de sintaxis.	( )	( )
8.- El compilador es un programa que facilita la detección y corrección de errores.	( )	( )
9.- Para realizar pruebas al programa se deben utilizar cualquier tipo de datos, tanto correctos como incorrectos	( )	( )
10.- En las pruebas al programa se deben verificar todos los cálculos que el programa realice.	( )	( )
11.- El objetivo de la documentación de programas es familiarizar al usuario final con lo que hacen los programas.	( )	( )
12.- El programador puede aplicar su criterio para cualquier cambio que se presente en las especificaciones del programa	( )	( )
13.- No es necesario incluir el diccionario de datos en la documentación de programas.	( )	( )

MESOGRAFÍA

SITIOS DE INTERÉS

http://www.uprb.edu/politicas/Manual-Programacion.pdf, 25/Marzo/2011, Universidad de Puerto Rico, Documento sobre normas de documentación y mantenimiento de programas.

http://informatica.uv.es/iiguia/TP/teoria/tema1.pdf, 25/Marzo/2011, Universidad de Valencia, Apunte sobre la documentación de programas.

BIBLIOGRAFÍA BÁSICA

- DE GIUSTI, A. ***Algoritmos, datos y programas con aplicaciones en Pascal, Delphi y Visual Da Vinci***, Buenos Aires, Pearson Education, 2001, 472 pp.
- MANZANO, G. ***Tutorial para la asignatura Análisis, diseño e implantación de algoritmos***, Fondo editorial FCA, México, 2003.

- LEE, R. TSENG, S. CHANG, R. y TSAI, Y. ***Introducción al diseño y análisis de algoritmos un enfoque estratégico***, México, McGraw Hill, 2007, 752 pp.
- SEDGEWICK, R. ***Algoritmos en C++***, México, Pearson Education, 1995.
-

BIBLIOGRAFÍA COMPLEMENTARIA

- VAN Gelder, Baase, ***Algoritmos Computacionales***, 3ª. ED., MÉXICO, THOMSON, 2003.
- OSVALDO Cairo Battistutti Aniei, ***Fundamentos de programación piensa en C***, 392 pag. (2006)

UNIDAD 1			
E1	E2	E3	E4
1.- F 2.- F	3. Alfabeto 1. Frase 4. Cadena vacía 5. Lenguaje 2. Gramática	1. F 2. F 3. F 4.V 5. F	1. d 2. d 3. a 4. b 5. b

UNIDAD 2		
E1	E2	E3
1. V 2. F 3. V 4. F 5. F	2. Computabilidad 1. Problema de decisión 4. Recursivo 3. Indecidible 5. Computables	2. Selección 5. Quicksort 3. Shell 1. Burbuja 4. Inserción

UNIDAD 4			
E1	E2	E3	E4
3. Prueba de escritorio 5. Diseño del algoritmo 1. Compilación 4. Prueba del programa 2. Codificación	3. Iteración, selección, secuencia. 1. Productividad del programador 4. Ser leído en secuencia. 2. Teorema de la estructura 5. Secuencia	1- F 2. F 3. F 4. V 5. F 6. V 7. F 8. V 9. V	1. F 2. F 3. V 4. V 5. F 6. F 7. V 8. V 9. V

ANEXO

1. ¿Qué puede hacer un computador? Esta área de estudio se denomina "decidibilidad", denominándose "decidibles" los problemas que puede resolver un computador. Este tema se estudia en el Capítulo 9.

2. ¿Qué puede hacer un computador eficientemente? Esta área de estudio se denomina "intratabilidad", recibiendo el nombre de "tratables" los problemas que un computador puede resolver en un tiempo proporcional a alguna función que crezca lentamente con el tamaño de la entrada. Se supone que las funciones polinómicas son de "crecimiento lento", mientras que se considera que las funciones que crecen más rápido que cualquier función polinómica crecen con excesiva rapidez. Este tema se estudia en el Capítulo 10.

1.2 Introducción a las demostraciones formales

Si ha estudiado geometría plana en el bachillerato antes de la década de 1990, lo más probable es que haya tenido que realizar con detalle algunas "demostraciones deductivas", con las que se demuestra la validez de una afirmación mediante una secuencia detallada de pasos y razonamientos. En este caso, la geometría tiene claras aplicaciones prácticas (por ejemplo, si se desea comprar la cantidad de moqueta adecuada para una habitación, es necesario conocer la regla que se utiliza para calcular el área de un rectángulo), pero el conocimiento de las metodologías utilizadas para la realización de demostraciones formales era razón suficiente para que se estudiara en el bachillerato esta rama de las matemáticas.

Durante la década de 1990, en Estados Unidos se hizo popular la enseñanza de los métodos de demostración como algo relacionado con la impresión personal de cada uno acerca de los enunciados. Aunque es conveniente formarse alguna impresión acerca del enunciado que se va a utilizar, el hecho es que en los colegios se ha abandonado la enseñanza de importantes técnicas de demostración. Sin embargo, es necesario que los informáticos entiendan las demostraciones. Algunos informáticos mantienen una postura radical al respecto, sosteniendo que la escritura de un programa debe ir de la mano de una demostración formal de la verificación del programa. Dudamos de que esa postura sea productiva. Por otra parte, también hay quien mantiene que las demostraciones no tienen sentido en la disciplina de la programación. En este entorno, a menudo se hace referencia al lema: "Si no estás seguro de que tu programa funcione, ejecútalo y compruébalo".

Nuestra postura se encuentra entre ambos extremos. Ciertamente, es esencial que los programas se prueben. Sin embargo, la realización de pruebas solo llega hasta cierto punto, dado que un programa no se puede probar con todas las entradas posibles. Aún más importante: el programa puede ser complejo, con recursividades o iteraciones delicadas. En esos casos, si no se comprende lo que sucede antes y después de un bucle, o en una llamada recursiva a una

función, es poco probable que se pueda escribir correctamente el código correspondiente. Cuando las pruebas indican que el código es incorrecto, todavía queda el trabajo de corregirlo.

Para conseguir que una iteración o recursión sea correcta, hace falta establecer una hipótesis inductiva, y es útil razonar, formal o informalmente, acerca de si dicha hipótesis es consistente con la iteración o recursión sobre la que se trabaja. Este proceso, que lleva a la comprensión del funcionamiento de un programa correcto, es esencialmente el mismo que el de la demostración de teoremas por inducción. Por ello, es tradicional que una parte de los cursos de teoría de autómatas se dedique a métodos formales de demostración, lo que además proporciona modelos útiles para el desarrollo de ciertos tipos de software. La teoría de autómatas, quizá más que cualquier otra materia de las que conforman el núcleo de la informática, se presta a demostraciones naturales e interesantes, tanto de tipo *deductivo* (secuencias de pasos justificados) como de tipo *inductivo* (demostraciones recursivas de una afirmación parametrizada, que se basa en la misma afirmación para valores "más bajos" de dicho parámetro).

1.2.1 Demostraciones deductivas

Como se ha mencionado anteriormente, una demostración deductiva consiste en una secuencia de afirmaciones o proposiciones, cuya validez nos conduce a una *conclusión* a partir de unas proposiciones iniciales, denominadas *hipótesis* o *postulados*. Cada paso de la demostración debe deducirse mediante algún principio lógico, bien de los postulados, bien de algunas de las proposiciones obtenidas previamente, bien de una combinación de ambas cosas.

La hipótesis puede ser verdadera o falsa, hecho que depende normalmente de los valores que se asignen a sus parámetros. A menudo, la hipótesis está formada por varias afirmaciones independientes que se relacionan entre sí mediante el operador lógico Y. En estos casos, se dice que cada una de las proposiciones constituye una hipótesis o un postulado.

El teorema que se demuestra cuando es posible llegar desde una hipótesis H a una conclusión C, es la proposición "si H, entonces C." Se dice entonces que C se *deduce* de H. Veamos un ejemplo de teorema de la forma "si H, entonces C", que servirá para ilustrar lo anterior.

Teorema 1.3: Si $x \geq 4$, entonces $2^x \geq x^2$. □

De manera informal, es fácil convencerse de que el Teorema 1.3 es verdadero, aunque la demostración formal requiere utilizar el método de inducción, y se pospondrá hasta el Ejemplo 1.17. En primer lugar, obsérvese que la hipótesis H es "$x \geq 4$". Esta hipótesis tiene un parámetro x, de modo que no es ni verdadera ni falsa. Más bien se puede decir que su validez depende del valor del parámetro x; por ejemplo, H es verdadera para $x = 6$, y falsa para $x = 2$.

Del mismo modo, la conclusión C es "$2^x \geq x^2$". Esta afirmación también utiliza el parámetro x, y es verdadera para ciertos valores de x y falsa para otros. Por ejemplo, C es falsa para $x = 3$, ya que $2^3 = 8$, que no es mayor o igual que

$3^2 = 9$. Por otra parte, C es verdadera para $x = 4$, ya que $2^4 = 4^2 = 16$. Para $x = 5$, la proposición también es verdadera, ya que $2^5 = 32$ es al menos tan grande como $5^2 = 25$.

Quizá ya se puede ver cuál es el argumento intuitivo que nos indica que la conclusión $2^x \geq x^2$ será verdadera siempre y cuando $x \geq 4$. Ya se vio que es verdadera para $x = 4$. Cuando x adquiere valores mayores que 4, la parte izquierda, 2^x, se multiplica por 2 cada vez que x aumenta en una unidad. Sin embargo, la parte derecha, x^2, aumenta en la proporción $\left(\frac{x+1}{x}\right)^2$. Si $x \geq 4$, $(x+1)/x$ no puede ser mayor que 1,25. Por tanto, $\left(\frac{x+1}{x}\right)^2$ no puede ser mayor que 1,5625. Dado que $1,5625 < 2$, cada vez que x aumenta de valor por encima de 4, la parte izquierda, 2^x, crece más que la parte derecha, x^2. Por tanto, siempre que se parta de un valor como $x = 4$, para el cuál la desigualdad $2^x \geq x^2$ se satisface, es posible incrementar x tanto como se desee, y la desigualdad continuará satisfaciéndose.

Acabamos de realizar una demostración informal, pero correcta, del Teorema 1.3. Volveremos a esta demostración y la realizaremos de forma más precisa en el Ejemplo 1.17, después de introducir las demostraciones "inductivas".

El Teorema 1.3, como todos los teoremas de interés, supone la existencia de un número infinito de hechos relacionados, en este caso la afirmación de que "si $x \geq 4$, entonces $2^x \geq x^2$" para cualquier entero x. De hecho, no es necesario suponer que x sea entero, pero durante la demostración se mencionó repetidamente que x se incrementaba en una unidad, a partir de $x = 4$, así que, en realidad, solo hemos abordado el problema para los casos en que x es un número entero.

El Teorema 1.3 se puede utilizar como ayuda para deducir otros teoremas. En el siguiente ejemplo se estudia la demostración deductiva completa de un teorema sencillo, en la que se utiliza el Teorema 1.3.

Teorema 1.4: Si x es la suma de los cuadrados de cuatro números enteros positivos, entonces $2^x \geq x^2$.

PRUEBA: La idea intuitiva de esta demostración es que, si la hipótesis es verdadera para x (es decir, si x es la suma de los cuadrados de cuatro números enteros positivos), entonces x debe valer al menos 4. Por lo tanto, la hipótesis del Teorema 1.3 se cumple. Dado que creemos que dicho teorema es verdadero, podemos afirmar que su conclusión también es verdadera para ese valor de x. Este razonamiento se puede expresar como una secuencia de pasos. Cada paso es, o bien la hipótesis del teorema que se desea demostrar, o bien parte de dicha hipótesis, o bien una proposición que se deriva de una o varias de las proposiciones anteriores.

Cuando decimos "se deriva" nos referimos a que, si la hipótesis de un teorema es una proposición previa, entonces la conclusión de dicho teorema es verdadera, y puede escribirse como una proposición que forma parte de la demostración. Esta regla lógica se denomina frecuentemente *modus ponens*; esto es, si sabemos que H es verdadera, y también sabemos que la afirmación "si H, entonces C"

es verdadera, podemos concluir que C es verdadera. También se permiten otros pasos lógicos para la obtención de proposiciones que se puedan derivar de una o varias proposiciones previamente disponibles. Por ejemplo, si A y B son dos proposiciones disponibles, se puede deducir y escribir la proposición "A y B".

La Figura 1.3 muestra la secuencia de proposiciones necesaria para demostrar el Teorema 1.4. Aunque por regla general no demostraremos los teoremas de manera tan elegante, esta forma de presentar la demostración ayuda a pensar en las demostraciones como listas de proposiciones muy explícitas, cada una de las cuales se justifica de forma precisa. En el paso (1), hemos repetido una de las premisas del teorema: que x es la suma de los cuadrados de cuatro enteros. Para el desarrollo de las demostraciones, a menudo es útil asignar nombre a las cantidades a las que se hace referencia, aunque no hayan sido identificadas inicialmente, y eso es lo que hemos hecho en este caso, asignando a los cuatro enteros los identificadores a, b, c, y d.

	Afirmación	Justificación
1.	$x = a^2 + b^2 + c^2 + d^2$	Postulado
2.	$a \geq 1$; $b \geq 1$; $c \geq 1$; $d \geq 1$	Postulado
3.	$a^2 \geq 1$; $b^2 \geq 1$; $c^2 \geq 1$; $d^2 \geq 1$	(2) y las propiedades aritméticas
4.	$x \geq 4$	(1), (3) y las propiedades aritméticas
5.	$2^x \geq x^2$	(4) y el Teorema 1.3

Figura 1.3: Demostración formal del Teorema 1.4

En el paso (2), anotamos la otra parte de la hipótesis del teorema: que los números base de los cuadrados valen al menos 1. Técnicamente, esta afirmación representa cuatro proposiciones diferentes, una por cada uno de los cuatro enteros a los que se hace referencia. Después, en el paso (3), observamos que si un número es mayor o igual que 1, su cuadrado también es mayor o igual que 1. La justificación que utilizamos para ello es el hecho de que la afirmación (2) es cierta, así como las "propiedades aritméticas". Es decir, suponemos que el lector sabe, o es capaz de probar, las afirmaciones sencillas relativas al funcionamiento de las desigualdades, tales como "si $y \geq 1$, entonces $y^2 \geq 1$."

El paso (4) utiliza las proposiciones (1) y (3). La primera establece que x es la suma de los cuatro cuadrados en cuestión, mientras que (3) postula que cada uno de dichos cuadrados es mayor o igual que 1. De nuevo, utilizando propiedades aritméticas por todos conocidas, concluimos que x vale al menos $1 + 1 + 1 + 1$, esto es, 4.

En el último paso de la demostración, el paso (5), utilizamos la proposición (4), que es la hipótesis del Teorema 1.3. Este teorema es la justificación que se utiliza para obtener la conclusión, dado que su hipótesis es una de las proposiciones previas de la demostración. Puesto que la proposición (5), que es la conclusión del Teorema 1.3, también es la conclusión del Teorema 1.4, hemos conseguido probar el Teorema 1.4.

Es decir, hemos comenzado con la hipótesis de dicho teorema, y hemos logrado deducir su conclusión. □

1.2.2 Reducción a definiciones

En los dos teoremas anteriores, ambas hipótesis utilizaban términos que deberían resultar familiares: entero, suma y multiplicación, por ejemplo. En muchos otros teoremas, incluyendo gran parte de los del área de la teoría de autómatas, los términos de los que se hace uso en las diversas proposiciones pueden tener implicaciones menos obvias. Una formal útil de proceder en muchas demostraciones es:

- Si no se está seguro de cómo comenzar una demostración, se transformarán todos los términos que intervienen en la hipótesis en sus respectivas definiciones.

He aquí un ejemplo de un teorema sencillo de demostrar, una vez que las proposiciones que lo componen se formulan en función de sus términos elementales. Se utilizan las dos definiciones siguientes:

1. Se dice que un conjunto S es ***finito*** si existe un entero n tal que S posee exactamente n elementos. Escribimos $\|S\| = n$, donde $\|S\|$ se utiliza para representar el número de elementos de un conjunto S. Si el conjunto S no es finito, decimos que S es ***infinito***. Intuitivamente, un conjunto infinito es un conjunto que contiene un número de elementos superior a cualquier número entero.

2. Si S y T son subconjuntos de algún conjunto U, entonces T es el ***conjunto complementario*** de S (con respecto a U) si $S \cup T = U$ y $S \cap T = \emptyset$. Esto es, cada elemento de U es, o bien un elemento de S, o bien un elemento de T; en otras palabras, T está formado exactamente por aquellos elementos de U que no pertenecen a S.

Teorema 1.5: Sea S un subconjunto finito de un conjunto infinito U. Sea T el conjunto complementario de S con respecto a U. Entonces, T es infinito.

PRUEBA: Intuitivamente, este teorema dice que, si se dispone de una cantidad infinita de algo (U), de la cuál se toma una cantidad finita (S), sigue quedando una cantidad infinita. Comencemos expresando los hechos que se mencionan en el teorema tal y como aparecen en la Figura 1.4.

Todavía no hemos avanzado significativamente, de modo que haremos uso de una técnica habitual que se utiliza para la realización de demostraciones, denominada "demostración por reducción al absurdo". Para aplicar este método de demostración, que se tratará en la Sección 1.3.3, suponemos que la conclusión es falsa. Utilizamos esta suposición, junto con parte de la hipótesis, para demostrar lo contrario de alguno de los postulados de la hipótesis. En ese caso, habremos demostrado que es imposible que todas las componentes de la

Proposición original	Nueva proposición
S es finito	Existe un entero n tal que $\|S\| = n$
U es infinito	No existe un entero p tal que $\|U\| = p$
T es el conjunto complementario de S	$S \cup T = U$, y $S \cap T = \emptyset$

Figura 1.4: Nuevo enunciado de los postulados del Teorema 1.5

hipótesis sean verdaderas y que la conclusión sea, a la vez, falsa. Entonces solo queda la posibilidad de que la conclusión sea verdadera siempre que la hipótesis lo sea. Esto es, el teorema es verdadero.

En el caso del Teorema 1.5, lo contrario de la conclusión es que "T es finito". Supongamos que T es finito, junto con la proposición que forma parte de la hipótesis y que dice que S es finito: esto es, que $\|S\| = n$ para algún entero n. Igualmente, la suposición de que T es finito puede reformularse como $\|T\| = m$ para algún entero m.

Ahora, uno de los postulados nos dice que $S \cup T = U$, y $S \cap T = \emptyset$. Es decir, que los elementos que hay en U son exactamente los mismos elementos que hay en S y en T. Por lo tanto, en U debe haber $n + m$ elementos. Dado que $n + m$ es entero, y hemos demostrado que $\|U\| = n + m$, se sigue que U es finito. Más exactamente, concluimos que el número de elementos del conjunto U es un número entero, lo cuál resulta ser la definición de "finito". Pero la afirmación de que U es finito contradice el postulado de que U sea infinito. Por lo tanto, hemos utilizado la negación de nuestra conclusión para demostrar que uno de los postulados de la hipótesis es contradictorio, y, por el principio de "reducción al absurdo", podemos concluir que el teorema es verdadero. □

Las demostraciones no tienen que ser tan prolijas. Una vez revisadas las ideas que subyacen en la demostración, vamos a demostrar de nuevo el teorema en unas pocas líneas.

PRUEBA: (del Teorema 1.5) Sabemos que $S \cup T = U$, y que S y T son disjuntos; por lo tanto, $\|S\| + \|T\| = \|U\|$. Dado que S es finito, $\|S\| = n$ para algún entero n, y dado que U es infinito, no existe ningún entero p tal que $\|U\| = p$. Suponemos que T es finito; es decir, que $\|T\| = m$ para algún entero m. Entonces, $\|U\| = \|S\| + \|T\| = n + m$, lo cual contradice el postulado de que no existía ningún entero p que fuera igual a $\|U\|$. □

1.2.3 Otras formas de teoremas

Los teoremas de la forma "SI ENTONCES" son los más habituales en áreas muy clásicas de las matemáticas. Sin embargo, existen afirmaciones de otra clase que también resultan ser teoremas. En esta sección examinaremos los

Postulados que incluyen cuantificadores

Muchos teoremas incluyen afirmaciones que utilizan los *cuantificadores* "para todo" y "existe", o variaciones equivalentes. El orden en que aparecen los cuantificadores afecta al significado de la afirmación. Suele ser útil contemplar este tipo de postulados como un "juego" entre dos jugadores (para todo y existe), que especifican por turno valores para los parámetros del teorema. "Para todo" debe tener en cuenta todas las opciones, de modo que sus elecciones suelen considerarse variables. "Existe" solo tiene que elegir un valor, que depende de los que se hayan elegido antes. Si el último jugador siempre puede elegir un valor aceptable, el postulado es verdadero.

Por ejemplo, considérese una definición alternativa de "conjunto infinito": "El conjunto S es *infinito* si y solo si para todo entero n, existe un subconjunto T de S que contiene exactamente n elementos". Aquí, "para todo" precede a "existe", así que debemos empezar tomando un entero arbitrario n. Ahora, "existe" elige un subconjunto T, pudiendo utilizar n para hacerlo. Por ejemplo, si S fuera el conjunto de los números enteros, "existe" podría elegir el subconjunto $T = \{1, 2, \ldots, n\}$, y por lo tanto acertar para cualquier valor de n. Esto demuestra que el conjunto de los enteros es infinito.

La proposición siguiente es similar a la anterior, pero es *incorrecta*, porque los cuantificadores aparecen en el orden contrario: "Existe un subconjunto T del conjunto S tal que para todo n, el conjunto T tiene exactamente n elementos". Aquí, dado un conjunto S, como el de los enteros, "existe" puede elegir cualquier conjunto T; digamos que elige $\{1, 2, 5\}$. Ahora, "para todo" debe demostrar que T posee n elementos para *cada* posible n. Pero "para todo" no puede hacerlo. Por ejemplo, es falso para $n = 4$ y, de hecho, para cualquier $n \neq 3$.

tipos más habituales de postulados y lo que normalmente es necesario realizar para poder demostrarlos.

Varias maneras de decir "SI ENTONCES"

En primer lugar, existen algunas formas de enunciar teoremas que a primera vista no parecen "si H, entonces C", pero que de hecho expresan lo mismo: si la hipótesis H es verdadera para cierto valor del parámetro o de los parámetros, entonces la conclusión C es verdadera para esos mismos valores. He aquí algunas otras formas en las que puede expresarse "si H, entonces C".

1. H implica C.
2. H solo si C.

3. C si H.

4. Siempre que se verifique H, se sigue C.

También pueden encontrarse otras muchas otras variantes de la formulación (4), como "si H se verifica, entonces C también", o "siempre que se verifique H, se verifica C".

Ejemplo 1.6 : En las cuatro formas anteriores, la proposición del Teorema 1.3 se enunciaría:

1. $x \geq 4$ implica $2^x \geq x^2$.
2. $x \geq 4$ solo si $2^x \geq x^2$.
3. $2^x \geq x^2$ si $x \geq 4$.
4. Siempre que se verifique $x \geq 4$, se sigue $2^x \geq x^2$.

□

Además, en el entorno de la lógica formal, a menudo se utiliza el operador $\rightarrow$ en lugar de "SI ENTONCES". Es decir, en algunos textos matemáticos la proposición "si H, entonces C" se expresaría $H \rightarrow C$. Nosotros no utilizaremos esta notación en nuestras exposiciones.

Proposiciones de la forma "SI Y SOLO SI"

A veces, encontramos proposiciones de la forma "A si y solo si B". Este tipo de proposición también puede aparecer como "A sii B" [1], "A es equivalente a B" o "A exactamente cuando B". En realidad, esta proposición está formada por dos proposiciones de la forma "SI ENTONCES": "Si A, entonces B", y "si B, entonces A". La proposición "A si y solo si B" se demuestra comprobando las dos proposiciones que la forman:

1. La *parte si*: "Si B, entonces A", y
2. La *parte solo si*: "Si A, entonces B", que a menudo se formula de manera equivalente como "A solo si B".

Ambas demostraciones se pueden realizar en cualquier orden. En muchos teoremas, una de las dos demostraciones es más fácil que la otra, siendo habitual presentar primero la demostración más sencilla de realizar para quitársela de en medio.

En lógica formal, se pueden utilizar los operadores $\leftrightarrow$ o $\equiv$ para expresar proposiciones de la forma "SI Y SOLO SI". Es decir, $A \equiv B$ y $A \leftrightarrow B$ significan lo mismo que "A si y solo si B".

[1] "Sii" no es realmente una palabra, sino una abreviación de "SI Y SOLO SI" que se utiliza en algunos tratados matemáticos por razones de brevedad.

¿Cómo tienen que ser las demostraciones formales?

No es fácil responder a esta pregunta. El objetivo final de una demostración es convencer a alguien, sea un alumno de la asignatura o nosotros mismos, de que la estrategia que se sigue para desarrollar cierto código es correcta. Si la demostración convence, es suficiente; si no convence al "destinatario", la demostración deja mucho que desear.

Parte de la incertidumbre respecto a las demostraciones proviene del hecho de que cada uno de sus posibles destinatarios puede tener conocimientos diferentes. En este sentido, en el Teorema 1.4 supusimos que se conocían todos los principios aritméticos y que sería creíble una proposición de la forma "si $y \geq 1$ entonces $y^2 \geq 1$". Si no se estuviera familiarizado con los principios aritméticos, habría sido necesario probar dicha proposición mediante la inclusión de algunos pasos adicionales en la demostración deductiva que se ha realizado.

Sin embargo, hay ciertas cosas que es necesario incluir en las demostraciones, cuya omisión en una demostración la hace claramente inadecuada. Por ejemplo, puede ser inadecuada una demostración deductiva que utilice proposiciones que no se justifiquen a partir de los postulados o de las proposiciones previas. Cuando se desea demostrar una proposición del tipo "SI Y SOLO SI", hace falta demostrar tanto la parte "SI" como la parte "SOLO SI".

Como ejemplo adicional, las demostraciones inductivas (que se examinan en la sección 1.4) requieren demostraciones de las bases y de los pasos de inducción.

Cuando se demuestra una proposición del tipo "SI Y SOLO SI", es importante recordar que deben probarse tanto la parte "SI" como la parte "SOLO SI". A veces, puede resultar útil descomponer una proposición "SI Y SOLO SI" en una sucesión de varias equivalencias. Esto es, para probar "A si y solo si B", primero podría demostrarse "A si y solo si C", y luego demostrarse "C si y solo si B". Ese procedimiento funciona, siempre y cuando no olvidemos que cada paso "SI Y SOLO SI" debe ser demostrado en ambos sentidos. Si existe un único paso que se haya probado solamente en un sentido, la demostración al completo ya no es válida.

Sigue un ejemplo de una demostración sencilla del tipo "SI Y SOLO SI", que utiliza la notación:

1. $\lfloor x \rfloor$, el *suelo* de un número real x, es el mayor entero igual o menor que x.

2. $\lceil x \rceil$, el *techo* de un número real x, es el menor entero igual o mayor que x.

Teorema 1.7: Sea x un número real. Entonces $\lfloor x \rfloor = \lceil x \rceil$ si y solo si x es entero.

PRUEBA: (SOLO SI) En esta parte, suponemos $\lfloor x \rfloor = \lceil x \rceil$ e intentamos demostrar que x es entero. Utilizando las definiciones de suelo y techo, se puede observar que $\lfloor x \rfloor \leq x$, y que $\lceil x \rceil \geq x$. Sin embargo, se parte de que $\lfloor x \rfloor = \lceil x \rceil$. Por lo tanto, podemos sustituir el suelo por el techo en la primera desigualdad, llegando a la conclusión de que $\lceil x \rceil \leq x$. Dado que tanto $\lceil x \rceil \leq x$ como $\lceil x \rceil \geq x$ son ciertas, concluimos, mediante las propiedades aritméticas de las desigualdades, que $\lceil x \rceil = x$. Dado que $\lceil x \rceil$ es siempre entero, entonces x también tiene que ser entero en este caso.

(SI) Ahora suponemos que x es entero e intentamos probar que $\lfloor x \rfloor = \lceil x \rceil$. Esta parte es fácil. Por definición de suelo y techo, cuando x es entero, tanto $\lfloor x \rfloor$ como $\lceil x \rceil$ son iguales a x, y, por lo tanto, iguales entre sí. □

1.2.4 Teoremas que no parecen ser proposiciones de la forma "SI ENTONCES"

A veces se encuentran teoremas que no parecen tener hipótesis. Un ejemplo muy conocido en trigonometría es:

Teorema 1.8: $\text{sen}^2\theta + \cos^2\theta = 1$. □

En realidad, esta proposición *sí* posee una hipótesis, que está formada por todas las proposiciones necesarias para poder interpretar dicha proposición. En concreto, la hipótesis oculta es que θ es un ángulo, y, por lo tanto, las funciones seno y coseno tienen el significado habitual cuando se refieren a ángulos. Este teorema se puede demostrar a partir de la definición de dichos términos y del teorema de Pitágoras ("en un triángulo rectángulo, el cuadrado de la hipotenusa es igual a la suma de los cuadrados de los otros dos lados"). Esencialmente, la forma "SI ENTONCES" del teorema es en realidad: "Si θ es un ángulo, entonces $\text{sen}^2\theta + \cos^2\theta = 1$".

1.3 Otras formas de demostración

En esta sección nos ocupamos de otros temas adicionales que tienen que ver con el modo de construir demostraciones:

1. Demostraciones sobre conjuntos.
2. Demostraciones que utilizan la conversión contradictoria.
3. Demostraciones por reducción al absurdo.
4. Demostraciones mediante contraejemplos.

1.3.1 Demostración de equivalencias entre conjuntos

En el área de la teoría de autómatas, a menudo es necesario demostrar teoremas que dicen que ciertos conjuntos construidos de diferentes formas son en realidad el mismo conjunto. Frecuentemente, se trata de conjuntos de cadenas de caracteres, que reciben el nombre de "lenguajes", pero en esta sección la naturaleza de dichos conjuntos no es relevante. Si E y F son dos expresiones que representan sendos conjuntos, entonces la proposición $E = F$ significa que ambos conjuntos son el mismo. Más exactamente, todos y cada uno de los elementos del conjunto representado por E están en F, y todos y cada uno de los elementos del conjunto representado por F están en E.

Ejemplo 1.9 : La ***propiedad conmutativa de la unión*** dice que se puede realizar la unión de dos conjuntos R y S en cualquier orden. Esto es, $R \cup S = S \cup R$. En este caso, E es la expresión $R \cup S$, y F es la expresión $S \cup R$. La propiedad conmutativa de la unión dice que $E = F$. □

Se puede escribir la igualdad entre conjuntos $E = F$ como una proposición de la forma "SI Y SOLO SI": "Un elemento x está en el conjunto E si y solo si x está en el conjunto F". En consecuencia, veamos el esquema de la demostración de cualquier proposición que afirme la igualdad entre dos conjuntos $E = F$; esta demostración tiene la misma forma que cualquier demostración del tipo "SI Y SOLO SI":

1. Probar que si x está en el conjunto E, entonces x está en el conjunto F.

2. Probar que si x está en el conjunto F, entonces x está en el conjunto E.

Como ejemplo de este procedimiento de demostración, vamos a probar la ***propiedad distributiva de la unión respecto de la intersección***:

Teorema 1.10: $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

PRUEBA: Los dos conjuntos involucrados se representan mediante las expresiones $E = R \cup (S \cap T)$ y $F = (R \cup S) \cap (R \cup T)$. Demostramos independientemente las dos partes del teorema. En la parte "SI" suponemos que el elemento x está en E y demostramos que está en F. Esta parte, resumida en la Figura 1.5, utiliza las definiciones de la unión y de la intersección, con las que se supone que el lector está familiarizado.

Ahora, debemos demostrar la parte "SOLO SI" del teorema. Aquí, suponemos que x está en F y demostramos que está en E. Los pasos de la demostración se resumen en la Figura 1.6. Dado que ya se han demostrado ambas partes de la proposición "SI Y SOLO SI", la propiedad distributiva de la unión respecto de la intersección queda demostrada. □

	Proposición	Justificación
1.	x está en $R \cup (S \cap T)$	Postulado
2.	x está en R o x está en $S \cap T$	(1) y la definición de la unión
3.	x está en R o x está tanto en S como en T	(2) y la definición de la intersección
4.	x está en $R \cup S$	(3) y la definición de la unión
5.	x está en $R \cup T$	(3) y la definición de la unión
6.	x está en $(R \cup S) \cap (R \cup T)$	(4), (5) y la definición de la intersección

Figura 1.5: Los pasos de la parte "SI" del Teorema 1.10

	Proposición	Justificación
1.	x está en $(R \cup S) \cap (R \cup T)$	Postulado
2.	x está en $R \cup S$	(1) y la definición de la intersección
3.	x está en $R \cup T$	(1) y la definición de la intersección
4.	x está en R o x está tanto en S como en T	(2), (3), y el razonamiento sobre la unión
5.	x está en R o x está en $S \cap T$	(4) y la definición de la intersección
6.	x está en $R \cup (S \cap T)$	(5) y la definición de la unión

Figura 1.6: Los pasos de la parte "SOLO SI" del Teorema 1.10

1.3.2 La conversión contradictoria

Toda proposición de la forma "SI ENTONCES" tiene un enunciado equivalente cuya demostración, en algunas circunstancias, puede resultar más sencilla. La *conversión contradictoria* de la proposición "si H, entonces C" es "si no C, entonces no H". Una proposición y su conversión contradictoria son, o bien ambas verdaderas, o bien ambas falsas, así que se puede optar por demostrar cualquiera de las dos para demostrar también la otra.

Para ver por qué "si H, entonces C", y "si no C, entonces no H", son lógicamente equivalentes, observemos primero que son cuatro los casos a considerar:

1. H y C son ambas verdaderas.
2. H es verdadera y C es falsa.
3. C es verdadera y H es falsa.
4. H y C son ambas falsas.

Solo hay una forma de que una proposición "SI ENTONCES" sea falsa; la hipótesis debe ser verdadera y la conclusión falsa, como en el caso (2). En los

Enunciados "SI Y SOLO SI" para conjuntos

Como se ha mencionado, los teoremas que afirman equivalencias entre expresiones que denotan conjuntos constituyen proposiciones "SI Y SOLO SI". Por tanto, el Teorema 1.10 podría haberse enunciado así: "Un elemento x pertenece a $R \cup (S \cap T)$ si y solo si x pertenece a

$$(R \cup S) \cap (R \cup T)"$$

Otra expresión habitual acerca de la equivalencia de conjuntos utiliza la locución "todos y solo ellos". Por ejemplo, el Teorema 1.10 también se podría haber enunciado así "Los elementos de $R \cup (S \cap T)$ son, todos y solo ellos, elementos de

$$(R \cup S) \cap (R \cup T)"$$

otros tres casos, incluyendo el caso (4) en el que la conclusión es falsa, la propia proposición "SI ENTONCES" es verdadera.

Ahora, considérense los casos para los que la conversión contradictoria "si no C entonces no H" es falsa. Para que esta proposición sea falsa, su hipótesis (que es "no C") ha de ser verdadera, y su conclusión (que es "no H") ha de ser falsa. Pero "no C" es verdadera exactamente cuando C es falsa, y "no H" es falsa exactamente cuando H es verdadera. Estas dos condiciones corresponden de nuevo al caso (2), lo cuál demuestra que en cada uno de los cuatro casos, la proposición original y su conversión contradictoria son o bien ambas verdaderas o bien ambas falsas; esto es, son lógicamente equivalentes.

Ejemplo 1.11 : Recordemos el Teorema 1.3, cuya proposición era: "Si $x \geq 4$, entonces $2^x \geq x^2$". La conversión contradictoria de esta proposición es "si no $2^x \geq x^2$, entonces no $x \geq 4$". En términos más coloquiales, utilizando el hecho de que "no $a \geq b$" es lo mismo que $a < b$, la conversión contradictoria es "si $2^x < x^2$, entonces $x < 4$". □

Cuando se pide demostrar un teorema de la forma "SI Y SOLO SI", el uso de la conversión contradictoria en una de las partes permite varias opciones. Por ejemplo, supongamos que se desea demostrar la equivalencia entre conjuntos $E = F$. En vez de demostrar "si x está en el conjunto E, entonces x está en el conjunto F, y si x está en el conjunto F, entonces x está en el conjunto E", se podría también enunciar uno de los sentidos en su forma contradictoria. Una forma equivalente para la demostración es:

- "Si x está en el conjunto E, entonces x está en el conjunto F, y si x no está en el conjunto E, entonces x no está en el conjunto F".

La inversa

No deben confundirse los términos "contradictorio" e "inverso". La *inversa* de una proposición "SI ENTONCES" es el "otro sentido"; esto es, la inversa de "si H, entonces C" es "si C, entonces H". A diferencia de la conversión contradictoria, que es equivalente a la original desde el punto de vista lógico, la proposición inversa *no* es equivalente a la proposición original. De hecho, las dos partes de una demostración "SI Y SOLO SI" siempre están constituidas por una proposición y su inversa.

En la proposición anterior, E y F podrían intercambiarse.

1.3.3 Demostración por reducción al absurdo

Otra manera de demostrar una proposición de la forma "si H, entonces C" consiste en probar la proposición

- "H y no C implica falsedad".

Es decir, se comienza suponiendo que son ciertas tanto la hipótesis H como la negación de la conclusión C. La demostración se completa probando que algo que se sabe que es falso deriva lógicamente de "H y no C". Este modo de demostración se conoce con el nombre de *demostración por reducción al absurdo*.

Ejemplo 1.12: Recordemos el Teorema 1.5, donde se probó la proposición "SI ENTONCES" cuya hipótesis era $H =$ "U es un conjunto infinito, S es un subconjunto finito de U, y T es el conjunto complementario de S respecto a U". Su conclusión C era "T es infinito". La demostración de este teorema se realizó por reducción al absurdo. Supusimos "no C", esto es, que T era infinito.

La demostración nos condujo a obtener una falsedad a partir de H y no C. Primero probamos que U tenía que ser finito a partir de las suposiciones de que tanto S como T eran finitos. Pero, dado que en la hipótesis H se establece que U es infinito, y un conjunto no puede ser simultáneamente finito e infinito, se demostró que la proposición lógica es "falsa". En términos lógicos, disponemos tanto de una proposición p (U es finito) como de su negación, no p (U es infinito). Utilizamos entonces el hecho de que "p y no p" es equivalente, desde el punto de vista lógico, a "falso". □

Para ver por qué las demostraciones por reducción al absurdo son lógicamente correctas, recordemos de la Sección 1.3.2 que existen cuatro combinaciones de valores de verdad para H y C. Únicamente el segundo caso, H verdadera y C falsa, hace que la proposición "si H, entonces C" sea falsa. Comprobando que H y no C conduce a una falsedad, estamos demostrando que el caso 2 no puede ocurrir. Por lo tanto, las únicas combinaciones posibles de valores de

verdad para H y C son las tres combinaciones que hacen que "si H, entonces C" sea verdadera.

1.3.4 Contraejemplos

En la vida real no se nos pide que demostremos un teorema. Más bien, nos enfrentamos con algo que parece ser cierto (por ejemplo, una estrategia para implementar un programa), y tenemos que decidir si el "teorema" es o no verdadero. Para resolver esta cuestión, podemos intentar demostrar el teorema, y, si no es posible, intentar la otra alternativa: demostrar que es falso.

Generalmente, los teoremas son proposiciones que incluyen un número infinito de casos, quizá todos los valores de sus parámetros. En realidad, el convenio matemático estricto solo considera que una proposición es digna de recibir el nombre de "teorema" si incluye un número infinito de casos; las proposiciones que no tienen parámetros, o que únicamente se aplican a un número finito de los valores de su(s) parámetro(s) reciben el nombre de *observaciones*. Basta probar que un supuesto teorema es falso para uno solo de los casos, para demostrar que dicha proposición no es un teorema. La situación es análoga a la programación, dado que se considera generalmente que un programa tiene un error si no funciona correctamente para una entrada sobre la que se esperaba que funcionara, aunque solo sea una.

A menudo es más sencillo demostrar que una proposición no es un teorema que demostrar que *sí* lo es. Como se mencionó, si S es cualquier proposición, entonces la proposición "S no es un teorema" es en sí misma una proposición sin parámetros, y, por tanto, puede considerarse una observación más que un teorema. He aquí dos ejemplos, el primero de los cuáles es obviamente un no-teorema, y el segundo una proposición que no está claro que sea un teorema, y que hay que examinar antes de resolver la cuestión de si es o no un teorema.

Presunto Teorema 1.13 : Todos los números primos son impares. (Más formalmente, podríamos decir: "Si un número entero x es primo, entonces x es impar".)

REFUTACION: El número entero 2 es primo, pero 2 es par. □

Ahora, examinemos un "teorema" de aritmética modular. Primero debemos establecer una definición esencial. Si a y b son números enteros positivos, a mod b representa el resto de la división de a entre b, esto es, el único número entero r entre 0 y $b-1$ tal que $a = qb + r$ para algún entero q. Por ejemplo, 8 mod 3 = 2 y 9 mod 3 = 0. El primer teorema que proponemos, el cual determinaremos que es falso, es:

Presunto Teorema 1.14 : No existe un par de números enteros a y b tal que

$$a \bmod b = b \bmod a$$

□

Cuando se pide operar con pares de objetos, como aquí a y b, a menudo es posible simplificar la relación entre ambos utilizando relaciones de simetría. En este caso, nos podemos concentrar en la situación en la que $a < b$, dado que si $b < a$ podemos cambiar a por b y viceversa, y obtener la mima ecuación, como sucede en el Presunto Teorema 1.14. Sin embargo, hay que tener cuidado para que no se nos olvide el tercer caso, en el que $a = b$. Este caso resulta ser fatal para nuestro intento de demostración.

Supongamos que $a < b$. Entonces $a \bmod b = a$, dado que en la definición de $a \bmod b$ tenemos $q = 0$ y $r = a$. Esto es, cuando $a < b$ tenemos $a = 0 \times b + a$. Pero $b \bmod a < a$, dado que cualquier mod a está entre 0 y $a - 1$. Por tanto, cuando $a < b$, $b \bmod a < a \bmod b$, de modo que $a \bmod b = b \bmod a$ es imposible. Utilizando el argumento de simetría expuesto anteriormente, también vemos que $a \bmod b \neq b \bmod a$ cuando $b < a$.

Sin embargo, considerereremos el tercer caso: $a = b$. Dado que $x \bmod x = 0$ para cualquier entero x, *tenemos* que $a \bmod b = b \bmod a$ si $a = b$. Disponemos así de una refutación del presunto teorema:

REFUTACION: (del Presunto Teorema 1.14) Sea $a = b = 2$. Entonces

$$a \bmod b = b \bmod a = 0$$

□

En el proceso de buscar un contraejemplo, lo que de hecho hemos descubierto son las condiciones exactas bajo las cuáles el presunto teorema es cierto. He aquí la versión correcta del teorema y su demostración.

Teorema 1.15 : $a \bmod b = b \bmod a$ si y solo si $a = b$.

PRUEBA: (SI) Supongamos que $a = b$. Entonces, tal y como observamos anteriormente, $x \bmod x = 0$ para cualquier número entero x. Por tánto, $a \bmod b = b \bmod a = 0$ siempre y cuando $a = b$.

(SOLO SI) Ahora, supongamos que $a \bmod b = b \bmod a$. La mejor técnica de demostración es por reducción al absurdo, así que supongamos también la negación de la conclusión, esto es, supongamos que $a \neq b$. Dado que $a = b$ está eliminado, solo tenemos que considerar los casos $a < b$ y $b < a$.

Ya observamos anteriormente que cuando $a < b$, tenemos que $a \bmod b = a$ y que $b \bmod a < a$. De modo que estas proposiciones, junto con la hipótesis $a \bmod b = b \bmod a$, nos permiten llegar a una contradicción.

Por simetría, si $b < a$, entonces $b \bmod a = b$ y $a \bmod b < b$. Llegamos de nuevo a una contradicción de la hipótesis, y concluimos que la parte "SOLO SI" también es verdadera. Ahora hemos demostrado ambos sentidos, y concluimos que el teorema es verdadero. □

1.4 Demostraciones inductivas

Existe un modo especial de demostración que recibe el nombre de "inductiva", que es esencial cuando se trata de objetos definidos recursivamente. Muchas de las demostraciones inductivas más conocidas tienen como objeto números enteros, pero también es necesario realizar demostraciones inductivas en la teoría de autómatas cuando se trata de objetos definidos de forma recursiva, como árboles y expresiones de varios tipos; por ejemplo, las expresiones regulares que se mencionaron brevemente en la Sección 1.1.2. En esta sección, introducimos el tema de las demostraciones inductivas mediante inducciones "sencillas" sobre números enteros. Luego, mostramos cómo realizar inducciones "estructurales" sobre cualquier concepto que esté definido de forma recursiva.

1.4.1 Inducciones sobre números enteros

Supongamos que hemos de probar una proposición $S(n)$ acerca de un número entero n. Un procedimiento habitual consiste en probar dos cosas:

1. La *base*, en la que demostramos $S(i)$ para cierto entero i. Normalmente, $i = 0$ o $i = 1$, pero hay ejemplos en los que se desea comenzar por un valor de i más elevado, tal vez porque la proposición S sea falsa para los valores pequeños de i.

2. El *paso de inducción*, en el que suponemos que $n \geq i$, siendo i el entero base, y demostramos que "si $S(n)$, entonces $S(n+1)$".

Intuitivamente, estas dos partes deberían convencernos de que $S(n)$ es cierta para cualquier número entero n que sea igual o mayor que el entero base i. El argumento puede ser el siguiente. Supongamos que $S(n)$ fuera falso para al menos uno de dichos números enteros. Entonces, tendría que existir un valor mínimo de n, digamos j, para el cual $S(j)$ fuera falso, aun siendo $j \geq i$. Sin embargo, j no podría ser i, ya que en la parte base se probó que $S(i)$ es verdadera. Por tanto, j debe ser mayor que i. Ahora sabemos que $j - 1 \geq i$, y que $S(j-1)$ es verdadera.

Sin embargo, en la parte inductiva se demostró que si $n \geq i$, entonces $S(n)$ implica $S(n+1)$. Supongamos que $n = j - 1$. Entonces sabemos, a partir del paso de inducción, que $S(j-1)$ implica $S(j)$. Dado que $S(j-1)$ es verdadero, se puede concluir que $S(j)$ también lo es.

Hemos supuesto la negación de lo que queremos probar; esto es, supusimos que $S(j)$ era falso para algún $j \geq i$. En cualquier caso, se obtiene una contradicción, así que tenemos una "demostración por reducción al absurdo" de que $S(n)$ es cierto para todo $n \geq i$.

Desgraciadamente, el razonamiento anterior tiene un defecto sutil. La suposición de que podemos elegir un $j \geq i$ mínimo para el que $S(j)$ sea falsa depende del principio de inducción. Esto es, la única manera de probar que podemos encontrar tal j es mediante un método que resulta ser en esencia una demostración inductiva. Sin embargo, la "demostración" realizada anteriormente tiene

sentido desde el punto de vista intuitivo, y encaja con nuestra comprensión del mundo real. Por ello, generalmente se considera parte integrante de nuestro sistema de razonamiento lógico

- ***El principio de inducción***: Si se prueba $S(i)$ y se prueba que para todo $n \geq i$, $S(n)$ implica $S(n+1)$, entonces puede concluirse $S(n)$ para todo $n \geq i$.

Los dos ejemplos siguientes ilustran el uso del principio de inducción para demostrar teoremas sobre los números enteros.

Teorema 1.16 : Para todo $n \geq 0$:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.1}$$

PRUEBA: La demostración tiene dos partes: la base y el paso de inducción; veamos cada una de ellas.

BASE: Para la base, elegimos $n = 0$. Podría parecer sorprendente que el teorema tenga sentido para $n = 0$, ya que cuando $n = 0$, la parte izquierda de la Ecuación (1.1) queda $\sum_{i=1}^{0}$.

Sin embargo, existe un principio general que dice que, cuando el límite superior de una suma (0 en este caso) es menor que el límite inferior (aquí 1), la suma no se realiza sobre ningún término, y, por tanto, la suma es 0. Esto es, $\sum_{i=1}^{0} i^2 = 0$.

La parte derecha de la Ecuación (1.1) es también 0, dado que $0 \times (0+1) \times (2 \times 0 + 1)/6 = 0$. De este modo, la Ecuación (1.1) es verdadera cuando $n = 0$.

PASO INDUCTIVO: Ahora, supongamos que $n \geq 0$. Tenemos que probar el paso de inducción, esto es, que la Ecuación (1.1) implica la misma fórmula cuando n se sustituye por $n+1$. Esta fórmula es

$$\sum_{i=1}^{[n+1]} i^2 = \frac{[n+1]([n+1]+1)(2[n+1]+1)}{6} \tag{1.2}$$

Las Ecuaciones (1.1) y (1.2) pueden simplificarse si se expanden las sumas y los productos de sus respectivas partes derechas.

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n)/6 \tag{1.3}$$

$$\sum_{i=1}^{n+1} i^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.4}$$

Para demostrar (1.4) es necesario utilizar (1.3), ya que en el principio de inducción estas proposiciones corresponden a $S(n+1)$ y $S(n)$, respectivamente.

El "truco" consiste en descomponer la suma hasta $n+1$ de la parte derecha de (1.4) en una suma hasta n más el término $(n+1)$-ésimo. Después, se puede sustituir la suma hasta n por la parte derecha de (1.3) y probar que (1.4) es verdadera. Veamos cómo:

$$\left(\sum_{i=1}^{n} i^2\right) + (n+1)^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.5}$$

$$(2n^3 + 3n^2 + n)/6 + (n^2 + 2n + 1) = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.6}$$

La comprobación final de que (1.6) es verdadera requiere únicamente utilizar álgebra polinómica sencilla sobre la parte izquierda para demostrar que es idéntica a la derecha. □

Ejemplo 1.17: En el siguiente ejemplo, se demuestra el Teorema 1.3 de la Sección 1.2.1. Recordemos que este teorema establece que si $x \geq 4$, entonces $2^x \geq x^2$. Ya dimos entonces una demostración informal basada en la idea de que la proporción $x^2/2^x$ disminuye cuando x crece por encima de 4. Esta idea se puede precisar si probamos la proposición $2^x \geq x^2$ por inducción sobre x, comenzando con la base $x = 4$. Nótese que la proposición es falsa para $x < 4$.

BASE: Si $x = 4$, entonces tanto 2^x como x^2 valen 16. Por tanto, $2^4 \geq 4^2$ es verdadera.

PASO INDUCTIVO: Supongamos que para algún $x \geq 4$ es $2^x \geq x^2$. Tomando esta proposición como hipótesis, necesitamos probar la misma proposición con $x+1$ en lugar de x, esto es, $2^{[x+1]} \geq [x+1]^2$. Éstas resultan ser las proposiciones $S(x)$ y $S(x+1)$ del principio de inducción. El hecho de que se utilice x como parámetro en lugar de n debería ser intranscendente; x o n solo son variables locales.

Como en el Teorema 1.16, $S(x+1)$ debe reescribirse para que haga uso de $S(x)$. En este caso, podemos escribir $2^{[x+1]}$ como 2×2^x. Dado que $S(x)$ nos dice que $2^x \geq x^2$, podemos concluir que $2^{x+1} = 2 \times 2^x \geq 2x^2$.

Pero necesitamos otra cosa. Hay que demostrar $2^{x+1} \geq (x+1)^2$. Una forma de probar esta proposición es demostrar que $2x^2 \geq (x+1)^2$ y utilizar la transitividad de $\geq$ para probar que $2^{x+1} \geq 2x^2 \geq (x+1)^2$. En nuestra demostración de que

$$2x^2 \geq (x+1)^2 \tag{1.7}$$

podemos utilizar la suposición de que $x \geq 4$. Comencemos por simplificar (1.7):

$$x^2 \geq 2x + 1 \tag{1.8}$$

Dividamos (1.8) por x, para obtener:

Los números enteros como conceptos definidos recursivamente

Hemos mencionado que las demostraciones inductivas son útiles cuando la materia de la que se trata está definida recursivamente. Sin embargo, nuestros primeros ejemplos han sido inducciones sobre números enteros, de los que normalmente no se piensa en términos de "definición recursiva". No obstante, existe una definición natural, de carácter recursivo, de los números enteros no negativos. Esta definición encaja con la manera en que se realizan inducciones sobre los números enteros: a partir de objetos definidos en primer lugar, hacia otros definidos con posterioridad.

BASE: 0 es un entero.

PASO INDUCTIVO: Si n es un entero, entonces $n+1$ también lo es.

$$x \geq 2 + \frac{1}{x} \tag{1.9}$$

Dado que $x \geq 4$, sabemos que $1/x \leq 1/4$. De modo que la parte izquierda de (1.9) es al menos 4, y la parte derecha es como mucho 2,25. Entonces, hemos probado que (1.9) es verdadera. Por tanto, las Ecuaciones (1.8) y (1.7) son también verdaderas. Por su parte, la Ecuación (1.7) nos dice que $2x^2 \geq (x+1)^2$ para $x \geq 4$, y demuestra la proposición $S(x+1)$, que recordemos era $2^{x+1} \geq (x+1)^2$. $\square$

1.4.2 Formas más generales de inducción sobre los enteros

A veces una demostración inductiva solo es posible si se utiliza un esquema más general que el propuesto en la Sección 1.4.1, donde probamos una proposición S para un valor base y luego demostramos que "si $S(n)$, entonces $S(n+1)$". Dos generalizaciones importantes de este esquema son:

1. Se pueden utilizar varios casos base. Esto es, se prueba $S(i), S(i+1), \ldots, S(j)$ para algunos valores de $j > i$.

2. Al probar $S(n+1)$, se puede utilizar la validez de todas las proposiciones

$$S(i), S(i+1), \ldots, S(n)$$

en lugar de utilizar únicamente $S(n)$. Además, si se han probado los casos base hasta $S(j)$, se puede suponer que $n \geq j$, en lugar de únicamente $n \geq i$.

La conclusión que se obtendría a partir de estas bases y este paso de inducción es que $S(n)$ es verdadera para todo $n \geq i$.

Ejemplo 1.18: El siguiente ejemplo ilustrará el potencial que tienen ambos principios. La proposición $S(n)$ que queremos demostrar es que si $n \geq 8$, n puede escribirse como una suma de treses y cincos. Por cierto, nótese que 7 no puede escribirse como una suma de treses y cincos.

BASE: Los casos base son $S(8)$, $S(9)$, y $S(10)$. Las comprobaciones son $8 = 3 + 5$, $9 = 3 + 3 + 3$, y $10 = 5 + 5$, respectivamente.

PASO INDUCTIVO: Supongamos que $n \geq 10$ y que $S(8), S(9), \ldots, S(n)$ son verdaderas. Tenemos que demostrar $S(n + 1)$ a partir de estos postulados. Nuestra estrategia es restar 3 a $n + 1$, observar que el número resultante ha de poderse escribir como una suma de treses y cincos, y añadir un 3 más a la suma para obtener una manera de escribir $n + 1$.

Más formalmente, observemos que $n - 2 \geq 8$, así que podemos dar por cierto $S(n - 2)$. Esto es, $n - 2 = 3a + 5b$ para ciertos números enteros a y b. Entonces $n + 1 = 3 + 3a + 5b$, así que $n + 1$ puede escribirse como la suma de $a + 1$ treses y b cincos. Esto prueba $S(n + 1)$, y concluye el paso de inducción. □

1.4.3 Inducciones estructurales

En la teoría de autómatas, existen varias estructuras definidas recursivamente sobre las que es necesario demostrar proposiciones. Los conceptos de árboles y expresiones son ejemplos conocidos e importantes de estas estructuras. Como las inducciones, todas las definiciones recursivas tienen un caso base, en el que se definen una o más estructuras elementales, y un paso de inducción, en el que se definen estructuras más complejas en términos de las estructuras previamente definidas.

Ejemplo 1.19: He aquí la definición recursiva de un árbol:

BASE: Un único nodo es un árbol, y dicho nodo constituye la *raíz* del árbol.

PASO INDUCTIVO: Si $T_1, T_2, \ldots, T_k$ son árboles, se puede construir un árbol nuevo de la siguiente manera:

1. Se comienza con un nuevo nodo N, que es la raíz del árbol.
2. Se añaden copias de todos los árboles $T_1, T_2, \ldots, T_k$.
3. Se añaden arcos desde el nodo N hasta las raíces de cada uno de los árboles $T_1, T_2, \ldots, T_k$.

La Figura 1.7 muestra la construcción inductiva de un árbol con la raíz N a partir de k árboles más pequeños. □

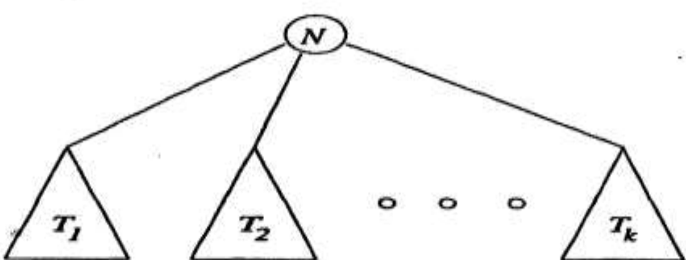


Figura 1.7: Construcción inductiva de un árbol

Ejemplo 1.20 : He aquí otra definición recursiva. Esta vez definimos *expresiones* utilizando los operadores aritméticos + y $*$, y admitiendo que los operandos puedan ser tanto números como variables.

BASE: Cualquier número o letra (esto es, una variable) es una expresión.

PASO INDUCTIVO: Si E y F son expresiones, también lo son $E + F$, $E * F$, y (E).

Por ejemplo, tanto 2 como x son expresiones según la base. El paso de inducción nos dice que $x + 2$, $(x + 2)$, y $2 * (x + 2)$ son expresiones. Nótese cómo cada una de estas expresiones depende de que las anteriores sean expresiones. □

Cuando tenemos una definición recursiva, se pueden probar teoremas acerca de ella utilizando la siguiente forma de demostración, que recibe el nombre de *inducción estructural*. Sea $S(X)$ una proposición acerca de estructuras X definidas mediante alguna definición recursiva determinada.

1. Como base, se prueba $S(X)$ para la(s) estructura(s) base X.
2. Para el paso de inducción, se toma una estructura X, que, según la definición recursiva, está formada a partir de $Y_1, Y_2, \ldots, Y_k$, se dan por ciertas las proposiciones $S(Y_1), S(Y_2), \ldots, S(Y_k)$, y se utilizan para probar $S(X)$.

La conclusión es que $S(X)$ es verdadera para todo X. Los dos teoremas siguientes son ejemplos de hechos que pueden probarse sobre árboles y expresiones.

Teorema 1.21 : El número de nodos de un árbol es superior al de arcos en una unidad.

PRUEBA: La proposición formal $S(T)$ que es necesario probar mediante inducción estructural es: "Si T es un árbol y T tiene n nodos y e arcos, entonces $n = e + 1$".

BASE: El caso base ocurre cuando T es un único nodo. Entonces, $n = 1$ y $e = 0$, así que la relación $n = e + 1$ es verdadera.

Explicación intuitiva de la inducción estructural

Se puede sugerir informalmente por qué la inducción estructural es un método válido de demostración. Imaginemos la definición recursiva que establece, una a una, que ciertas estructuras $X_1, X_2, \ldots$ cumplen la definición. En primer lugar se encuentran los elementos base, junto con el hecho de que X_i forma parte del conjunto de estructuras definido como consecuencia de la pertenencia a dicho conjunto de las estructuras que preceden a X_i en la lista. Vista de esta forma, una inducción estructural no es más que una inducción sobre el entero n de la proposición $S(X_n)$. Esta inducción puede ser de la forma generalizada que se examinó en la Sección 1.4.2, con múltiples casos base y un paso de inducción que utiliza todos los casos previos de la proposición. Sin embargo, se debe recordar, como se explicó en la Sección 1.4.1, que esta explicación intuitiva no constituye una demostración formal, y que, de hecho, tenemos que suponer la validez de este principio de inducción, tal y como se hizo en dicha sección con la validez del principio de inducción original.

PASO INDUCTIVO: Sea T un árbol construido mediante el paso de inducción de la definición a partir del nodo raíz N y de k árboles más pequeños $T_1, T_2, \ldots, T_k$. Podemos suponer que las proposiciones $S(T_i)$ son ciertas para $i = 1, 2, \ldots, k$. Esto es, suponiendo que T_i tiene n_i nodos y e_i arcos, entonces $n_i = e_i + 1$.

El conjunto de todos los nodos de T está constituido por el nodo N y todos los nodos de los T_i. De esta forma, hay $1 + n_1 + n_2 + \cdots + n_k$ nodos en T. Los arcos de T son los k arcos que se añadieron explícitamente en el paso de definición inductiva, más los arcos de los T_i. Por lo tanto, T tiene

$$k + e_1 + e_2 + \cdots + e_k \tag{1.10}$$

arcos. Si sustituimos n_i por $e_i + 1$ en la cuenta del número de nodos de T, encontramos que T tiene

$$1 + [e_1 + 1] + [e_2 + 1] + \cdots + [e_k + 1] \tag{1.11}$$

nodos. La expresión anterior puede reagruparse así:

$$k + 1 + e_1 + e_2 + \cdots + e_k \tag{1.12}$$

Esta expresión es exactamente 1 más que la expresión de (1.10) que daba el número de arcos de T. Por tanto, T tiene un nodo más que arcos. □

Teorema 1.22: Cualquier expresión tiene el mismo número de paréntesis abiertos y cerrados.

PRUEBA: Formalmente, la proposición $S(G)$ se demuestra sobre cualquier expresión G que se defina mediante el proceso recursivo del Ejemplo 1.20: el número de paréntesis abiertos y cerrados de G es el mismo.

BASE: Si G se define a partir de la base, entonces G es o bien un número, o bien una variable. Estas expresiones tienen 0 paréntesis abiertos y 0 paréntesis cerrados, así que ambos números son idéntidos.

PASO INDUCTIVO: Hay tres reglas mediante las cuales puede construirse la expresión G de acuerdo con el paso de inducción de la definición:

1. $G = E + F$.
2. $G = E * F$.
3. $G = (E)$.

Podemos suponer que $S(E)$ y $S(F)$ son verdaderas; esto es, que E tiene el mismo número, digamos n, de paréntesis abiertos y cerrados, e igualmente F tiene el mismo número, digamos m, de paréntesis abiertos y cerrados. Entonces podemos calcular el número de paréntesis abiertos y cerrados de G para cada uno de los tres casos de la siguiente manera:

1. Si $G = E+F$, entonces G tiene $n+m$ paréntesis abiertos y $n+m$ paréntesis cerrados: n de cada tipo provienen de E, y m de cada tipo provienen de F.
2. Si $G = E * F$, el total de paréntesis de G es de nuevo $n + m$ de cada tipo, por la misma razón que en el caso (1).
3. Si $G = (E)$, entonces hay $n+1$ paréntesis abiertos en G (uno de los cuáles aparece explícitamente, y los otros n provienen de E). Igualmente, hay $n + 1$ paréntesis cerrados en G; uno es explícito, y los otros n provienen de E.

En cada uno de los tres casos, vemos que el número de paréntesis abiertos y cerrados de G es el mismo. Esta observación completa el paso inductivo y la demostración. □

1.4.4 Inducciones mutuas

Algunas veces no es posible demostrar una única proposición mediante inducción, siendo necesario demostrar conjuntamente un grupo de proposiciones $S_1(n), S_2(n), \ldots, S_k(n)$ mediante inducción sobre n. En la teoría de autómatas se dan muchas situaciones de ese tipo. En el Ejemplo 1.23 se verá la situación típica en la que es necesario explicar lo que hace un autómata mediante un grupo de proposiciones, una por cada uno de sus estados. Estas proposiciones nos dicen bajo qué secuencias de las entradas alcanza el autómata cada uno de sus estados.

Estrictamente, un grupo de proposiciones es lo mismo que la *conjunción* (Y lógico) de todas las proposiciones. Por ejemplo, el grupo de proposiciones $S_1(n), S_2(n), \ldots, S_k(n)$ podría reemplazarse por la única proposición $S_1(n)$ Y $S_2(n)$ Y $\cdots$ Y $S_k(n)$. Sin embargo, cuando hay que demostrar realmente varias proposiciones independientes, en general suele ser menos confuso mantener separadas las proposiciones y demostrar cada una con su base y sus pasos de inducción. Este tipo de demostración recibe el nombre de *inducción mutua*. Un ejemplo servirá para ilustrar los pasos necesarios para realizar una inducción mutua.

Ejemplo 1.23: Revisemos otra vez el interruptor de encendido/apagado que representamos con un autómata en el Ejemplo 1.1. El autómata correspondiente se muestra en la Figura 1.8. Dado que la acción de pulsar el botón cambia el estado de *on* a *off*, y que el interruptor comienza en el estado *off*, se supone que el siguiente conjunto de proposiciones explica el modo de operación del interruptor:

> $S_1(n)$: El autómata está en el estado *off* después de n pulsaciones si y solo si n es par.
>
> $S_2(n)$: El autómata está en el estado *on* después de n pulsaciones si y solo si n es impar.

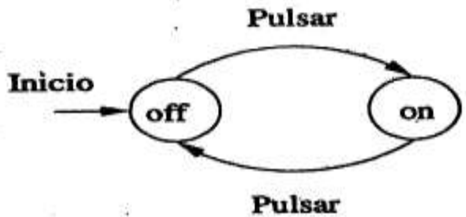


Figura 1.8: Repetición del autómata de la Figura 1.1

Podría suponerse que S_1 implica S_2 y viceversa, dado que se sabe que un número n no puede ser simultáneamente par e impar. Sin embargo, no siempre es cierto que un autómata tenga que estar necesariamente en uno y solo un estado. Sucede que el autómata de la Figura 1.8 siempre se encuentra exactamente en un estado, pero éste es un hecho que hay que demostrar como parte de la inducción mutua.

A continuación se establecen la base y las partes inductivas de las demostraciones de las proposiciones $S_1(n)$ y $S_2(n)$. Las demostraciones dependen de varios hechos que se cumplen para los números enteros pares e impares: si sumamos o restamos 1 a un entero par, obtenemos un entero impar, y si sumamos o restamos 1 a un entero impar, obtenemos un entero par.

BASE: Para la base elegimos $n = 0$. Dado que hay dos proposiciones, y tiene que demostrarse cada una de ellas en ambos sentidos (porque tanto S_1 como S_2 son proposiciones de la forma "SI Y SOLO SI"), realmente existen cuatro casos base, así como cuatro casos de inducción.

1. [S_1; SI] Dado que 0 es par, tenemos que comprobar que, después de 0 pulsaciones, el autómata de la Figura 1.8 está en el estado *off*. Dado que éste es el estado inicial, el autómata está de hecho en el estado *off* después de 0 pulsaciones.

2. [S_1; SOLO SI] El autómata está en el estado *off* después de 0 pulsaciones, así que tenemos que demostrar que 0 es par. Pero 0 es par por la definición de "par", así que no es necesario demostrar nada más.

3. [S_2; SI] La hipótesis correspondiente a la parte "SI" de S_2 es que 0 sea impar. Dado que esta hipótesis H es falsa, cualquier proposición de la forma "si H, entonces C" es verdadera, tal y como se discutió en la Sección 1.3.2. Por tanto, esta parte de la base también es cierta.

4. [S_2; SOLO SI] La hipótesis de que el autómata se encuentra en el estado *on* después de 0 pulsaciones también es falsa, dado que la única forma de acceder al estado *on* es a través de un arco con la etiqueta *Pulsar*, que requiere que el botón sea pulsado al menos una vez. Dado que la hipótesis es falsa, podemos concluir de nuevo que la proposición es verdadera.

PASO INDUCTIVO: Ahora supongamos que $S_1(n)$ y $S_2(n)$ son verdaderas, e intentemos demostrar $S_1(n+1)$ y $S_2(n+1)$. De nuevo, la demostración se divide en cuatro partes.

1. [$S_1(n+1)$; SI] En esta parte, la hipótesis es que $n+1$ es par. Por tanto, n es impar. La parte "SI" de la proposición $S_2(n)$ dice que tras n pulsaciones el autómata se encuentra en el estado *on*. El arco que va desde el estado *on* al estado *off*, con la etiqueta *Pulsar*, indica que la $(n+1)$-ésima pulsación hará que el autómata entre en el estado *off*. Esto completa la demostración de la parte "SI" de $S_1(n+1)$.

2. [$S_1(n+1)$; SOLO SI] La hipótesis es que el autómata se encuentra en el estado *off* después de $n+1$ pulsaciones. El examen del autómata de la Figura 1.8 nos indica que la única manera de acceder al estado *off*, después de uno o más movimientos, es encontrarse en el estado *on* y recibir como entrada *Pulsar*. Así, si nos encontramos en el estado *off* después de $n+1$ pulsaciones, tenemos que haber estado en el estado *on* después de n pulsaciones. Entonces, se puede utilizar la parte "SOLO SI" de la proposición $S_2(n)$ para concluir que n es impar. En consecuencia, $n+1$ es par, que es la conclusión que se deseaba obtener para la parte "SOLO SI" de $S_1(n+1)$.

3. $[S_2(n+1)$; SI] En esencia, esta parte es la misma que la parte (1), intercambiando los papeles de las proposiciones S_1 y S_2, e intercambiando los papeles de "par" e "impar". El lector debería poder construir fácilmente esta parte de la demostración.
4. $[S_2(n+1)$; SOLO SI] En esencia, esta parte es la misma que la parte (2), intercambiando los papeles de las proposiciones S_1 y S_2, e intercambiando los papeles de "par" e "impar".

□

A partir del Ejemplo 1.23 se puede abstraer el patrón para todas las inducciones mutuas:

- Debe demostrarse independientemente tanto la base como el paso de inducción de cada una de las proposiciones.
- Si las proposiciones son del tipo "SI Y SOLO SI", tanto la base como el paso de inducción de cada una de ellas debe demostrarse en ambos sentidos.

1.5 Conceptos centrales de la teoría de autómatas

En esta sección introduciremos las definiciones de los términos más importantes que impregnan la teoría de autómatas. Entre estos conceptos están el de "alfabeto" (un conjunto de símbolos), "cadena" (una lista de símbolos pertenecientes a un alfabeto), y "lenguaje" (un conjunto de cadenas del mismo alfabeto).

1.5.1 Alfabetos

Un *alfabeto* es un conjunto finito no vacío de símbolos. Por convenio, se utiliza el símbolo Σ para representar un alfabeto. Entre los alfabetos más comunes se encuentran:

1. $\Sigma = \{0, 1\}$, el alfabeto *binario*.
2. $\Sigma = \{a, b, \ldots, z\}$, el conjunto de todas las letras minúsculas.
3. El conjunto de todos los caracteres ASCII o el conjunto de todos los caracteres ASCII imprimibles.

1.5.2 Cadenas

Una *cadena* (a veces llamada *palabra*) es una secuencia finita de símbolos pertenecientes a un alfabeto. Por ejemplo, 01101 es una cadena del alfabeto binario $\Sigma = \{0, 1\}$. La cadena 111 es otra cadena de este alfabeto.