Carlos Armando Carlon Gerardo 17211498

Resolución de Problemas

El proceso de resolución de un problema con una computadora conduce a la escritura de un programa y a su ejecución en la misma. Aunque el proceso de diseñar programas es, esencialmente, un proceso creativo, se puede considerar una serie de fases o pasos comunes, que generalmente deben seguir todos los programadores. Las fases de resolución de un problema con computadora son:

• Análisis del problema.

• Diseño del algoritmo.

• Codificación.

• Compilación y ejecución.

• Verificación.

• Depuración.

• Mantenimiento.

• Documentación.

Las características más sobresalientes de la resolución de problemas son:

• Análisis. El problema se analiza teniendo presente la especificación de los requisitos dados por el cliente de la empresa o por la persona que encarga el programa.

• Diseño. Una vez analizado el problema, se diseña una solución que conducirá a un algoritmo que resuelva el problema.

• Codificación (implementación). La solución se escribe en la sintaxis del lenguaje de alto nivel y se obtiene un programa fuente que se compila a continuación.

• Ejecución, verificación y depuración. El programa se ejecuta, se comprueba rigurosamente y se eliminan todos los errores (denominados “bugs”, en inglés) que puedan aparecer.

• Mantenimiento. El programa se actualiza y modifica, cada vez que sea necesario, de modo que se cumplan todas las necesidades de cambio de sus usuarios.

• Documentación. Escritura de las diferentes fases del ciclo de vida del software, esencialmente el análisis, diseño y codificación, unidos a manuales de usuario y de referencia, así como normas para el mantenimiento.

Las dos primeras fases conducen a un diseño detallado escrito en forma de algoritmo. Durante la tercera fase (codificación) se implementa el algoritmo en un código escrito en un lenguaje de programación, reflejando las ideas desarrolladas en las fases de análisis y diseño.

Las fases de compilación y ejecución traducen y ejecutan el programa. En las fases de verificación y depuración el programador busca errores de las etapas anteriores y los elimina. Comprobará que mientras más tiempo se gaste en la fase de análisis y diseño, menos se gastará en la depuración

del programa. Por último, se debe realizar la documentación del programa. Antes de conocer las tareas a realizar en cada fase, se considera el concepto y significado de la palabra algoritmo. Un algoritmo es un método para resolver un problema mediante una serie de pasos precisos, definidos y finitos. Un algoritmo debe producir un resultado en un tiempo finito. Los métodos que utilizan algoritmos se denominan métodos algorítmicos, en oposición a los métodos que implican algún juicio o interpretación que se denominan métodos heurísticos. Los métodos algorítmicos se pueden implementar en computadoras; sin embargo, los procesos heurísticos no han sido convertidos fácilmente en las computadoras. En los últimos años las técnicas de inteligencia artificial han hecho posible la implementación del proceso heurístico en computadoras.

CONCEPTO Y CARACTERÍSTICAS DE ALGORITMOS

El programador de computadora es antes 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.

Los pasos para la resolución de un problema son:

1. Diseño del algoritmo, que describe la secuencia ordenada de pasos —sin ambigüedades— que conducen a la solución de un problema dado. (Análisis del problema y desarrollo del algoritmo.)

2. Expresar el algoritmo como un programa en un lenguaje de programación adecuado. (Fase de codificación.)

3. Ejecución y validación del programa por la computadora.

Para llegar a la realización de un programa es necesario el diseño previo de un 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 los 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. 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 sólo un medio para expresar un algoritmo y una computadora es sólo 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. 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:

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

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

• Un algoritmo debe ser finito.

Carlos Armando Carlon Gerardo 17211498

Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos. La definición de un algoritmo debe describir tres partes: Entrada, Proceso y Salida.

Diseño del algoritmo

La información proporcionada al algoritmo constituye su entrada y la información producida por el algoritmo constituye su salida. Los problemas complejos se pueden resolver más eficazmente con la computadora cuando se rompen en subproblemas que sean más fáciles de solucionar que el original. Es el método de divide y vencerás (divide and conquer), y que consiste en dividir un problema complejo en otros más simples. Así, el problema de encontrar la superficie y la longitud de un círculo se puede dividir en tres problemas más simples o subproblemas. La descomposición del problema original en subproblemas más simples y a continuación la división de estos subproblemas en otros más simples que pueden ser implementados para su solución en la computadora se denomina diseño descendente (top-down design). Normalmente, los pasos diseñados en el primer esbozo del algoritmo son incompletos e indicarán sólo unos pocos pasos (un máximo de doce aproximadamente). Tras esta primera descripción, éstos se amplían en una descripción más detallada con más pasos específicos. Este proceso se denomina refinamiento del algoritmo (stepwise refinement).

Para problemas complejos se necesitan con frecuencia diferentes niveles de refinamiento antes de que se pueda obtener un algoritmo claro, preciso y completo.

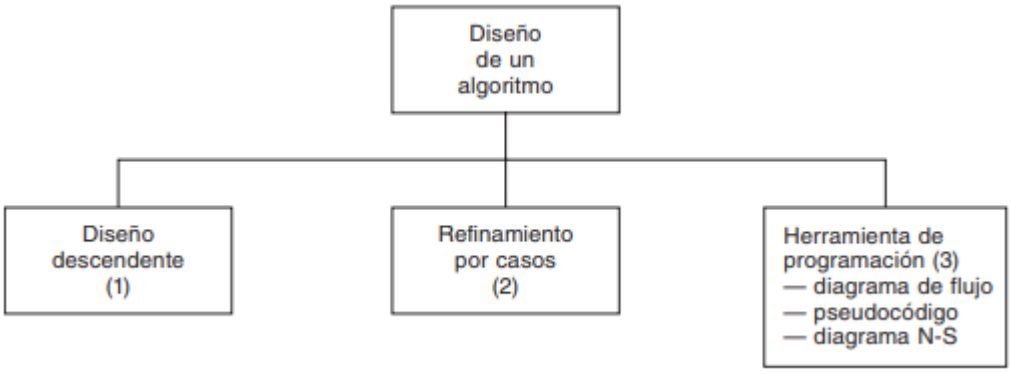
Las ventajas más importantes del diseño descendente son:

• El problema se comprende más fácilmente al dividirse en partes más simples denominadas módulos.

• Las modificaciones en los módulos son más fáciles.

• La comprobación del problema se puede verificar fácilmente.

Tras los pasos anteriores (diseño descendente y refinamiento por pasos) es preciso representar el algoritmo mediante una determinada herramienta de programación: diagrama de flujo, pseudocódigo o diagrama N-S.



La eficiencia de un algoritmo Principio de invarianza

Carlos Armando Carlon Gerardo 17211498

Dado un algoritmo y dos implementaciones suyas I1 e I2, que tardan T1(n) y T2(n) respectivamente, el Principio de invarianza afirma que existe una constante real c > 0 y un número natural n0 tales que para todo n ≥ n0 se verifica que:

T1(n) ≤ cT2(n)

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa. Esto significa que el tiempo para resolver un problema mediante un algoritmo depende de la naturaleza del algoritmo y no de la implementación del algoritmo. Decimos entonces que el tiempo de ejecución de un algoritmo es asintóticamente de del orden de T(n) si existen dos constantes reales c y n0 y una implementación I del algoritmo tales que el problema se resuelve en menos de cT(n), para toda n > n0. Cuando se quiere comparar la eficiencia temporal de dos algoritmos, tiene mayor influencia el tipo de función que la constante c.

n	Algoritmo 1: $T_1(n) = 10^6 n^2$		Algoritmo 2: $T_2(n) = 5 n^3$
2	10^6 x 4: 4,000,000	>	5 x 8: 40
200	10^6 x 40,000: $4x10^{10}$	>	5 x $8x10^6$: $4x10^7$
200,000	10^6 x $4x10^{10}$: $4x10^{16}$	=	5 x $8x10^{15}$: $4x10^{16}$
2,000,000	10^6 x $4x10^{12}$: $4x10^{18}$	<	5 x $8x10^{18}$: $4x10^{19}$

A partir de cierto valor de n, la función cúbica es siempre mayor a pesar de que la constante es mucho menor a ésta.

El tiempo de ejecución T(n) de un algoritmo

Para medir T(n) usamos el número de operaciones elementales. Una operación elemental puede ser:

- Operación aritmética.
- Asignación a una variable.
- Llamada a una función.
- Retorno de una función.
- Comparaciones lógicas (con salto).
- Acceso a una estructura (arreglo, matriz, lista ligada…).

Se le llama tiempo de ejecución, no al tiempo físico, sino al número de operaciones elementales que se llevan a cabo en el algoritmo.

Análisis de la complejidad algorítmica

Medidas asintóticas

Las cotas de complejidad, también llamadas medidas asintóticas sirven para clasificar funciones de tal forma que podamos compararlas. Las medidas asintóticas permiten analizar qué tan rápido crece el tiempo de ejecución de un algoritmo cuando crece el tamaño de los datos de entrada, sin

importar el lenguaje en el que esté implementado ni el tipo de máquina en la que se ejecute. Existen diversas notaciones asintóticas para medir la complejidad, las tres cotas de complejidad más comunes son: la notación O (o mayúscula), la notación Ω (omega mayúscula) y la notación θ (theta mayúscula) y todas se basan en el peor caso.

1 Cota superior asintótica: Notación O (o mayúscula)

O(g(n)) es el conjunto de todas las funciones fi para las cuales existen constantes enteras positivas k y n0 tales que para n ≥ n0 se cumple que: fi(n) ≤ kg(n) kg(n) es una "cota superior" de toda fi para n ≥ n0 Cuando la función T(n) está contenida en O(g(n)), entonces la función cg(n) es una cota superior del tiempo de ejecución del algoritmo para alguna c y para toda n ≥ n0. Lo que indica que dicho algoritmo nunca tardará más que: k g(n). Recordar que el tiempo de ejecución es el número de operaciones elementales que se llevan a cabo y que n es el tamaño de los datos de entrada.

2 Cota inferior asintótica: Notación Ω (omega mayúscula)

Ω(g(n)) es el conjunto de todas las funciones fi para las cuales existen constantes enteras positivas k y n0 tales que para n ≥ n0 se cumple que:

fi (n) ≥ kg(n)

kg(n) es una "cota inferior" de toda fi para n ≥ n0

En resumen, f(n) ≥ kg(n) implica que f(n) es Ω(g(n)), es decir, f(n) crece asintóticamente más rápido que g(n) cuando n →∞.

3 Orden exacto o cota ajustada asintótica: Notación θ (theta mayúscula)

θ(g(n)) es el conjunto de todas las funciones f i para las cuales existen constantes enteras positivas k1 ,k2 y n0 tales que para n ≥ n0 se cumple que:

k1g(n) ≤ f i (n) ≤ k2 g(n)

En resumen, k1g(n) ≤ f (n) ≤ k2 g(n) implica que f (n) es θ(g(n)), es decir, f (n) y g(n) crecen asintóticamente a la misma velocidad cuando n →∞.

Clasificando algoritmos con la notación O

La notación O sirve para identificar si un algoritmo tiene un orden de complejidad mayor o menor que otro, un algoritmo es más eficiente mientras menor sea su orden de complejidad.

Los diferentes tipos de complejidad

Existen diferentes tipos de complejidad, lo que se desea de un algoritmo es que su complejidad sea la menor posible. A continuación se presentan diferentes tipos de complejidad (las más comunes) ordenadas de menor a mayor.

- O(1) Complejidad constante.- Es la más deseada. Por ejemplo, es la complejidad que se presenta en secuencias de instrucciones sin repeticiones o ciclos.
- O(log n) Complejidad logarítmica.- Puede presentarse en algoritmos iterativos y recursivos.

- O(n) Complejidad lineal.- Es buena y bastante usual. Suele aparecer en un ciclo principal simple cuando la complejidad de las operaciones interiores es constante.
- O(n log n) Complejidad n·log·n.- Se considera buena. Aparece en algunos algoritmos de ordenamiento.
- O(n^2) Complejidad cuadrática.- Aparece en ciclos anidados, por ejemplo ordenación por burbuja. También en algunas recursiones dobles.
- O(n^3) Complejidad cúbica.- En ciclos o en algunas recursiones triples. El algoritmo se vuelve lento para valores grandes de n.
- O(n^k) Complejidad polinomial.- Para ($k \in N, k>3$) mientras más crece k, el algoritmo se vuelve más lento.
- O(C^n) Complejidad exponencial.- Cuando n es grande, la solución de estos algoritmos es prohibitivamente costosa. Por ejemplo, problemas de explosión combinatoria.

Cuando se emplean las cotas asintóticas para evaluar la complejidad de los algoritmos, el análisis tiene sentido para entradas de tamaño considerable, ya que en ocasiones, comparar algoritmos utilizando un número pequeño de datos de entrada puede conducir a conclusiones incorrectas sobre sus bondades. Esto se debe a que, cuando n es pequeña, varias funciones tienen un comportamiento hasta cierto punto similar, por ejemplo O(n^3), O(n^2), O(n log(n)) y O(n). Sin embargo, conforme n crece, el comportamiento de las funciones cambia de manera radical.

Para saber qué programa es mejor, el que implementa un algoritmo con tiempo de ejecución O(n^2) u otro que implementa un algoritmo con tiempo O(n^3) obtenemos el límite:

$$\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0$$

lo que indica que el tiempo de ejecución del primero crece más lentamente que el segundo conforme n crece. De lo anterior podemos concluir que O(n^2) $\subset$ O(n^3), es decir un O(n^2) es más rápido y por lo tanto más eficiente que O(n^3).

Bibliografía

Joyanes, L. Fundamentos de Programación. Algoritmos y estructura de datos. McGraw-Hill. México. 1990.

María del Carmen Gómez Fuentes Jorge Cervantes Ojeda. (2014). Introducción al Análisis y al Diseño de Algoritmos. México: Publidisa Mexicana S. A. de C.V..