UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

Autor: GILBERTO MANZANO PEÑALOZA

Análisis, diseño e implantación de algoritmos		Clave: 1164
Plan: 2005		Créditos: 8
Licenciatura: Informática		Semestre: 1°
Área: Informática (Desarrollo de sistemas)		Hrs. Asesoría: 4
Requisitos: Ninguno		Hrs. Por semana: 4
Tipo de asignatura:	Obligatoria (x)	Optativa ( )

Objetivo general de la asignatura

Al finalizar el curso, el alumno conocerá las técnicas más importantes para estudiar una amplia variedad de problemas y podrá utilizar estrategias algorítmicas para su solución.

Temario oficial (64 horas sugeridas)

1. Fundamentos de algoritmos (12 horas)
2. Análisis de algoritmos (12 horas)
3. Diseño de algoritmos para la solución de problemas (12 horas)
4. Implantación de algoritmos (12 horas)
5. Evaluación de algoritmos (16 horas)

Introducción

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.

La asignatura se ha dividido en cinco temas, en el **primero** se definen los conceptos necesarios para comprender los algoritmos y sus características, así como también 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.

El análisis del problema, los problemas computables y no computables, la recursividad y los algoritmos de ordenación y búsqueda; es el **segundo tema** en el que se estudian los problemas que se pueden resolver mediante la máquina de Turing, por lo que se les denomina problemas computables o decidibles, y aquellos que no se puedan solucionar por esta forma o tarde bastante su proceso por la complejidad del algoritmo 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 cuáles 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 analizaran los 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.

El **tercer tema** 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 tales como: algoritmos voraces, la programación dinámica, divide y vencerás, vuelta atrás y la ramificación y poda.

En el **cuarto tema** 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 tales 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 tales 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.

Y por último, en el **tema 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.

TEMA 1. FUNDAMENTOS DE ALGORITMOS

Objetivo particular

Al culminar el aprendizaje del tema, reconocerá la definición de un algoritmo y sus características, los conceptos generales de los autómatas y más específicamente la de los autómatas finitos deterministas.

Temario detallado

1.1 Definición de algoritmo

1.2 Autómatas y lenguajes formales

1.3 Máquina de Turing

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 lo 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 el autómata.

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 solo una actividad en una situación dada.

Es importante que el alumno, analice 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 el alumno ejercite un pensamiento lógico y abstracto sobre la forma de abordar los problemas de su área de desempeño que es la Informática.

1.1. Definición de algoritmo[1]

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

Como ejemplo tenemos el instructivo para armar un modelo de un avión a escala, si una persona sigue en forma estricta los pasos indicados en el instructivo, al final obtendrá como resultado el avión a escala, lo mismo obtendría otra persona que se dedicara a armar el mismo modelo.

Los pasos deben ser lo suficientemente detallados para que el procesador lo entienda, en el caso de nuestro ejemplo, el procesador es el cerebro de la persona que está armando el modelo, pero el ser humano tiende a obviar muchas cosas y es muy factible que el humano haga en forma automática algunos de los pasos del instructivo, sin detenerse a pensar mucho en cómo llevarlos a cabo, pero para una computadora resultaría imposible realizar la tarea, ya que la máquina requiere de instrucciones muy detallados para poder ejecutarlas.

---

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

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 muchas tareas todavía la computadora no las puede realizar.

Pasemos ahora a describir las características de un algoritmo, las cuáles son las siguientes:

- **Finito**. El algoritmo debe de 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 dos personas en distintos lugares se les indicara que prepararan 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 la 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 de 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 carece de cualquiera de estas características, entonces los pasos en cuestión no pueden considerarse como un algoritmo.

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

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

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

---

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

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 último 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 MathisonTuring.

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

Técnicamente existen diferentes herramientas para poder definir el comportamiento de un autómata, entre las cuales se encuentra el diagrama de estado. En él se pueden visualizar los estados como círculos que en su interior registran su significado. Existen flechas que representan la transición entre estados y la notación de Entrada/Salida que provoca la transición entre estados.

En el ejemplo de al lado se muestran cuatro diferentes estados de un autómata y se define lo siguiente: partiendo del estado "00", si se recibe una entrada "0", la salida es "0" y el autómata conserva el estado actual, pero si la entrada es "1", la salida será "1" y el autómata pasa al estado "01". Este comportamiento es homogéneo para todos los estados del autómata. Vale la pena

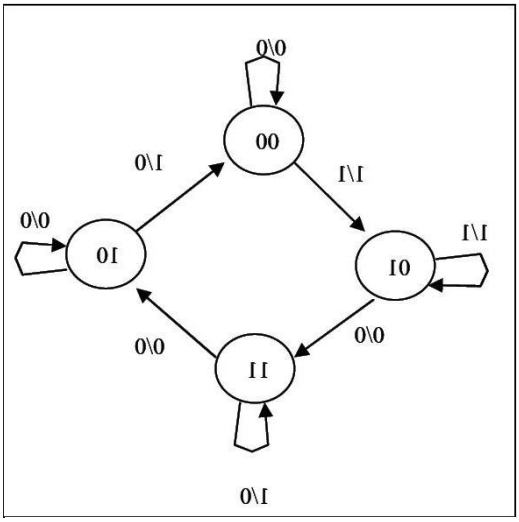

Figura 1.1. Diferentes estados de un autómata

resaltar que el autómata que se muestra aquí tiene un alfabeto binario (0 y 1).

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

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

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

$$y = A' x$$

Terminología de autómatas

En el siguiente apartado, abordaremos la terminología necesaria para la comprensión de los autómatas.

- **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 aquella que cumple con las reglas que define la gramática establecida.

- **Cadena vacía**

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

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

- **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.

- **Gramáticas formales**

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

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

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

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

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

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.

---

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

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

<raiz> = ARCHIVO

| EDICION

| OPCIONES

| AYUDA.

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

<raiz> = <opcion>.

<opcion> = ARCHIVO

| EDICION

| OPCIONES

| AYUDA.

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

<raiz> = Hola mundo | Hola todos.

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

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

<peticion> = <comando> | <titulo> <comando> .

<titulo> = Señor | Señora | Señorita.

<comando> = Me puede mostrar su licencia.

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

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

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

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

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

- **Lenguaje formal**

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

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

- **Jerarquización de gramáticas**

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

- **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 aquella que cumple con las propiedades de la gramática sensible al contexto y que, además, tiene la característica de que el lado izquierdo de la regla de producción sólo debe tener un elemento y éste no puede ser un elemento terminal.
- **Gramáticas regulares.** Es aquella que cumple con las características de la gramática independiente del contexto y, además, se restringe a través de las reglas de producción para generar sólo reglas de los dos tipos anteriores.

- **Propiedades de indecidibilidad**

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

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

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

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

1.3. 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

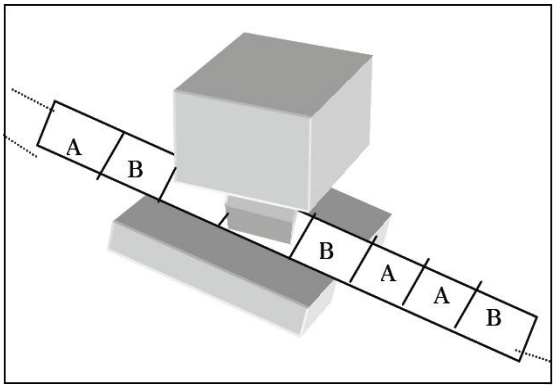

Figura 1.2. Máquina de Turing

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**

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

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

Hay problemas que pueden resolverse mediante una Máquina de Turing y otros 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:

- **Proceso computable:** es aquel que puede implementarse en un algoritmo o Máquina de Turing y definirse en un lenguaje decidible. Un proceso computable puede implementarse como el programa de la Máquina de Turing.
- **Proceso no computable:** es aquel que no puede implementarse con una Máquina de Turing por no tener solución para todas sus posibles entradas.

El lenguaje en que se especifica es un lenguaje indecidible que no puede ser interpretado por una Máquina de Turing.

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:

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

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

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

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

Si en esta MT se comienza con la cabeza de lectura / escritura sobre el 0 tenemos la siguiente secuencia:

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

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

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 *ccdddd*.

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)

$$\partial (q3, c) = (q1, c, D)$$
$$\partial (q5, a) = (q3, a, I)$$
$$\partial (q5, b) = (q3, d, I)$$
$$\partial (q5, c) = (q4, c, ALTO)$$

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, aaabbb) ├ (q2, caabbb) ├ (q2, caabbb) ├ (q2, caabbb) ├ (q2, caabbb) ├
(q2, caabbb) ├ (q2, caabbb ß) ├ (q5, caabbb ß) ├ (q3, caabbdß) ├ (q3, caabbd) ├
(q3, caabbd) ├ (q3, caabbd) ├ (q3, caabbd) ├ (q1, caabbd) ├ (q2, ccabbd) ├
(q2, ccabbd) ├ (q2, ccabbd) ├ (q2, ccabbd) ├ (q5, ccabbd) ├ (q3, ccabdd) ├
(q3, ccabdd) ├ (q3, ccabdd) ├ (q1, ccabdd) ├ (q2, cccbdd) ├ (q2, cccbdd) ├
(q5, cccbdd) ├ (q3, ccdddd) ├ (q1, cccddd) ├ (q4, cccddd)

ALTO.

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

Bibliografía del tema 1

Hopcroft, John, Rajeev Motwani, y J. D. Ullman, *Introducción a la teoría de autómatas, lenguajes y computación*, 2ª edición. Madrid, Pearson Addison Wesley, 2002, pp 584.

Manzano Peñaloza, Gilberto, *Tutorial para la asignatura Análisis, diseño e implantación de algoritmos*, 1ª edición, Fondo editorial FCA, México, 2003.

Direcciones electrónicas del tema 1

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

Actividades de aprendizaje

A.1.1. Consulta el libro de *Introducción a la Teoría de Autómatas, Lenguajes y Computación* de John E. Hopcroft, 2ª ed., Madrid, Pearson Addison Wesley, 2002 y realice las siguientes actividades:

Lee la unidad 1 "¿Para qué sirven los autómatas?" y da un ejemplo de situaciones en las que se pueden aplicar las siguientes demostraciones:

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

Envía la actividad en un documento a tu asesor.

A.1.2. Lee la unidad 2 en las pp. 41-58 y elabora un resumen de media cuartilla. Envía el documento a tu asesor.

A.1.3. Soluciona los ejemplos de la sección 2.2.6, pp. 58-60 del capítulo 2 "Autómatas finitos" y envíalos en un documento a tu asesor.

A.1.4. Diseña una MT para determinar si la cantidad de paréntesis de apertura y de cierre están o no balanceados. Ejemplo: Para la cadena de paréntesis ((()) la MT determinará que no están balanceados. Envía el diseño en un documento a tu asesor.

A.1.5. Dado el lenguaje $a^n b^m$ donde n=2 y m=3, diseña una MT para determinar si está contenido en la siguiente cadena: ababaaabbaabbba. Envía el diseño en un documento a tu asesor.

Cuestionario de autoevaluación

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 como 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

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

_____1. Es una característica de un algoritmo:

a. Acepta criterios en su desarrollo.
b. Se pueden omitir pasos al seguir el algoritmo.
c. En ocasiones, no obtiene un resultado.
d. Contiene una condición que detiene su ejecución.

_____2. Inventor del “El pato con aparato digestivo”:

a. Pierre Jacquet Droz
b. Falcon
c. Joseph Marie Jaquard
d. Jacques Vaucanson

_____3. Autómata que está formado por una cinta, una cabeza de lectura – escritura y un programa:

a. Máquina de Turing
b. El dibujante
c. Los músicos
d. El telar automático

_____4. Problema que no puede implementarse en una Máquina de Turing por no tener solución para todas sus posibles entradas:

a. Computable
b. Indecidible
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 sensibles al contexto
c. Gramáticas regulares
d. Ninguna de las anteriores

II. Relación de columnas. Escribe sobre la línea la opción que mejor complete la sentencia.

_____ 1. Conjunto de todos los símbolos válidos o posibles para una aplicación.

_____ 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. Una cadena que no tiene caracteres asociados.

_____ 4. Conjunto de cadenas que obedecen a un alfabeto fijado.

_____ 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.

a. Cadena vacía
b. Gramática
c. Lenguaje
d. Frase
e. Alfabeto

TEMA 2. ANÁLISIS DE ALGORITMOS

Objetivo particular

El alumno identificará los elementos de un problema, definiendo la factibilidad de su solución, a través de algoritmos que pueden invocarse a sí mismos de búsqueda y de ordenación

Temario detallado

2.1 Análisis del problema

2.2 Computabilidad

2.3 Algoritmos recursivos

2.4 Algoritmos de búsqueda y ordenación

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 a 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.

Así mismo 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.

Y por último, se estudiarán los diferentes métodos de ordenación y búsqueda, los cuáles 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 para que facilite la tarea del usuario de la información, a la vez que facilita su búsqueda y el acceso a un elemento determinado.

2.1 Análisis del problema

El análisis del problema es un proceso para recabar la información necesaria para emprender una acción que 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, p. ej. que el 50% de datos se encuentre ordenado y el 50% de datos restante no lo esté.
- Peor caso: datos de entrada más desfavorable, p. ej. 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 y por ende, entre ellos no sean comparables en absoluto.
- Algoritmos bastante complejos de los cuales 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, y así poder seleccionar el algoritmo que tenga el mejor rendimiento para un problema en particular.

2.2 Computabilidad[3]

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.

---

[3]Véase mi *Tutorial Análisis, diseño e implantación de algoritmos*, UNAM FCA, 2003.

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 un cierto nivel de conocimientos acerca de arte, podrá tener una interpretación distinta a la de otra persona con el mismo nivel cuando se observa una pintura; pero un receptor con un nivel de conocimientos matemáticos análogo al nivel de otro receptor siempre dará la misma interpretación a una expresión matemática. Esto se debe a que en el primer caso intervienen factores personales de interpretación, que hacen válidas las diferencias, mientras que en el segundo caso se tiene un cúmulo de reglas que no permiten variedad de interpretaciones sobre una misma expresión. En 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 aquel que no puede resolverse por un algoritmo en todos sus casos. Asimismo, su lenguaje asociado no puede ser reconocido por una Máquina de Turing.

2.3. Algoritmos recursivos[4]

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

---

[4]Véase mi *Tutorial Análisis, diseño e implantación de algoritmos*, UNAM FCA, 2003.

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

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

El funcionamiento de una función recursiva puede verse como:

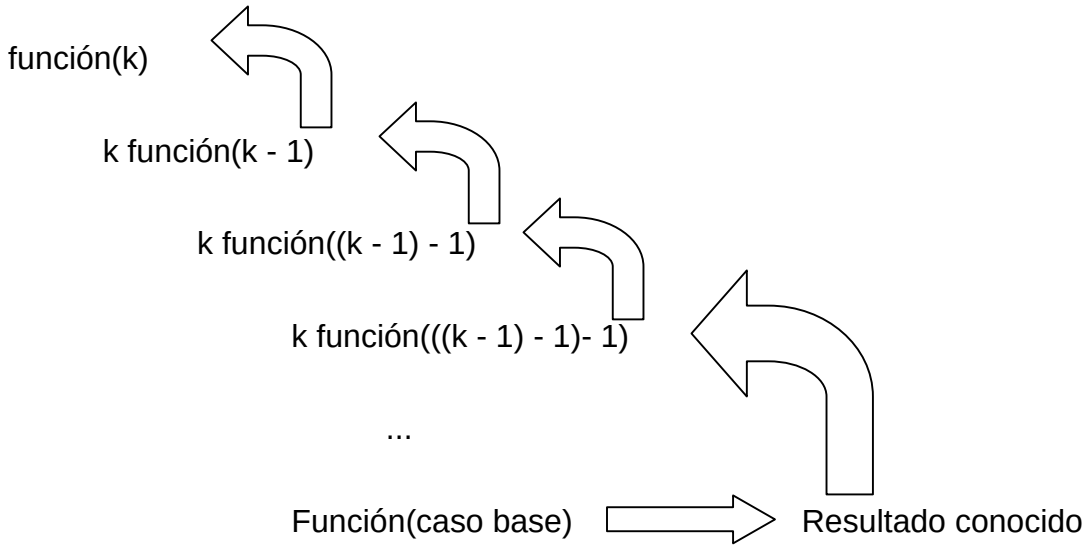

Figura 2.1. Esquema de una función recursiva

Introducción a la inducción

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

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

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

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

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

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 válida.

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

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

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

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

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.

2.4 Algoritmos de búsqueda y ordenación

Al utilizar matrices o bases de datos las tareas que más comúnmente se utilizan son la ordenación y/o la búsqueda de los datos para los cuáles existen diferentes métodos más o menos complejos según lo rápido o lo eficaz que sean.

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 nosotros 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 el 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 n-ésimo elemento.

Ejemplo:

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

Primera comparación:

Si (valor posición 1 > valor posición 2): 9 > 3? Verdadero, intercambiar.

Quedando como 3, 9, 4, 7 y 2

Segunda comparación:

Si (valor posición 2 > valor posición 3): 9 > 4? Verdadero, intercambiar.

Quedando como 3, 4, 9, 7 y 2

Si (valor posición 1 > valor posición 2): 3 > 4? Falso, no intercambiar.

Tercera comparación:

Si (valor posición 3 > valor posición 4): 9 > 7? Verdadero, intercambiar.

Quedando como 3, 4, 7, 9 y 2

Si (valor posición 2 > valor posición 3): 4 > 7? Falso, no intercambiar. Con esta circunstancia se interrumpen las comparaciones, puesto que ya no se realiza la comparación de la posición 2 con la 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

sub-problemas 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. El elemento "a" se procede a comparar 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 y 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 tiene 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".

Se toma el sub-arreglo izquierdo (los menores de "a") y se realiza el mismo procedimiento. Se toma el sub-arreglo derecho (los mayores de "a") y se realiza el mismo procedimiento. Este proceso se realiza hasta que los sub-arreglos sean de un elemento (solución).

Al final los sub-arreglos conformarán el arreglo *M* el cual contendrá elementos ordenados en forma ascendente.

- **Shell**

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

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

Métodos 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 este ordenado. El método consiste en dividir el arreglo por su elemento medio en dos sub-arreglos 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 sub-arreglo, 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 vacío {}, 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 y
- 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 el que las claves con muchos ceros se reflejarán en valores hash también con ceros, y el que el tamaño de la tabla está restringido a ser una potencia de 2.

Para evitar las restricciones anteriores se debe de calcular

$$h(k) = \text{entero } [Q * \text{Frac}(C*k)]$$

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.

Bibliografía del tema 2

Manzano Peñaloza, Gilberto, *Tutorial para la asignatura Análisis, diseño e implantación de algoritmos*, México, Fondo editorial FCA, 2003.

Sedgewick, Robert, *Algoritmos en C++*, México, Pearson Education, 1995.

Direcciones electrónicas

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

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

Actividades de aprendizaje

A.2.1. Desarrolla un mapa conceptual con el contenido del tema. Envíalo a tu asesor.

A.2.2. Investiga 5 ejemplos de problemas no decidibles, coméntalos en el foro.

A.2.3. Investiga las diferencias que existen entre la solución iterativa y la solución recursiva, coméntalas en el foro de la asignatura.

A.2.4. Investiga y comenta en el foro, la razón por la cual utilizarías o no, la recursividad en un cálculo de la serie de Fibonacci.

A.2.5. Realiza un cuadro comparativo con las características de los métodos de ordenación: burbuja, inserción, selección, quick sort y shell. Envíalo en un documento a tu asesor.

A.2.6. 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. Envíalo en un documento a tu asesor.

Cuestionario de auto evaluación

1. ¿Qué elementos se deben de 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 de 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 estas.

Examen de autoevaluación

I. Opción Múltiple. Escribe sobre la línea, la opción que mejor complete la sentencia.

1. En el tiempo de ejecución de un algoritmo, cuando el tipo de datos de entrada está parcialmente ordenado se refiere a:

a. Caso óptimo.
b. Caso medio.
c. Peor caso.
d. Ninguno de los anteriores.

2. Una función recursiva es aquella que:

a. Utiliza la estructura Mientras para realizar los ciclos iterativos.
b. Invoca a otras funciones para realizar ciertas tareas.
c. Contiene un caso base en donde se le asigna un resultado a la función.
d. No consume mucha cantidad de memoria y tiempo de procesador.

3. Se dice que un problema es no decidible cuando:

a. El algoritmo tiene varios casos que no pueden resolverse.
b. Se puede solucionar mediante la Máquina de Turing.
c. Sólo se resuelve mediante el empleo de la computadora.
d. Está implícita la recursión en su solución.

4. Cláusula de la función recursiva que establece la manera en que los elementos del dominio pueden ser combinados para generar los elementos del contradominio:

a. Básica.
b. Extrema.
c. Recursiva.
d. Inductiva.

5. Método de búsqueda que utiliza tablas de claves calculadas de registros para acceder a los datos:

a. Dicotómica.

b. Secuencial.

c. Hash.

d. Binaria.

II. Relación de columnas. Escribe sobre la línea la opción que mejor complete la sentencia.

_____ 1. Intercambia elementos que están muy distantes.

_____ 2. Emplea la técnica de “divide y vencerás” para separar el problema en sub-problemas más pequeños.

_____ 3. Se base en seleccionar el elemento más pequeño del arreglo y colocarlo en la posición más baja del arreglo.

_____ 4. Es el método más sencillo pero el menos eficiente.

_____ 5. Método que consiste en tomar un elemento y colocarlo en la posición ordenada correspondiente.

a. Burbuja

b. Selección

c. Shell

d. Inserción

e. Quick sort

TEMA 3. DISEÑO DE ALGORITMOS PARA LA SOLUCIÓN DE PROBLEMAS

Objetivo particular

El alumno utilizará las herramientas para la construcción y abstracción de algoritmos, así como también las técnicas de diseño de éstos.

Temario detallado

3.1 Niveles de abstracción para la construcción de algoritmos

3.2 Técnicas de diseño de algoritmos

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 sub-problema y la suma de estas soluciones es la solución del problema general.

Programación dinámica. Definen sub-problemas 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.

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.

El primer proceso, análisis, consiste en reconocer cada una de las características del problema, lo cual se logra señalando los procesos y variables que rodean al problema. Los procesos pueden identificarse como operaciones que se aplican a las variables del problema. Al analizar los procesos o funciones del problema, éstos deben relacionarse con sus variables. 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.

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.

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 y HASTA QUE. El primero se caracteriza por realizar la verificación de la condición antes de ejecutar las instrucciones asociadas al ciclo; el segundo evalúa la condición después de ejecutar las instrucciones una vez. Las instrucciones definidas dentro de ambos ciclos deben modificar, en algún punto, la condición para que sea alcanzable, de otra manera serían ciclos infinitos, un error de programación común.

En este tipo de ciclos el número de iteraciones que se realizarán es variable y depende del contexto de ejecución del algoritmo.

El ciclo MIENTRAS tiene el siguiente pseudocódigo:

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

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

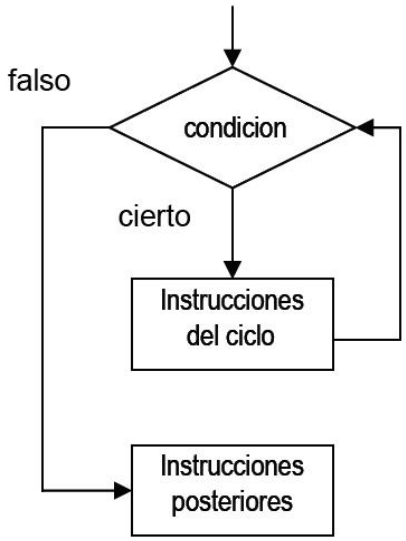

Figura 3.1. Diagrama asociado al ciclo mientras

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

hacer
Instruccion1
Instruccion2
...
Instrucción n
Hasta que <condicion>

Su diagrama se puede representar como:

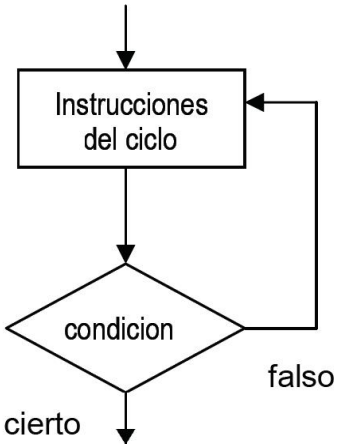

Figura 3.2. Diagrama asociado al ciclo hasta que

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 esta 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
    Instruccion1
    Instruccion2
    ...
    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:

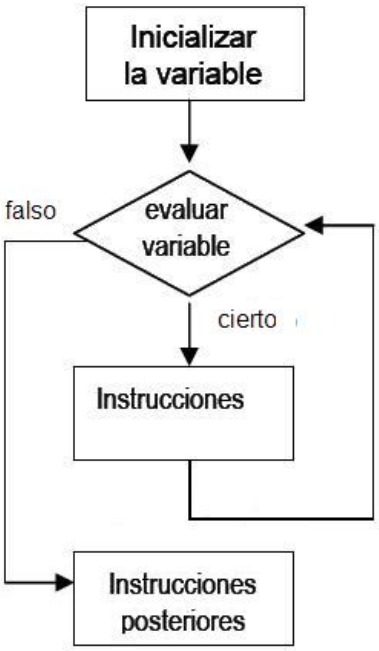

Figura 3.3. Diagrama de contadores

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

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

El diagrama asociado a esta estructura se muestra a continuación:

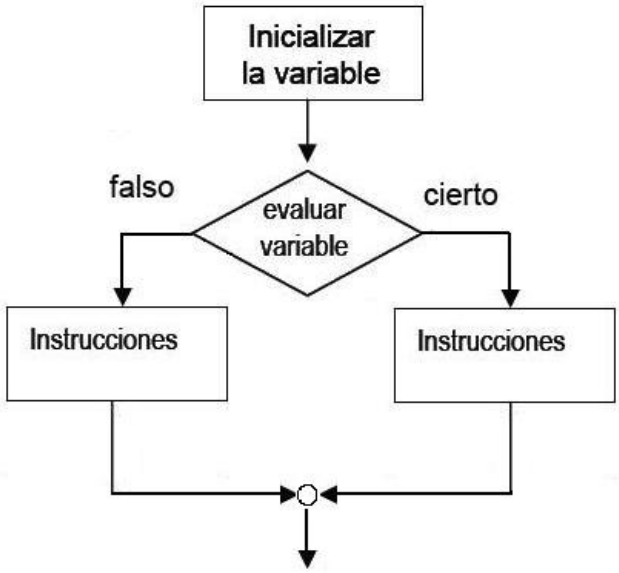

Figura 3.4. Diagrama de condiciones

Rutinas recursivas

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

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

Las rutinas recursivas regularmente contienen una cláusula condicional (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 2.1 de funciones recursivas.

3.2 Técnicas de diseño de algoritmos

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.

Existe una función 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.

Existe 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, etc. Habitualmente, los elementos que intervienen son:

- un conjunto o lista de candidatos (tareas a procesar, vértices del grafo, etc.);
- un conjunto de decisiones ya tomadas (candidatos ya escogidos);
- una función que determina si un conjunto de candidatos es una solución al problema (aunque no tiene por qué ser la óptima);
- una función que determina si un conjunto es completable, es decir, si añadiendo a este conjunto nuevos candidatos es posible alcanzar una solución al problema, suponiendo que esta exista;
- una función de selección que escoge el candidato aún no seleccionado que es más prometedor;

- una función objetivo que da el valor/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 el de “divide y vencerás” y 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 todos ellos 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:

Dividir el problema en subproblemas más pequeños.

Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.

Usar estas soluciones óptimas para construir una solución óptima al problema original.

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

Vuelta Atrás (*Back Tracking*)

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 a 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 puede 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 (*Backtracking*) 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 Ultima Entrada Primera Salida) o 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 solución hallada hasta el momento, no continua explorando esa rama del árbol, lo cuál 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 que 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:

1. **Selección**: extrae un nodo de entre el conjunto de los nodos vivos.
2. **Ramificación**: se construyen los posibles nodos hijos del nodo seleccionado en la etapa anterior.
3. **Poda**: Se eliminan algunos de los nodos creados en la etapa anterior. Aquellos 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.

El 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 de que garantice la poda y que su cálculo no sea muy costoso.

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

Estos algoritmos tienen la posibilidad de ejecutarlos 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 cuál consiste en colocar 8 reinas en un tablero de ajedrez cuyo tamaño es de 8 por 8 cuadros, las reinas deben de 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.

Bibliografía del tema 3

De Giusti, Armando E., *Algoritmos, datos y programas con aplicaciones en Pascal, Delphi y Visual Da Vinci*, Buenos Aires, Pearson Education, 2001.

Joyanes Aguilar Luis, *Estructuras de datos, algoritmos, abstracción y objetos*, México, McGraw Hill, 1998.

Manzano Peñaloza Gilberto, *Tutorial para la asignatura Análisis, diseño e implantación de algoritmos*, México, Fondo editorial FCA, 2003.

R.C.T. Lee, S.S.Tseng, R.C. Chang, Y.T. Tsai, *Introducción al diseño y análisis de algoritmos un enfoque estratégico*, México, McGraw Hill, 2007.

Sedgewick, Robert, *Algoritmos en C++*, México, Adisson-Wesley Iberoamericana, 1995.

Van Gelder, Baase, *Algoritmos computacionales Introducción al análisis y diseño*, 3ª ed., México, Thomson, 2002.

Actividades de aprendizaje

A.3.1. Elabora un mapa conceptual del contenido del tema 3. Envíalo en un documento a tu asesor.

A.3.2. Diseña un algoritmo para dar solución a un problema que tu propongas en donde se ocupen todas las estructuras de control: Mientras, Hasta que, Si entonces Sino y el contador Para. Envíalo en un documento a tu asesor.

A.3.3. Diseña un algoritmo que calcule la potencia de un número sin utilizar la operación de multiplicación o la función de potencia, utilizando la estructura Mientras. Envíalo en un documento a tu asesor.

A.3.4. Elabora un cuadro comparativo de las diferentes técnicas de diseño de algoritmos. Envíalo en un documento a tu asesor.

A.3.5. 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.

A.3.6. Investiga el juego de las torres de Hanoi y diseña las funciones recursivas necesarias para su ejecución. Envíalo en un documento a tu asesor.

Cuestionario de auto evaluación

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 de “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. ¿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?

Examen de autoevaluación

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

_____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 incremento alguno, sólo es de control.

c. Aumenta su valor con el valor propio más el del incremento.

d. No tiene algo que ver con la solución arrojada por el algoritmo.

_____3. Cuando se sabe con exactitud el número de iteraciones que debe de 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. Hacer ...Hasta que

d. Si entonces si no

_____5. Técnica de diseño de algoritmos que contiene 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

II. Relación de columnas. Escribe la opción que mejor complete la sentencia o responda la pregunta.

_____ 1. Divide el problema en sub-problemas que resuelve recursivamente para finalmente reunir las soluciones individuales.	a. Algoritmos voraces
_____2. Resuelve el problema en su conjunto a través de subestructuras óptimas.	b. Divide y vencerás
_____3. Técnica en que esencialmente encuentra la mejor combinación en un momento determinado (búsqueda en profundidad).	c. Programación dinámica
_____4. Se utilizan en solución de problemas de optimización, aunque son poco eficientes.	d. Vuelta atrás

TEMA 4. IMPLANTACIÓN DE ALGORITMOS

Objetivo particular

El alumno utilizará las estructuras de control de la programación estructurada y manipulará la modularidad en el diseño descendente y ascendente de un sistema de información.

Temario detallado

4.1 El programa como una expresión computable del algoritmo

4.2 Programación estructurada

4.3 Modularidad

4.4 Enfoque de algoritmos

Introducción

En este tema se abordará 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 estudiarán las **estructuras de control** básicas a las que hace referencia el **Teorema de la Estructura** y que son piezas clave en la **programación estructurada** cuya principal característica es la de 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 analizarán los dos enfoques de diseño de sistemas: el **refinamiento progresivo** y el **procesamiento regresivo** comparando sus ventajas y limitaciones.

4.1. El programa como una expresión computable del algoritmo

Como ya se ha mencionado, el algoritmo es una secuencia lógica y detallada de pasos para solucionar un problema. Una vez diseñada la solución se 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 revisemos desde el principio la manera en como un algoritmo se convierte en un programa de computadora:

1. Definición del algoritmo. Es el enunciado del problema, para saber qué se espera que haga el programa.
2. Análisis del algoritmo. Para resolver el problema, debemos estudiar las salidas que se esperan del programa, para definir las entradas requeridas. También se deben de bosquejar los pasos a seguir por el algoritmo.
3. 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.
4. Diseño del algoritmo. Se diagraman los pasos del problema. También se puede utilizar el pseudocódigo, como la descripción abstracta del problema.
5. 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.
6. Codificación. Traducimos cada gráfico del diagrama o línea del pseudocódigo a una instrucción de algún lenguaje de programación. El código fuente lo guardamos en archivo electrónico.
7. 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. 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 enlazará con alguna librería de programas (edición de enlace) y obtendrá un archivo ejecutable.

8. 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.

9. 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 listarán 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)

SINO

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 de 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.
- Reducción del 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 el cual 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[5], de la forma: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 y 89, es un ejemplo de la programación estructurada:

PSEUDOCÓDIGO	**PROGRAMA FUENTE EN LENGUAJE C.**

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

Cuadro 4. 1. Ejemplo de un programa estructurado

Como se aprecia en la cuadro 4.1, 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.

[5] 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 este.

4.3. Modularidad

Un problema se puede dividir en sub-problemas más sencillos. Estos sub-problemas se conocen como módulos. Dentro de los programas se les conoce como sub-programas 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. Enfoque de algoritmos

Existen dos enfoques que se refieren a la forma en que 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.

Top 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 sub-algoritmos 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 este 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 la siguiente gráfica para un sistema de nómina:

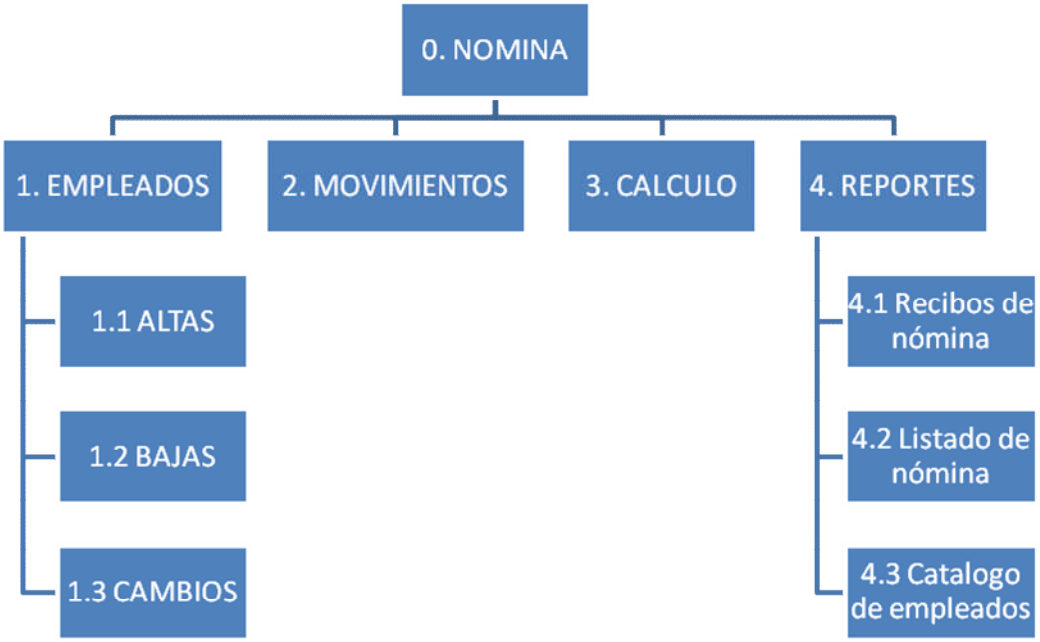

Figura 4.1. Diseño top down de un sistema de nómina

Como se puede observar en la figura 4.1, el diseño descendente es jerárquico, el módulo 0 de Nómina contendrá el menú principal que integrará al sistema, controlando desde este, los sub-menús del siguiente nivel. El módulo número 1 de empleados, contendrá un sub-menú 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 sub-programas para consultar en pantalla e imprimir los recibos de nómina, la nómina misma y un catalogo de empleados, aunque no es limitativo, puesto que se le pueden incluir más reportes o informes al sistema.

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

Procesamiento regresivo

Esta 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 sub-algoritmos 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 sub-algoritmos al grado tal de que el desempeño global, sea fluido. Los problemas de integración entre los sub-algoritmos no se solucionan hasta que la programación alcanza la fecha límite para la integración total del programa.

Aunque cada sub-algoritmo parece ofrecer lo que se requiere, cuando se contempla al programa final, 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 de algoritmo no fue completamente considerado y en consecuencia no se satisface plenamente.
- A diferencia del diseño descendente en donde si 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.

Bibliografía del tema 4

Lozano, Letvin, *Diagramación y Programación Estructurada y Libre*, 3ª edición, México, McGraw-Hill, 1986, 384 pp.

Actividades de aprendizaje

A.4.1 A partir del estudio de la bibliografía específica sugerida, elabora un mapa de este tema. Envíalo en un documento a tu asesor.

A.4.2 Investiga qué otras estructuras de control existen que se deriven de las básicas explicadas en este apunte. Coméntalo en el foro de la asignatura.

A.4.3 Desarrolla un diagrama top down y uno de bottom up para un sistema de inventarios. Envíalo en un documento a tu asesor.

A.4.4. Investiga en una empresa qué tipo de enfoque aplica para desarrollar sus sistemas de información. Sube al foro tus comentarios.

Cuestionario de autoevaluación

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?

Examen de autoevaluación

Verdadero Falso. Escribe sobre la línea una V por verdadero o una F por falso dependiendo de la sentencia.

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, sólo se tiene 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 desvío del flujo de control. ____
6. Sólo hay un tipo de modulo y esta es la función. ____
7. Un procedimiento devuelve una tarea y una función un resultado. ___
8. El refinamiento progresivo contiene al procedimiento de Bottom Up ___

9. El enfoque de diseño descendente es el procedimiento más costoso pero el más eficiente para integrar los módulos de un sistema. ___
10. El Bottom Up tiene la limitación de duplicar esfuerzos al introducir datos ya que se introducen al sistema muchos datos carentes de valor. ___

TEMA 5. EVALUACIÓN DE ALGORITMOS

Objetivo particular

El alumno identificará un algoritmo que sea la solución más eficiente del problema en cuestión. Documentará el algoritmo para futuras revisiones y llevará a efecto el mantenimiento preventivo, correctivo y adaptativo para su óptima operación.

Temario detallado

5.1. Refinamiento progresivo

5.2. Depuración y prueba

5.3. Documentación del programa

5.4. Mantenimiento de programas

Introducción

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

Entre las técnicas más confiables se encuentran aquellas 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 de dar el adecuado mantenimiento, con este propósito se analizarán los diferentes tipos de mantenimiento: preventivo, correctivo y adaptativo.

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 estas 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 por ejemplo: 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 algoritmo (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 por ejemplo en los algoritmos de ordenación, el peor caso es que las entradas se encuentren totalmente desordenadas, en el

mejor de los casos que se encuentren totalmente ordenadas y en el caso promedio que estén parcialmente ordenadas, veamos como ejemplos a los algoritmos de ordenación por inserción y el 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 aleatoriamente.

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 $O(n^2)$, 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 aleatoriamente.

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 $O(n^2)$.

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.

Se dice que un programa está depurado cuando está libre de errores. Cuando se depura un programa se hace un seguimiento del funcionamiento de dicho programa 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 errores, 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 que tal vez se omitió alguna puntuación necesaria para el programa. Existen también los errores lógicos, 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 y hay que 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:

1. Todos los registros que se incluyen en los datos de prueba.
2. Todos los cálculos efectuados por el programa.
3. Todos los campos del registro cuyo valor determine una acción a seguir dentro de la lógica del programa.
4. Todos los campos que el programa actualice.
5. Los casos en que haya comparación contra otro archivo.
6. Todas las condiciones especiales del programa.

7. Se cotejará la lógica del programa.

5.3. Documentación del programa

El objetivo de la documentación de programas es familiarizar a analistas y programadores con lo que hace cada programa en particular.

La documentación de programas es una extensión de la documentación del sistema. El programador convierte las especificaciones de programas en lenguaje de la computadora. El programador deberá 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.

La documentación detallada de cada programa deberá incluir los siguientes elementos que apliquen:

1. Nombre del Programa (código). Indicará código que identifica el programa y el título del programa.
2. Descripción. Indicará la función que realiza el programador.
3. Frecuencia de Procesamiento. Diaria, semanal, quincenal, mensual, etcétera.
4. Fecha de Vigencia. Fecha a partir de la cual se comienza a ejecutar en producción la versión modificada o desarrollada del programa.
5. Archivos de Entrada.
6. Lista de Archivos de Salida: Indicará el nombre y copia y descripción de los archivos.
7. 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.
8. Datos de Prueba. Se incluirá una copia de los datos usados para prueba.

9. Mensajes al Operador - Pantallas la definición de todos los mensajes al operador por consola y las posibles contestaciones con una breve explicación de cada una de ellas.
10. Datos de Control para ejecutar el programa (parámetros).
11. Transacciones.
12. Nombre del Programador Deberá indicar el nombre del programador que escribió el programa o que efectuó el cambio, según sea el caso.
13. Fecha. Indicará fecha en que se escribió el programa o que se efectuó el cambio, según sea el caso.
14. 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.
15. Lista de Programas. Deberá incluir copia de la última compilación del programa con todas las opciones.

5.4. Mantenimiento de programas

Los usuarios de los programas solicitarán los cambios necesarios al área de sistemas con el fin de que 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 debe realizar las pruebas al sistema, imprimiendo los reportes que genera y verificar si los cálculos que estos 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 de 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.

Bibliografía del tema 5

R.C.T. Lee, S.S.Tseng, R.C. Chang, Y.T. Tsai, *Introducción al diseño y análisis de algoritmos un enfoque estratégico*, México, McGraw Hill, 2007, 752 pp.

De Giusti, Armando E., *Algoritmos, datos y programas con aplicaciones en Pascal, Delphi y Visual Da Vinci,* Buenos Aires, Pearson Education, 2001, 472 pp.

Actividades de aprendizaje

A.5.1. A partir del estudio de la bibliografía específica sugerida, elabora un mapa conceptual con los temas de la unidad. Envía el documento a tu asesor.

A.5.2. Elabora un cuadro comparativo de evaluación de métodos de ordenación para determinar su eficiencia con base en la complejidad de sus algoritmos. Envía el documento a tu asesor.

A.5.3. Elabora un mini manual para documentar programas. Envía el documento a tu asesor.

A.5.4. Investiga en una empresa, el procedimiento para llevar a cabo el mantenimiento de programas. Coméntalo en el foro de la asignatura.

Cuestionario de autoevaluación

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 de 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

Verdadero / Falso. Escribe sobre la línea una letra V si la sentencia es verdadera o una F si es Falsa.

_____ 1.En la evaluación de algoritmos solamente debe de considerarse el tiempo de proceso.

_____2. Para medir la complejidad de un algoritmo no es necesario utilizar funciones matemáticas.

_____3. El término “debugging” significa eliminación de bichos.

_____ 4. Un error lógico es cuando un programa tiene errores de sintaxis.

_____ 5. El compilador es un programa que facilita la detección y corrección de errores.

_____ 6. Para realizar pruebas al programa se debe de utilizar cualquier tipo de datos tanto correctos como incorrectos.

_____ 7. En las pruebas al programa se deben verificar todos los cálculos que el programa realice.

_____ 8. El objetivo de la documentación de programas es familiarizar al usuario final con lo que hacen los programas.

_____ 9. El programador puede aplicar su criterio para cualquier cambio que se presente en las especificaciones de programa.

_____ 10. No es necesario incluir el diccionario de datos en la documentación de programas.

Bibliografía básica

De Giusti, Armando E., *Algoritmos, datos y programas con aplicaciones en Pascal, Delphi y Visual Da Vinci*, Buenos Aires, Pearson Education, 2001, 472 pp.

Hopcroft, John, Rajeev Motwani, y J. D. Ullman, *Introducción a la teoría de autómatas, lenguajes y computación*, 2ª edición. Madrid, Pearson Addison Wesley, 2002, pp. 584.

Joyanes Aguilar, Luis, *Estructuras de datos, algoritmos, abstracción y objetos*, México, McGraw Hill, 1998.

Lozano, Letvin, *Diagramación y Programación Estructurada y Libre*, 3ª edición, México, McGraw-Hill, 1986, 384 pp.

Manzano Peñaloza, Gilberto, *Tutorial para la asignatura Análisis, diseño e implantación de algoritmos*, Fondo editorial FCA, México, 2003.

R.C.T. Lee, S.S.Tseng, R.C. Chang, Y.T. Tsai, *Introducción al diseño y análisis de algoritmos un enfoque estratégico*, México, McGraw Hill, 2007, 752 pp.

Sedgewick, Robert, *Algoritmos en C++*, México, Pearson Education, 1995.

Van Gelder, Baase, *Algoritmos computacionales Introducción al análisis y diseño*, 3ª ed., México, Thomson, 2002.

Sitios Web

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

RESPUESTAS A LOS EXÁMENES DE AUTOEVALUACIÓN

Análisis, diseño e implementación de algoritmos

	Tema 1		Tema 2		Tema 3	
	I	**II**	**I**	**II**	**I**	**II**
1.	d	1. e	b	c	b	b
2.	d	2. d	c	e	c	c
3.	a	3. a	a	b	a	d
4.	b	4. c	d	a	d	a
5.	b	5. b	c	d	a	
6.					b	

	Tema 4	Tema 5
1.	F	F
2.	F	F
3.	V	V
4.	V	F
5.	F	V
6.	F	V
7.	V	V
8.	F	F
9.	V	F
10.	V	F