Algoritmos computacionales

Índice

1. Introducción

En el siguiente trabajo pretendemos presentar una serie de concepto y definiciones propios del estudio de los Algoritmos, su análisis y diseño.

En el mismo podremos encontrar los conceptos de algoritmo y algunos de sus componentes, análisis y diseño. También veremos los diferentes tipos de formas y tamaños o medidas en que se pueden almacenar y representar los datos y estructuras en un algoritmo o programa. En ese mismo orden encontraremos las diferentes técnicas para diseñarlos como son el método de la fuerza bruta, el voraz, divide y vencerás, programación dinámica, de vuelta atrás, entre otros.

De igual forma podremos ver las definiciones y algunas características, reglas, normas, tipos de algoritmos de búsqueda y ordenación así como sus aplicaciones.

Finalmente veremos los que es la verificación y derivación de programas, donde daremos los conceptos básicos de semántica y sus tipos haciendo mayor énfasis en la semántica axiomática, la recursividad e iteración, los diseños de estos últimos, así como los típicos ciclos utilizados en algoritmos y programas y los paso a tener en cuenta al momento de desarrollar un algoritmo iterativo o recursivo.

Justificación

Es importante el estudio y conocimiento de lo que hoy conocemos como Algoritmos Computacionales, que desde su aparición hasta nuestros días es, y seguirá siendo; vital para el desarrollo de aplicaciones para computadoras y el manejo y dominio de la lógica de programación para resolver problemas.

Motivación

Como estudiantes de la Facultad de Ciencias y Tecnología Escuela de Informática y Computación de la Universidad Dominicana Organización y Métodos O&M con aspiraciones de iniciarnos como Ingeniero en Sistemas y Computación. Con el objetivo inmediato de aprobar con los mejores meritos la asignatura de Algoritmos Computacionales.

Objetivos

General :

Posibilitar la estudiante alcanzar una visión sistemática de lo que conocemos sobre Los Algoritmos Computacionales.

Específicos :

Introducir los conceptos propios sobre Algoritmo, su importancia en el mundo de las aplicaciones para computadoras y el manejo de lógica de programación.

ð Proporcionar una idea de su uso.

ð Visualizar sus ventajas e importancia.

ð Definir sus tipos y variantes.

ð Proporcionar conceptos sobre su análisis y diseño.

ð Proporcionar concepto sobre las técnicas de diseño.

ð Desglosar sus variantes (ordenación, búsqueda, etc.).

2. Marco Histórico

Un algoritmo es un conjunto de operaciones y procedimientos que deben seguirse para resolver un problema. La palabra algoritmo se deriva del nombre latinizado del gran Matemático Árabe Mohamed Ibn Al Kow Rizmi, el cual escribió sobre los años 800 y 825 su obra Quitad Al Mugabala, donde se recogía el sistema de numeración hindú y el concepto del cero. Fue Fibinacci, el que tradujo la obra al latín y el inicio con la palabra: Algoritmi Dicit.

El lenguaje algorítmico es aquel por medio al cual se realiza un análisis previo del problema a resolver y encontrar un método que permita resolverlo. El conjunto de todas las operaciones a realizar y e orden en que se deben efectuarse, se le denomina algoritmo.

Es un método para resolver un problema mediante una serie de datos precisos, definidos y finitos.

3. Generalidades

El programador de computadoras es ante que nada una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático. A la metodología necesaria para resolver problemas mediante programas se denomina Metodología de la Programación. El eje central de esta metodología es el concepto, ya tratado, de algoritmo.

Un algoritmo es un método para resolver un problema. Aunque la popularización del término ha llegado con el advenimiento de la era informática, algoritmo proviene de Mohammed al−Khowarizmi, matemático persa

que vivió durante el siglo IX y alcanzo gran reputación por el enunciado de las reglas para sumar, restar, multiplicar y dividir números decimales; La traducción al latín del apellido de la palabra algorismus derivo posteriormente en algoritmo. Euclides, el gran matemático griego (del siglo IV antes de Cristo) que invento un método para encontrar el máximo común divisor de dos números, se considera con Al−Khowarizmi el otro gran padre de la algoritmia (ciencia que trata de los algoritmos).

El profesor Niklaus Wirth, inventor de Pascal, Modula−2 y Oberon, titulo uno de sus más famosos libros, Algoritmos + Estructuras de Datos = Programas, significándonos que solo se puede llegar a realizar un buen programa con el diseño de un algoritmo y una correcta estructura de datos. Esta ecuación será de una de las hipótesis fundamentales consideradas en esta obra.

La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto.

Los pasos para la resolución de un problema son:

- Diseño de algoritmo, que describe la secuencia ordenada de pasos que conducen a la solución de un problema dado. (Análisis del problema y desarrollo del algoritmo).
- Expresar el algoritmo como un programa de lenguaje de programación adecuado. (Fase de codificación.)
- Ejecución y validación del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de algoritmo, de modo que sin algoritmo no puede existir un programa.

Los algoritmos son independientes tanto del lenguaje de programación en que se expresan como de la computadora que lo ejecuta. En cada problema el algoritmo se puede expresar en un lenguaje diferente de programación y ejecutarse en una computadora distinta; sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, ingles o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizaran sin importar el idioma del cocinero.

En la ciencia de la computación y en la programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan solo un medio para expresar un algoritmo y una computadora es solo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente.

Dada la importancia del algoritmo en la ciencia de la computación, un aspecto muy importante será el diseño de algoritmos. El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la programación. En esencia, la solución de un problema se puede expresar mediante un algoritmo.

Características de los Algoritmos:

Las características fundamentales que debe cumplir todo algoritmo son:

ð Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.

ð Un algoritmo debe estar definido. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.

ð Un algoritmo debe ser finito. Si se sigue un algoritmo se debe terminar en algún momento; o sea, debe tener un numero finito de pasos.

La definición de un algoritmo debe definir tres partes: Entrada, Proceso y Salida. En el algoritmo de receta de cocina citado anteriormente se tendrá:

Entrada: ingrediente y utensilios empleados.

Proceso: elaboración de la receta en la cocina.

Salida: terminación del plato (por ejemplo, cordero).

Ejemplo de Algoritmo:

Un cliente ejecuta un pedido a una fábrica. Esta examina en su banco de datos la ficha del cliente; si el cliente es solvente entonces la empresa acepta el pedido; en caso contrario rechazara el pedido. Redactar el algoritmo correspondiente.

Los pasos del algoritmo son:

- inicio
- leer el pedido
- examinar la ficha del cliente
- si el cliente es solvente aceptar pedido; en caso contrario, rechazar pedido
- fin

Diseño del Algoritmo:

En la etapa de análisis del proceso de programación se determina que hace el programa. En la etapa de diseño se determina como hace el programa la tarea solicitada. Los métodos mas eficaces para el proceso de diseño se basan en el conocido por Divide y Vencerás, es decir, la resolución de un problema complejo se realiza dividiendo el problema en sub− problemas y a continuación dividir estos sub problemas en otros de nivel mas bajo, hasta que pueda ser implementada una solución en la computadora. Este método se conoce técnicamente como diseño descendente (Top Down) o modular. El proceso de romper el problema en cada etapa y expresar cada paso en forma más detallada se denomina refinamiento sucesivo.

Cada sub programa es resuelto mediante un modulo (sub programa) que tiene un solo punto de entrada y un solo punto de salida.

Cualquier programa bien diseñado consta de un programa principal (el modulo de nivel mas alto) que llama a sub programas (módulos de nivel mas bajo) que a su vez pueden llamar a otros sub programas. Los programas estructurados de esta forma se dice que tienen un diseño modular y el método de romper el programa en módulos más pequeño se llama Programación Modular. Los módulos pueden ser planeados, codificados, comprobados y depurados independientemente (incluso por diferentes programadores) y a continuación combinarlos entre si. El proceso implica la ejecución de los siguientes pasos hasta que el programa se termina:

ð programar modulo.

ð Comprobar el modulo.

ð Si es necesario, depurar el modulo.

ð Combinar el modulo con los módulos anteriores.

El proceso que convierte los resultados del análisis del problema en un diseño modular con refinamiento

sucesivo que permitan una posterior traducción al lenguaje se denomina diseño de algoritmo.

El diseño del algoritmo es independiente del lenguaje de programación en el que se vaya a codificar posteriormente.

4. Análisis De Algoritmos

Recursos De Computadores Y Complejidad

Algoritmo: Conjunto de reglas para resolver un problema. Su ejecución requiere unos recursos.

Un algoritmo es mejor cuantos menos recursos consuma, su facilidad de programarlo, corto, fácil de entender, robusto, etc.

Criterio empresarial: Maximizar la eficiencia.

Eficiencia: Relación entre los recursos consumidos y los productos conseguidos.

Recursos consumidos:

Tiempo de ejecución.

Memoria principal:

Entradas/salidas a disco.

Comunicaciones, procesadores, etc.

Lo que se consigue:

Resolver un problema de forma exacta, forma aproximada o algunos casos.

Recursos consumidos:

Ejemplo. ¿Cuántos recursos de tiempo y memoria consume el siguiente algoritmo sencillo?

```
i:= 0

a[n+1]:= x

repetir

i:= i + 1

hasta a[i] = x
```

Respuesta: Depende.

¿De qué depende? De lo que valga **n** y **x**, de lo que haya en **a**, de los tipos de datos, de la máquina...

En general los recursos dependen de:

Factores externos**.**

El ordenador donde lo ejecutemos: 286, Pentium III, Cray,...

El lenguaje de programación y el compilador usado.

La implementación que haga el programador del algoritmo. En particular, de las estructuras de datos utilizadas.

Tamaño de los datos de entrada.

Ejemplo. Calcular la media de una matriz de NxM.

Contenido de los datos de entrada.

Mejor caso. El contenido favorece una rápida ejecución.

Peor caso. La ejecución más lenta posible.

Caso promedio. Media de todos los posibles contenidos.

Los factores externos no aportan información sobre el algoritmo.

Conclusión: Estudiar la variación del tiempo y la memoria necesitada por un algoritmo respecto al tamaño de la entrada y a los posibles casos, de forma aproximada (y parametrizada).

externos no aportan información sobre el algoritmo.

Normalmente usaremos la notación T(N)=..., pero ¿qué significa T(N)?

Tiempo de ejecución en segundos. T(N) = bN + c.

Suponiendo que b y c son constantes, con los segundos que tardan las operaciones básicas correspondientes.

Instrucciones ejecutadas por el algoritmo. T(N) = 2N + 4.

¿Tardarán todas lo mismo?

Ejecuciones del bucle principal. T(N) = N+1.

¿Cuánto tiempo, cuántas instrucciones,...?

Sabemos que cada ejecución lleva un tiempo constante, luego se diferencia en una constante con los anteriores.

Asignación de tiempos, para el conteo de instrucciones. Algunas reglas básicas.

Operaciones básicas (+, −, *, :=,...): Una unidad de tiempo, o alguna constante.

Operaciones de entrada salida: Otra unidad de tiempo, o una constante diferente.

Bucles FOR: Se pueden expresar como una sumatoria, con los límites del FOR.

IF y CASE: Estudiar lo que puede ocurrir. Mejor caso y peor caso según la condición. ¿Se puede predecir cuándo se cumplirán las condiciones?

Llamadas a procedimientos: Calcular primero los procedimientos que no llaman a otros.

Bucles WHILE y REPEAT: Estudiar lo que puede ocurrir. ¿Existe una cota inferior y superior del número de ejecuciones? ¿Se puede convertir en un FOR?

El análisis de algoritmos también puede ser a posteriori: implementar el algoritmo y contar lo que tarda para distintas entradas.

Ejemplo. Programa cifras.exe:

N= 4, T(4)= 0.1 ms

N= 5, T(5)= 5 ms

N= 6, T(6)= 0.2 s

N= 7, T(7)= 10 s

N= 8, T(8)= 3.5 min

¿Qué conclusiones podemos extraer?

Análisis a priori: Evitamos la implementación, si el algoritmo es poco eficiente. Podemos hacer previsiones. Podemos comparar con otros algoritmos.

Medidas Asintoticas

Notación asintótica:

El tiempo de ejecución T(n) está dado en base a unas constantes que dependen de factores externos.

Nos interesa un análisis que sea independiente de esos factores.

Notaciones asintóticas: Indican como crece T, para valores suficientemente grandes (asintóticamente) sin considerar constantes.

O(T): Orden de complejidad de T.

W(T): Orden inferior de T, u omega de T.

Q(T): Orden exacto de T.

Orden de complejidad de f(n): O(f)

Dada una función f: N ® R+, llamamos orden de f al conjunto de todas las funciones de N en R+ acotadas superiormente por un múltiplo real positivo de **f**, para valores de **n** suficientemente grandes.

O(f)= { t: N ® R+ / **$** c Î R+, **$** n0 Î N, " n ³ n0: t(n) £ c" f(n) }

Nota:

O(f) es un conjunto de funciones, no una función.

Valores de n sufic. Grandes...: no nos importa lo que pase para valores pequeños.

Funciones acotadas superiormente por un múltiplo de f...: nos quitamos las constantes.

La definición es aplicable a cualquier función de N en R, no sólo tiempos de ejec.

Propiedades

P1. Si f Î O(g) y g Î O(h) entonces f Î O(h).

Si f Î W(g) y g Î W(h) entonces f Î W(h)

Ej. 2n+1 Î O(n), n Î O(n2) Þ 2n+1 Î O(n2)

P2. Si f Î O(g) entonces O(f) Í O(g).

¿Cómo es la relación para los W?

P3. Dadas f y g de N en R+, se cumple:

i) O(f) = O(g) Û f Î O(g) y g Î O(f)

ii) O(f) Í O(g) Û f Î O(g)

¿La relación de orden entre O(..) es completa? Dadas f y g, ¿se cumple O(f)ÍO(g) ó O(g)ÍO(f)?

P4. Dadas f y g, de N en R+, O(f+g) = O(max(f, g)).

W(f+g) = W(max(f+g))

¿Y para los Q(f+g)?

¿Es cierto que O(f − g) = O(max(f, −g))?

P5. Dadas f y g de N en R+, se cumple:

i) limn¥® f(n) Î R+ Þ O(f)=O(g), W(f)=W(g), Q(f)=Q(g)

g(n)

ii) limn¥® f(n) = 0 Þ O(f) Í O(g), W(g) Í W(f)

g(n)

P5. Ej. ¿Qué relación hay entre O(log2 n) y O(log10 n)?

P6. Dadas f y g de N en R+, O(f)=O(g) Û Q(f)=Q(g) Û f Î Q(g) Û W(f)=W(g)

P7. Dadas f y g de N en R+, se cumple:

i) limn¥® f(n) Î R+ Þ O(f) = O(g)

g(n)

ii) limn¥® f(n) = 0 Þ O(f) Ì O(g)

g(n)

iii) limn¥® f(n) = +¥ Þ O(f) É O(g)

g(n)

Notación con varios parámetros:

En general, el tiempo y la memoria consumidos pueden depender de muchos parámetros.

f: Nm ® R+ (f: Nx...m..xN ® R+)

Ej. Memoria en una tabla hash. M(B,n, l, k) = kB+l+n+2kn

Orden de complejidad de f(n1, n2, ..., nm): O(f)

Dada una función f: Nm ® R+, llamamos orden de f al conjunto de todas las funciones de Nm en R+ acotadas superiormente por un múltiplo real positivo de f, para valores de (n1, ..., nm) suficientemente grandes.

O(f)= { t: Nm ® R+ / $ c Î R+, $ n1, n2, .., nm Î N, " k1 ³ n1 ,

k2 ³ n2 ,..," km ³ nm : t(k1, k2, ..., km) £ c" f(k1, k2, ..., km) }

De la misma forma, podemos extender los conceptos de W(f) y Q(f), para funciones con varios parámetros.

Las propiedades se siguen cumpliendo ® Demostrarlo.

Ejemplo. T(N) = T(N, a, b) = a" N + b

El tiempo depende del tamaño del problema N, y del tiempo de inicialización b y de ejecución de un paso a.

Podemos suponerlos constantes T(N), o variables T(N,a,b).

¿Qué relación hay entre los siguientes órdenes?

O(n+m), O(nm) O(n2), O(n+2m)

Notaciones condicionales:

En algunos casos interesa estudiar el tiempo sólo para ciertos tamaños de entrada.

Ejemplo. Algoritmo de búsqueda binaria: Si N es potencia de 2 el estudio se simplifica.

Orden condicionado de f(n): O(f | P)

Dada una función f: N ® R+, y P: N ® B, llamamos orden de f según P (o condicionado a P) al conjunto:

O(f | P)= { t: N ® R+ / $ c Î R+, $ n0 Î N, " n ³ n0:

P(n) Þ t(n) £ c" f(n) }

De igual forma, tenemos W(f | P) y Q(f | P).

O(f) = O(f | true). Para cualquier f y g, f Î O(g | false).

¿O(f) « O(f | P)?

Ordenes De Complejidad

Uso de los órdenes de complejidad:

Dado un tiempo t(n), encontrar la función f más simple tal que t Î O(f), y que más se aproxime asintóticamente.

Ejemplo. t(n) = 2n2/5 + 3p/2; t(n) Î O(n2).

Relación de orden entre O(..) = Relación de inclusión entre conjuntos.

−O(f) £ O(g) Û O(f) Í O(g) Û Para toda t Î O(f), t Î O(g)

Se cumple que:

O(c) = O(d), siendo c y d constantes positivas.

O(c) Ì O(n)

O(cn + b) = O(dn + e)

O(p) = O(q), si p y q son polinomios del mismo grado.

O(p) Ì O(q), si p es un polinomio de menor grado que q.

Orden inferior u omega de f(n): W(f):

Dada una función f: N ® R+, llamamos omega de f al conjunto de todas las funciones de N en R+ acotadas inferiormente por un múltiplo real positivo de f, para valores de n suficientemente grandes.

W(f)= { t: N ® R+ / $ c Î R+, $ n0 Î N, " n ³ n0: t(n) ³ c" f(n) }

La notación omega se usa para establecer cotas inferiores del tiempo de ejecución.

Relación de orden: igual que antes.

Orden exacto de f(n): Q(f):

Dada una función f: N ® R+, llamamos orden exacto de f al conjunto de todas las funciones de N en R+ que crecen igual que f, asintóticamente y salvo constantes.

Q(f) = O(f) Ç W(f) =

= { t: N ® R+ / $ c, d Î R+, $ n0 Î N, " n ³ n0: c" f(n) ³ t(n) ³ d" f(n) }

Notación o pequeña de f(n): o(f):

Dada una función f: N ® R+, llamamos o pequeña de f al conjunto de todas las funciones de N en R+ que crecen igual que f asintóticamente:

o(f)= { t: N ® R+ / lim t(n)/f(n) = 1}n¥®

Esta notación conserva las constantes multiplicativas para el término de mayor orden.

Ejemplo. t(n) = amnm + am−1nm−1 + ... +a1n + a0

t(n) Î o(amnm) ¹ o(nm)

¿o(amnm) Í O(amnm)? ¿o(t) Í O(t)?

Costa de complejidad con frecuencia

Algunas relaciones entre órdenes frecuentes:

O(1) Ì O(log n) Ì O(n) Ì O(n" log n) Ì O(n" (log n)2) Ì O(n1.001...) Ì O(n2) Ì O(n3) Ì ... Ì O(2n) Ì O(n!) Ì O(nn)

¿Qué pasa con las omegas? ¿Y con los órdenes exactos?

El orden de un polinomio anxn+...+a1x+a0 es O(xn).

n n n

å1 = n Î O(n); åi = n(n+1)/2 Î O(n2); åim Î O(nm+1)

i=1 i=1 i=1

Si hacemos una operación para n, otra para n/2, n/4, ..., aparecerá un orden logarítmico O(log2 n). Los logaritmos son del mismo orden, independientemente de la base.

5. Técnica de diseño de algoritmos

Diseño de Algoritmos:

Empezando en el nodo 1.

Solución: (1, 4, 5, 3, 2)

Coste: 30+15+25+10+45=125

Empezando en el nodo 3.

Solución: (5, 4, 3, 2, 1)

Coste: 15+20+10+45+50=140

Heurística voraz 2

− Una solución será un conjunto de aristas (a1, a2, ..., an−1) que formen un ciclo hamiltoniano, sin importar el orden.

Empezar con un grafo sin aristas.

Selección: seleccionar la arista candidata de menor coste.

Factible: una arista se puede añadir a la solución actual si no se forma un ciclo (excepto para la última arista añadida) y si los nodos unidos no tienen grado mayor que 2.

Ejemplo.

Solución: ((2, 3), (4, 5), (3, 4), (1, 2), (1, 5))

Coste = 10+15+20+45+50 = 140

Conclusiones:

Ninguno de los dos algoritmos garantiza una solución óptima. Sin embargo, normalmente ambos dan soluciones buenas, próximas a la óptima.

Posibles mejoras: buscar heurísticas mejores; repetir la heurística 1 con varios orígenes; ó bien, a partir de la solución del algoritmo intentar hacer modificaciones locales para mejorar esa solución.

Algoritmos De Aproximación

Dado un problema NP completo, es probable que no sepamos resolverlo de manera precisa y completa utilizando un algoritmo polimico en tiempo. Para este tipo de problemas, los algoritmos que no conducen a una solución óptima se llaman algoritmos de aproximación. Sin embargo, resulta parcialmente interesante que estos garanticen una cota en el margen de imprecisión.

A continuación se ilustra este tipo de tratamiento de problemas al problema de recubrimiento de un grafico:

Dado un grafo G=(V,A), se trata de encontrar un conjunto con el menor numero de vértices tal que toda arista sea incidente por lo menos de un vértice de V.

Este problema se puede resolver a través de otro aproximado, como es calcular el ajuste maximizal del grafo G. Se trata de calcular un subconjunto A' de aristas tal que dos aristas cualquiera de A' no tengan ningún vértice común y toda arista de A−A' comparta algún vértice común con una arista de A'. Este nuevo problema garantiza conseguir un recubrimiento que contiene no más de dos vértices del recubrimiento mínimo. El procedimiento para construir un ajuste maximizal de un grafo G consistiría en ir tomando aristas de G, de una en una y en cualquier orden e ir eliminando las incidentes al conjunto que se esta construyendo hasta recubrir todo en grafo.

Para poder aplicar el nuevo problema aproximado, seria necesario demostrar que el conjunto de todos los vértices inciden a las aristas de un ajuste maximal M para un grafo G es un recubrimiento con no mas de dos

veces el numero de veces el recubrimiento de tamaño mínimo. Esto es evidente, ya que por la definición de ajuste maximal, los vértices incidentes a las aristas de M son un recubrimiento de G. también por la propia definición, ningún vértice perteneciente a M puede recubrir a mas de una arista en M. En consecuencia, por lo menos la mitad de los vértices de M deben pertenecer a un recubrimiento.

6. Algoritmos de búsqueda y ordenación

En muchas situaciones de programación es necesario repetir ciertos procedimientos hasta alcanzar un punto en que no se puede o no se desea continuar. Esta repetición de tareas puede llevarse a cabo básicamente de dos maneras diferentes: la iteración y la recursión. Como se supone que en cada repetición se procesa un estado diferente de cosas −sin lo cual el proceso tendería al infinito−, ambos métodos presentan también alguna forma de control de fin de tarea.

La idea básica en un algoritmo iterativo es que la repetición de la tarea se controla desde afuera. Se ejecuta un conjunto de acciones en forma completa, se verifica la condición de salida y si es necesario se vuelve a ejecutar el conjunto de acciones en forma completa. El orden en que se ejecuta y evalúa determina que el algoritmo sea de evaluación previa (primero se evalúa la condición de salida y luego se ejecutan las acciones) o de evaluación posterior (primero se ejecutan las acciones y luego se evalúa el resultado). En ambos casos, sin embargo, el control de las repeticiones es exterior al grupo principal de acciones.

En un algoritmo recursivo, en cambio, la tarea se controla desde adentro. Se comienza a ejecutar un conjunto de acciones, pero antes de finalizar se evalúa si se ha llegado a la condición de salida; si no es así, se continúa ordenando una nueva ejecución del mismo conjunto de acciones. Finalmente se concluye con la tarea iniciada. Dicho en otros términos, el procedimiento se llama repetidas veces a sí mismo, y el control de esas llamadas forma parte del grupo principal de acciones.

Por otra parte, si bien hay problemas que se resuelven más directamente en forma iterativa y otros que son más naturalmente recursivos, ambas técnicas son compatibles e intercambiables, por lo que todo algoritmo recursivo puede transformarse en iterativo y viceversa.

Algoritmos De Búsqueda

Cuando se trata de buscar un valor en un arreglo ordenado de datos, el algoritmo de búsqueda binaria es el más frecuentemente utilizado. La idea central de este algoritmo es comparar el elemento ubicado en el lugar central del arreglo con el valor buscado. Si el elemento central es igual al valor buscado la búsqueda finaliza con éxito. Si no es así, puede ocurrir o bien que el elemento central sea mayor que el buscado −en cuyo caso el elemento coincidente debe estar en la mitad inferior del arreglo− o bien que sea menor −y el elemento coincidente se encuentra en la mitad superior. En ambos casos se prosigue la búsqueda en la mitad que corresponde, si es que quedan elementos en esa dirección, o bien se finaliza la búsqueda sin éxito, en caso contrario. Existe naturalmente una solución recursiva, ya que si el valor buscado no es hallado en la posición del centro se repite el mismo procedimiento con una de las mitades del arreglo, y así hasta que se encuentre el valor o no queden más "mitades". Compárese esto con el problema de las bolillas dentro de las cajas, en el cual la "bolilla blanca" sería el valor buscado y la "caja interior" sería la mitad que se debe seguir examinando.

En ocasiones es necesario determinar no sólo si el valor buscado se halla en el arreglo, sino además saber en qué posición está (o debería estar, si es que no existe). Por ejemplo, si se desea insertar un nuevo valor en un arreglo ordenado, una solución eficaz es "buscar" el valor en el arreglo (aunque se sepa de antemano que no se encuentra) para determinar la posición correcta en la que debe ser insertado. En esos casos se debe informar por algún medio (generalmente un puntero pasado como parámetro o una variable global) cuál es la posición lógica del elemento buscado dentro del arreglo.

Ejemplo de un Algoritmo de Búsqueda

A modo de ejemplo se presenta una versión de la función

int busbin (int *vec, unsigned tam, int val, unsigned *ord);

Ésta realiza una búsqueda binaria del elemento val en el vector de enteros apuntado por vec de tam elementos y deja en la memoria apuntada por ord la posición lógica que tiene (o debería tener) el elemento buscado. Retorna 1 si el elemento es hallado o 0 en caso contrario. Para calcular el elemento mitad del vector se desplaza tam un bit a la derecha, lo que equivale a dividirlo en dos, pero resulta mucho más rápido para el procesador que la operación de división.

int busbin (int *vec, unsigned tam, int val, unsigned *ord)

{if (!(vec && tam && ord)) return 0;

unsigned mitad = tam >> 1; // Divide tam en 2 desplazando un bit a la// derecha. Es más rápido que tam / 2.

if (vec [mitad] == valor) { *ord += mitad; return 1; }

if (vec [mitad] < valor)

{ mitad++; *ord += mitad; vec += mitad; tam −= mitad; }

else tam = mitad;

return tam? busbin (vec, tam, va, ord): 0;}

Algoritmos De Ordenacion

Se presentan aquí dos métodos muy utilizados para obtener el ordenamiento de un conjunto de datos: el algoritmo de ordenamiento por inserción y el algoritmo conocido como quick sort (ordenamiento rápido). Estos dos algoritmos son ilustrativos, además, porque el algoritmo de ordenamiento por inserción es esencialmente iterativo, mientras que el algoritmo de quick sort es esencialmente recursivo.

A continuación se comentan las características de ambos algoritmos para ordenar en forma ascendente los valores dentro de un vector en memoria. Siguiendo este ejemplo, para obtener ordenamientos descendentes basta cambiar el sentido de las comparaciones por desigualdad, y para otro tipo de soporte de datos (archivos en disco, por ejemplo) los cambios se referirán principalmente al modo de leer y escribir los datos.

Ordenación por Inserción:

La idea central de este algoritmo es recorrer secuencialmente y hacia delante el conjunto de datos comparando cada elemento con el anterior. Si el elemento actual es igual o mayor que el anterior entonces ese par de datos se encuentra en la secuencia correcta, por lo que no hay que modificar nada. Si, en cambio, el actual es menor que el anterior, significa que el actual está fuera de secuencia, por lo que debe ser insertado en el lugar correcto entre los valores que han quedado atrás. Una vez resuelta esta cuestión, se repite el proceso con el elemento siguiente del conjunto hasta que no haya más elementos.

Dos características surgen directamente de esta descripción:

1) Si hay que comparar cada valor con el anterior, entonces se debe comenzar el proceso por el segundo

elemento, ya que es el primero en tener uno anterior.

2) Si se van reinsertando correctamente "hacia atrás" los valores a medida que se avanza (o si se avanza sin reinsertar porque no es necesario), entonces los elementos anteriores al actual ya están ordenados.

La tarea de reinsertar un elemento "hacia atrás" cuando se descubre que está fuera de secuencia es accesoria y puede ser realizada por una función auxiliar. Esta función auxiliar se puede implementar de varias maneras, pero todas ellas deben partir de la certeza de que ese subconjunto de elementos ya está ordenado. A continuación se presentan una implementación de la función principal

int *ordins (int *vec, unsigned tam);

y dos implementaciones alternativas de la función auxiliar

void insertar (int *ult, unsigned tam);

La función ordins () ordena por inserción el vector de enteros apuntado por vec de tamaño tam y retorna vec, mientras que ambas versiones de insertar (), ubican el valor contenido en el elemento apuntado por ult entre los tam elementos ordenados que han quedado atrás. Ambas funciones auxiliares son estáticas −esto es, de uso "privado" dentro del módulo donde están insertas− ya que la tarea que realizan es subsidiaria y dependiente de la función ordins ().

int *ordenar (int *vec, unsigned tam)

{int *ant = vec, *act = vec + 1;unsigned i;for (i = 1; i < tam; i++, ant++, act++)

if (*act < *ant) insertar (act, i);return vec;}

La primera versión de insertar () es típicamente iterativa y efectúa una especie de rotación de los valores de los elementos. Primeramente resguarda en la variable local auxiliar aux el valor del ítem a reinsertar −que se encuentra en *ult− y a continuación procede a mover una posición hacia delante cada uno de los elementos anteriores a aux retrocediendo en el vector hasta que no hay más elementos o encuentra uno que no es mayor que aux. Finalmente reinserta el valor de aux en el lugar donde se encontraba el último elemento removido.

static void insertar (int *ult, unsigned tam)

{int *ant = ult − 1; // ant apunta siempre al ítem anterior a ult.

int aux = *ult; // aux contiene el valor a reinsertar.

do *(ult−−) = *(ant−−); // Evaluación posterior porque ya se sabe que el

while (−−tam && aux < *ant); // primer ítem anterior a ult es mayor que *ult.

*ult = aux; // Restituye el valor de aux en la última} // posición vacante.

La segunda versión de insertar () recurre a su vez a la función auxiliar de búsqueda binaria busbin (), con el fin de determinar la posición que debe ocupar el elemento *ult entre los anteriores a ult. Para ello inicializa las variables globales estáticas valor y orden, que son utilizadas por busbin () para conocer el elemento buscado e informar su posición. Obsérvese que para que esta versión de insertar () pueda acceder a otras funciones y variables globales estáticas, debe estar incluida a su vez en el mismo módulo de código que ellas. Luego de obtener a través de busbin () el lugar que debe ocupar el nuevo ítem, calcula la cantidad de elementos a

desplazar y −como la primera versión− mueve una posición hacia delante cada uno de los elementos anteriores a ult retrocediendo en el vector hasta llegar a la posición correcta. Ya no necesita comparar y contar puesto que, al conocer la cantidad de elementos a desplazar, sólo necesita contar. Finalmente reinserta el valor de aux en el lugar donde estaba el último elemento removido.

```
static void insertar (int *ult, unsigned tam)

{unsigned _ord = 0; // _ord contendrá la posición correcta de *ult.

int *ant = ult − 1; // ant apunta al ítem anterior a ult.

valor = *ult; // Inicializa las variables globales estáticas

orden = &_ord; // que utilizará busbin ().

if (−−tam) // Si hay más de un elemento llama a busbin ()

busbin (ant − tam, tam); // con la dirección inicial del vector.

tam −= _ord; // Descuenta de tam la posición del nuevo ítem.

do *(ult−−) = *(ant−−); // Desplaza la cantidad necesaria de elementos

while (tam−−); // hacia delante y restituye el valor de valor

*ult = valor; // en la última posición vacante.}
```

La ventaja de la primera versión es que es autónoma, ya que no depende de otras funciones ni variables globales estáticas, por lo que puede ser incluida en cualquier módulo de código. Presenta sin embargo el inconveniente de que tiene que comparar una y otra vez cada elemento del arreglo para encontrar su posición correcta. En vectores de gran tamaño esto puede resentir en forma notable la eficiencia. La segunda versión, en cambio, soluciona el problema de las comparaciones sucesivas utilizando un algoritmo de búsqueda más eficiente, aunque esto la obliga a convivir en el mismo módulo con otras funciones y variables globales estáticas de las cuales depende.

Algoritmo de Quick Sort:

La idea central de este algoritmo es la siguiente: si se toma un conjunto de elementos desordenados y se ubica uno cualquiera de ellos −llamado pivote− en una posición tal que todos los que estén antes sean menores o iguales y todos los que estén después sean mayores, entonces esa posición particular sería la correcta para ese elemento si el conjunto estuviera ordenado. Asimismo se verifica que el conjunto ha quedado dividido en dos partes: en la primera están todos los elementos menores o iguales al pivote y en la segunda todos los mayores. Esto permite aplicar recursivamente el mismo procedimiento en ambas partes del conjunto hasta que éste quede completamente ordenado. La tarea de dividir el conjunto de datos en dos grupos en torno al pivote es accesoria y puede ser realizada por una función auxiliar. Esta función auxiliar se puede implementar de varias maneras, pero todas ellas deben informar de algún modo en qué posición ha quedado ubicado el pivote.

Al igual que busbin (), la función qsort () es recursiva, y cada llamada provocará que la función se llame "internamente" a sí misma varias veces. Los valores de los parámetros vec y tam seguramente variarán entre una llamada y otra, ya que son los que definen la parte del arreglo que se ordenará cada vez. Sin embargo, tanto los controles de que los parámetros sean legales como el retorno de la función con la dirección original del vector deben hacerse sólo una vez.

Para evitar tener que hacer estos controles redundantes en cada llamada recursiva, la tarea se divide entre dos funciones. La primera está declarada públicamente como

int *qsort (int *vec, unsigned tam);

No es recursiva y sólo realiza las tareas que deben ser ejecutadas una sola vez: verifica que vec y tam sean legales y llama a la segunda función, que es la que completa la tarea. Si vec o tam son ilegales la función retorna vec sin hacer nada más. La segunda función, definida como

void _qsort (int *vec, unsigned tam);

Es estática (por lo tanto inaccesible desde fuera del módulo), y es la que realmente lleva a cabo la tarea recursiva de ordenamiento. Recibe los parámetros vec y tam, que varían entre una llamada y otra, pero que tienen siempre valores legales.

A continuación se presentan dos pequeños módulos de librería con una versión optimizada de la función: el módulo qsort.h contiene simplemente la declaración, mientras que el módulo qsort.cpp incluye la implementación de la función principal

int *qsort (int *vec, unsigned tam);

y una implementación de las dos funciones auxiliares

int dividir (int *vec, unsigned tam);

void _qsort (int *vec, unsigned tam);

La función qsort () verifica la validez de los parámetros vec y tam y llama a _qsort () antes de retornar vec. La función _qsort () ordena por el algoritmo de quick sort el vector de enteros apuntado por vec de tamaño tam, llamando a su vez a la función dividir (). Ésta divide los tam elementos del vector apuntado por vec en dos grupos alrededor de un pivote y retorna la cantidad de elementos del primer grupo −que incluye al pivote. Ambas funciones auxiliares son estáticas −esto es, de uso "privado" dentro del módulo donde están insertas− ya que la tarea que realizan es subsidiaria y dependiente de la función qsort ().

7. Verificación y derivación de programas

Conceptos Basicos

La definición del significado de un elemento del lenguaje se puede realizar de distintas formas, cada una de las cuales define una semántica diferente del lenguaje. En esta lección se van a introducir los conceptos más importantes de algunas de estas formas semánticas, y se van a tratar más extensamente los conceptos de corrección, verificación y prueba, ya mencionados en la lección.

Semántica:

Ahondando en la situación de la introducción anterior sobre la definición del lenguaje pascal, se puede conjeturar que la manera informar de definir el significado de las construcciones del lenguaje es insatisfactoria. El motivo es la falta de precisión con la que se define el significado de dichos conceptos, lo cual deja abierta como posibilidad que dicho significado dependa de la maquina donde se utilice el lenguaje. Otro problema importante es la ambigüedad del lenguaje natural, que permite que distintos implementadotes o programadores entiendan de modo distintos una definición. Acerca de este tema, el lenguaje pascal vuelve a servirnos de ejemplos. Hasta hace no mucho tiempo existía una gran variedad de versiones del mismo, cada

una realizada específicamente.

Dependiendo del objetivo prioritario que se presenta cubrir al dar el significado de un lenguaje podemos encontrar diversas aproximaciones. Entre todas ellas, las más frecuentemente utilizadas son la sematica operacional, la sematica declarativa y la sematica axiomatica. Veamos a continuación una peque son la sematica operacional, la sematica declarativa y la sematica axiomatica. Veamos a continuación una pequeña introducción a cada una de estas aproximaciones.

Semántica operacional:

La semántica operacional define el significado de un lenguaje de programación en términos de los cambios de estado que producen las instrucciones primitivas del lenguaje. Estos cambios no se reflejan directamente en la maquina real, sino en una maquina (virtual) abstracta asociada que sirve como instrumento de conexión con aquella. Expresado de otra forma, podemos decir que la semántica operacional define el significado de un programa en términos del efecto producido por la ejecución paso a paso del mismo, de tal modo que la especificación de las instrucciones del lenguaje mediante instrucciones primitivas de la maquina abstracta es, precisamente, la definición semántica del mismo.

A pesar de la aparente simplicidad de este formalismo, este tipo de semántica no describe con igual claridad todo tipo de lenguaje de programación. El motivo es que el mecanismo que emplean los distintos lenguajes de programación para realizar un cómputo no siempre puede expresarse de una manera clara, comprensible y concisa.

Semántica denotacional:

La semántica denotacional define unas aplicaciones (funciones) de valoración semántica que asignan a cada construcción denotada tal objeto matemático que modela su significado.

Se dice que la construcción denota tal objeto o que este objeto es la denotación de dicha construcción. En otras palabras, la semántica denotacional indica que función matemática se obtiene a la salida ante unas entradas del programa, sin preocuparse de la ejecución paso a paso del programa. Existe una variante de esta semántica que es la semántica algebraica, en la que se utiliza conceptos algebraicos a la hora de modelar el significado de las construcciones.

El primer paso a realizar en la definición de la semántica denotacional de un determinado lenguaje es el establecimiento de un dominio semantico al que pertenecerán los resultados obtenidos de la evaluación de las construcciones del lenguaje. Esta evaluación es proporcionada por un conjunto de funciones de significado cuyo dominio esta constituido por el conjunto de construcciones del lenguaje y cuyo rango (o imajen) viene dado por el dominio semantico.

Este tipo de semántica dotan de significado a los elementos del lenguaje de una manera mas formal y abstracta, pero sin embargo también necesitan un mayor conocimiento de conceptos matemáticos, que no tienen por que ser básicos ni elementales.

Recursividad:

Una función recursiva es una función que se define en términos de si misma., es decir, en su cuerpo aparece alguna aplicación suya. La recursividad es un mecanismo que se usa cuando hay que repetir cierto tratamiento o cálculo pero el número de repeticiones es variable. De hecho ya se han visto definiciones recursivas, el tipo suma permite definir tipos de datos recursivos, como el tipo de lista. La definición del tipo lista permite repetir la adición de elementos a una lista. el numero de adicciones es variable y determina el numero final de elementos de lista.

Iteración:

Una instrucción iterativa engloba una secuencia de instrucciones que se escribe una sola vez, pero permite que se ejecute varias veces. Una instrucción iterativa también se llama Bucle. Las instrucciones englobadas suelen denominarse cuerpo del bucle; cada ejecución del cuerpo de un bucle se llama iteración del mismo. Hay varios formatos de bucles que vemos a continuación.

Una instrucción LOOP tiene el siguiente formato:

LOOP

<Instrucción>

Su efecto es ejecutar eternamente la instrucción englobada. Sin embargo, puede terminarse la ejecución del bucle si se ejecuta una instrucción especial, formada por la palabra clave EXIT. Obsérvese que la instrucción EXIT debe ir incluida dentro de una instrucción condicional; en caso contrario, el bucle LOOP terminaría su ejecución al llegar a ella, realizando una iteración como mucho. Pero esto no tiene mucho sentido, porque si se hubiera querido realizar una sola ejecución de dichas instrucciones, no se hubieran incluido en una instrucción iterativa.

La instrucción EXIT no puede usarse en ninguna de las dos clases de bucles que se exponen a continuación.

Una instrucción WHILE tiene el siguiente formato:

WHILE <condición>DO

<Instrucción>

La ejecución de una instrucción WHILE comienza evaluando la condición booleana. Si es cierta, se ejecuta la secuencia de instrucciones y se comienza de nuevo. Si la condición es falsa, la ejecución del WHILE termina y el algoritmo continúa por la instrucción siguiente al bucle.

La instrucción FOR tiene el siguiente formato:

FOR <variable> IN <subrango>BY<expresión entera>DO

<Instrucción>

La instrucción FOR usa una variable, llamada variable de índice, que toma distintos valores en las sucesivas ejecuciones del cuerpo. La variable de índice debe ser entera, enumerada o subrango. Los valores que toma la variable de índice se indica mediante un subrango del mismo tipo y un incremento. El subrango se expresa de la manera usual: un valor inicial y un valor final separados por dos puntos. El incremento viene dado por una expresión entera; puede suprimirse, en cuyo caso se supone que es igual a uno.

Al comenzar la ejecución del bucle, la variable índice toma el valor inicial del subrango y se evalúa la expresión entera para determinar el incremento.

Veamos el resto del comportamiento del bucle cuando el tipo de la variable índice es un subrango entero. Tras cada iteración, el valor del índice se actualiza, sumándose a su valor el incremento. Si el incremento es positivo, solo se realiza la siguiente iteración si el valor de la variable índice es menor o igual que el valor final del subrango; por el contrario, si el incremento es negativo, solo se itera de nuevo cuando su valor es mayor o igual que el valor del final del subrango.

Si la variable es enumerada o de un subrango enumerado, el comportamiento del bucle es parecido. Un valor es menor que otro si tiene un ordinal menor. El incremento de un valor enumerado produce otro valor cuyo ordinal es mayor que el inicial en la cantidad expresada por el incremento.

Semantica Axiomatica

La semántica axiomatica asociada a cada construcción del lenguaje un axioma lógico que relaciona el estado del computo (valores que tienen las variables utilizadas) antes y después de ejecutar esta construcción. Existen distintas semánticas, axiomatica, de la que la mas conocida es la introducida por el sistema formal de hoare y que es utilizada para definir el significado de lenguaje imperativos (por ejemplo, pascal).

El método axiomatico expresa la semántica de un lenguaje de programación asociado al lenguaje una teoría matemática o sistema formal que permita demostrar propiedades de los programas escritos en ese lenguaje. Esta aproximación formal contrasta con el método denotacional mostrando anteriormente, que asocia a cada construcción del lenguaje una denotación (significado) en un intento de encontrar un modelo matemático (colección abstracta de objetos matemáticos) del lenguaje.

Un sistema formal puede entenderse como un metalenguaje que pose su propia sintaxis y semántica. Desde el punto de vista sintáctico, es necesario definir una gramática en la que se reflejen las contrucciones validas del sistema formal, normalmente se suele emplear la sintaxis sde la lógica de predicado de primer orden ampliada com. expresiones aritméticas y de teoría de conjuntos. Una vez fijada la sintaxis a emplear, un sistema formal se define mediante un conjunto de axiomas (propiedades ciertas escritas en forma de sentencias logicas) que expresen las propiedades de las construcciones básicas y un conjuntos de reglas de inferencias (forma de deducir una propiedades de otras) que permitan deducir las propiedades de construcciones mas complejas.

Las definiciones de sistemas formales para lenguajes funcionales suele hacerse fundamentalmente para demostrar propiedades que tiene que ver con el tipo de una expresión y de las subexpresiones que la componen.

Diseño De Algoritmos Recursivos

Las definiciones recursivas pueden resultar sorprendentes a primera vista.Incluso puede dar lugar a funciones que nunca terminen (inútiles como algoritmos), Sin embargo, usadas juiciosamente son un instrumento potentisimo de diseño de algoritmos.

En primer lugar, hay que identificar el proceso repetitivo por realizar. En segundo lugar, hay que considerar que una función recursiva solo es útil si su evaluación termina.

Clasificación

Según el número de llamadas recursivas realizadas, se distinguen tres clases:

Recursividad lineal: El cuerpo de la función contiene una llamada recursiva. Son las funciones recursivas. Son las funciones recursivas más sencillas. Todas las funciones especificadas en este capitulo son recursivas lineales.

Un caso importante de recursividad lineal aparece cuando la expresión más externa del caso recursivo de la función es la misma llamada recursiva. Esta clase de recursividad se llama recursividad de cola.

Recursividad no lineal.

El cuerpo de la función contiene varias llamadas recursivas. Lo mas frecuente que contenga dos llamadas

recursivas, en cuyo caso se habla de recursividad binaria.

Recursividad mutua: Es un caso curioso de recurcion, una función no contienen ninguna llamada recursiva, pero durante la evaluación de su aplicación surgen llamadas recursivas. Auque a primera vista parezca imposible obtenérsete comportamiento, la recursividad puede conseguirse indirectamente si hay dos funciones que se llaman entre si.

Otra clasificación basada en el formato de los parámetros de las llamadas recursivas.

Recursividad estructural: Las sucesivas llamadas recursivas sobre un dato se hacen siguiendo la estructura recursiva del mismo. En el caso de los elementos enteros, una recurcion estructurar significa que el valor se vaya decrementando en uno.

Recurcion bien fundada: Aunque hablando con propiedad, cualquier recursividad esta bien fundada, suele utilizarse este término para aquellas recursividades que no son estructurales. Un ejemplo de función recursiva bien es invertir Entero, donde el parámetro entero se divide entre diez.

La primera clasificación dada es útil para decidir si es conveniente usar una definición recursiva o iterativa en los algoritmos imperactivo.

La segunda clasificación es útil para decidir el principio de inducción necesaria para necesaria para verificar el algoritmo.

Diseño De Algoritmos Iterativos

Cada clase de bucle resulta útil en situaciones diferentes. El bucle FOR se utiliza cuando se conoce el número de iteraciones que se quiere realizar. El bucle WHILE es usado en las otras situaciones en que no se conoce el número de iteraciones. El bucle LOOP es útil cuando puede haber varias condiciones de salida del bucle, situadas en diferentes partes del cuerpo, o cuando la condición de salida no esta al comienzo del bucle.

Sin embargo, conocer la instrucción iterativa que mas conviene para un algoritmo es solo el comienzo de la construcción de la instrucción. Es conveniente tener algunas guías para el diseño de algoritmos iterativos.

El diseño de un algoritmo iterativo se hace sobre una base distinta de la de uno recursivo. En los algoritmos imperativos se tiene disponible toda la información del problema en las variables. Por esta razón, es útil partir los datos del problema en dos:

- La parte que representa el problema aun no resuelto.
- La parte que representa el problema ya resuelto; es decir, el resultado ya calculado.

Una vez que se han distinguido estas dos partes, el bucle se construye a partir de tres decisiones hechas sobre dicha partición:

a) Fijar la situación inicial. El problema a resolver se encuentra en los parámetros de entrada o de entrada/salida; si estos datos quieren modificarse durante el algoritmo, deben copiarse en variables auxiliares. también hay que dar un valor inicial a las variables que representan la solución vacía.

b) Formar el cuerpo del bucle. Para ello puede usarse un razonamiento parecido al recursivo. Se supone, por generalidad, que la iteración se encuentra en un estado intermedio. Debe entonces determinarse que hacer en una iteración para avanzar en la resolución del problema.

c) Determinar la condición de terminación. Corresponde a la situación en que se ha hallado la solución

completa. Esta parte esta totalmente relacionada con la elección de la instrucción iterativa. Según la forma de la iteración, se elige un bucle FOR, WHILE o LOOP, como se describió antes.

Normalmente, la terminación del bucle puede especificarse de forma sencilla, incluso en casos relativamente complicados. Por ejemplo, cuando se tiene un bucle WHILE con una condición compuesta es posible que pueda usarse la técnica del centinela para obtener una especificación más sencilla.

Algo con lo que hay que tener especial cuidado es la terminación de los bucles. Esta es fácil de determinar en un algoritmo recursivo porque basta con tomar una medida de los parámetros y comprobar que disminuye en cada llamada recursiva, de forma que se acerca al tamaño de los casos básicos. En una solución iterativa se razona igual, pero la tarea es algo mas complicada porque el control del proceso repetitivo no tiene parámetros, sino que se hace a partir de la información contenida en las variables.

8. Análisis Foda

Introducción:

El análisis FODA es una herramienta que permite conformar un cuadro de la situación actual de la empresa u organización, permitiendo de esta manera obtener un diagnóstico preciso que permita en función de ello tomar decisiones acordes con los objetivos y políticas formulados.

El término FODA es una sigla conformada por las primeras letras de las palabras Fortalezas, Oportunidades, Debilidades y Amenazas (en inglés SWOT: Strenghts, Weaknesses, Oportunities, Threats). De entre estas cuatro variables, tanto fortalezas como debilidades son internas de la organización, por lo que es posible actuar directamente sobre ellas. En cambio las oportunidades y las amenazas son externas, por lo que en general resulta muy difícil poder modificarlas.

ð Fortalezas: son las capacidades especiales con que cuenta la empresa, y por los que cuenta con una posición privilegiada frente a la competencia. Recursos que se controlan, capacidades y habilidades que se poseen, actividades que se desarrollan positivamente, etc.

ð Oportunidades: son aquellos factores que resultan positivos, favorables, explotables, que se deben descubrir en el entorno en el que actúa la empresa, y que permiten obtener ventajas competitivas.

ð Debilidades: son aquellos factores que provocan una posición desfavorable frente a la competencia. recursos de los que se carece, habilidades que no se poseen, actividades que no se desarrollan positivamente, etc.

ð Amenazas: son aquellas situaciones que provienen del entorno y que pueden llegar a atentar incluso contra la permanencia de la organización.

Análisis:

El Análisis FODA es un concepto muy simple y claro, pero detrás de su simpleza residen conceptos fundamentales de la Administración. Intentaré desguazar el FODA para exponer sus partes fundamentales.

Tenemos un objetivo: convertir los datos del universo (según lo percibimos) en información, procesada y lista para la toma de decisiones (estratégicas en este caso). En términos de sistemas, tenemos un conjunto inicial de datos (universo a analizar), un proceso (análisis FODA) y un producto, que es la información para la toma de decisiones (el informe FODA que resulta del análisis FODA).

Sostengo que casi cualquier persona puede hacer un análisis FODA. Digo casi porque esa persona tiene que tener la capacidad de distinguir en un sistema:

- Lo relevante de lo irrelevante
- Lo externo de lo interno
- Lo bueno de lo malo

Parece fácil, ¿verdad?

Pongámoslo en otras palabras: el FODA nos va a ayudar a analizar nuestra empresa siempre y cuando podamos responder tres preguntas: Lo que estoy analizando, ¿es relevante? ¿Está fuera o dentro de la empresa? ¿Es bueno o malo para mi empresa?

Estas tres preguntas no son otra cosa que los tres subprocesos que se ven en el proceso central del dibujo de arriba. Pasemos a explicar:

La relevancia es el primer proceso y funciona como filtro: no todo merece ser elevado a componente del análisis estratégico. Es sentido común ya que en todos los órdenes de la vida es fundamental distinguir lo relevante de lo irrelevante. En FODA este filtro reduce nuestro universo de análisis disminuyendo nuestra necesidad de procesamiento (que no es poca cosa).

Ejemplos: dudosamente sea una ventaja comparativa el sistema de limpieza de baños de una petroquímica, o el color de los monitores, o si el papel que se usa es carta o A4. Parece tonto, pero es increíble la cantidad de veces que a los seres humanos nos cuesta distinguir lo principal de lo accesorio, ya sea en una discusión, una decisión o donde sea.

Claro que la relevancia de algo depende de dónde estemos parados, y este concepto de relatividad es importante. La higiene de los baños puede ser clave en un Hospital o un Hotel. El orden en el que se hacen los pasos al efectuar una compraventa no es tan importante como los pasos que toman los bomberos para apagar un incendio. La disciplina y la autoridad formal son dejadas de lado en muchas empresas de la Nueva Economía... pero a un ejército en batalla eso puede costarle la vida. Es por eso que quien hace un análisis FODA debe conocer el negocio (ni más ni menos que saber de lo que está hablando).

Filtrados los datos sólo nos queda clasificarlos. Aplicando el sentido común, podemos construir una matriz con dos dimensiones (dentro/fuera, bueno/malo):

	Positivas	Negativas
Exterior	Oportunidades	Amenazas
Interior	*Fortalezas*	Debilidades

Quien haya inventado el Análisis FODA eligió para cada intersección una palabra: así la intersección de bueno y exterior es una oportunidad, mientras que las cuestiones positivas del interior de nuestra empresa son una fortaleza, y así sucesivamente.

Distinguir entre el adentro y el afuera de la empresa a veces no es tan fácil como parece. Es fácil decir que desde el punto de vista de la Ferrari, M. Schumager es una fortaleza (interna), y que si M. Hakkinen se queda sin empleo en su escudería, será una Oportunidad (externa) para la Ferrari. Pero el control de un recurso escaso (petróleo) o un proveedor exclusivo están físicamente fuera de mi empresa... y sin embargo son Fortalezas. La clave está en adoptar una visión de sistemas y saber distinguir los límites del mismo. Para esto hay que tener en cuenta, no la disposición física de los factores, sino el control que yo tenga sobre ellos. Recordando una vieja definición de límite: lo que me afecta y controlo, es interno al sistema. Lo que me afecta pero está fuera de mi control, es ambiente (externo).

Sólo nos queda la dimensión positivo/negativo, que aparentemente no debería ofrecer dificultad, pero hay que tener cuidado. El competitivo ambiente de los negocios está lleno de maniobras, engaños, etc. En la Segunda

Guerra Mundial, el Eje estaba feliz de que el desembarco de los Aliados fuera en Calais, porque tenía muchas fortalezas en ese caso. Pero el día D fue en Normandía y por eso hoy el mundo es lo que es.

Las circunstancias pueden cambiar de un día para el otro también en el interior de la empresa: la Fortaleza de tener a ese joven y sagaz empleado puede convertirse en grave Debilidad si se marcha (y peor si se va con la competencia). Y la Debilidad de tener a un empleado próximo a jubilarse y a quien le cuesta adaptarse a las nuevas tecnologías puede revelarse como Fortaleza demasiado tarde... cuando se retira y nos damos cuenta de que dependíamos de él porque era el único que sabía dónde estaba todo y cómo se hacen las cosas.

La sagacidad del empresario debe convertir las Amenazas en Oportunidades y las Debilidades en Fortalezas. Ejemplos: Asociarnos con nuestra competencia de toda la vida para enfrentar a un enemigo más pesado; pasar a un empleado desestructurado y extrovertido de una tarea organizativa que hace mal, a la línea de fuego de atención al público. Las posibilidades son muchas.

Y esos son los tres pasos necesarios para analizar la situación actual de la organización mediante el Análisis FODA.

El Análisis FODA de este trabajo:

Fortaleza:

Los conceptos que se nos encomendaron investigar, están desarrollados de forma amplia y lógica.

Tiene un contenido y una presentación al nivel de los que significa ser estudiante universitario de un 7mo cuatrimestre de Ing. En Sistemas.

Continuidad y constancia de la estructura, calidad y cantidad de contenido que los demás trabajos entregados, a demás de mejoras considerables, lo cual nos garantiza obtener calificaciones similares o superiores a las ya obtenidazas de referidos trabajos.

Oportunidades:

Debido a las mejoras realizadas al mismo y una mejor coordinación y participación de los integrantes del grupo este se concluirá en menor tiempo y así podremos completar el procedimiento correcto de entrega de trabajos de esta materia.

Debilidades:

A causa de la dificultad para conseguir la información solicitada algunos de los conceptos relacionados con los problemas, tipos, etc. No se encuentran en nuestro trabajo.

Amenazas:

Que las debilites de nuestro trabajo influyan considerablemente en la evaluación del mismo provocando que las expectativas y el esfuerzo realizados por los que participamos en el mismo no lleguen a cumplirse.

9. Conclusión

Luego de realizar este trabajo hemos visto como los algoritmos son una de las herramientas más complejas y aplicables en el área de la informática y el mundo de los computadores.

Pudimos comprobar que mientras más potente, completo y eficiente es el computador o la aplicación que

corre sobre el mismo mas grande, complejo y exacto es el algoritmo que utiliza.

Las técnicas de desarrollo de algoritmos nos permiten encontrar la mejor solucion a los problemas que se nos presentan y deben ser solucionados por el computador, estas técnicas están orientadas para utilizarse en cada uno de los niveles de complejidad y variedad o alternativas para las cuales se aplican los algoritmos.

Un algoritmo es el conjunto de operaciones y procedimientos que deben seguirse para resolver un problema, es por ellos que debemos estudiarlos y conocerlos.

10. Bibliografía

Correa Uribe, Guillermo. Desarrollo de Algoritmos y sus aplicaciones, Editora MacGraw − Hill Inc. USA, III Edición. Abril/1992, Colombia. pp. 251.

Gálvez, Javier. Gonzáles, Juan. Algorítmica, Análisis y Diseño de Algoritmos, Editora RA−MA (Addison−Wesley Iberiamericana), II Edición. Septiembre/1993, USA. pp 502.

Matías, Cristian Manual de Informática Educativa, Editora Taller. 2da. Edición. Julio/1999. Sto. Dgo. R.D. pp 260.

Sean, James A. Análisis y Diseño de Sistemas de Información, Editora MacGraw − Hill Inc. USA, 2 ta. Edición. Diciembre/1998, México. pp 941.

World Wide Wed:

www.altavista.com

www.elrincondelvago.com

www.aulaclick.com