Programación

Algoritmos

IAGP
2002/11/09

La Programación de ordenadores comenzó como un arte, y aún hoy en día mucha gente aprende a programar sólo mirando a otros proceder (ej. un profesor, un amigo, etc.), y mediante el hábito, sin conocimiento de los principios que hay detrás de esta actividad.

Sin embargo en los últimos años, la investigación en el área ha arrojado teoría suficiente para que se pueda comenzar a enseñar estos principios en los cursos básicos de enseñanza de informática.

1

Problema

IAGP

Un problema es una situación donde se desea algo sin poder ver inmediatamente la serie de acciones a seguir para obtener ese algo.

Resolver un problema consiste primero que nada en comprender de qué trata y especificarlo lo más formalmente posible con el propósito de eliminar la **ambigüedad,** la **inconsistencia** y la **incompletitud**, con miras a obtener un enunciado claro, preciso, conciso y que capte todos los requerimientos.

La especificación en lenguaje natural lleva consigo problemas y se usan lenguajes más formales como las matemáticas, para evitar estos problemas.

2

Problema

Ejemplo de ambigüedad: la frase “Mira al hombre en el patio con un telescopio” se presta a ambigüedad. ¿Utilizamos un telescopio para mirar al hombre en el patio o el hombre que miramos tiene un telescopio?

Ejemplo de inconsistencia: Se entiende que en el enunciado podamos incurrir en contradicciones. Ejemplo sobre el procesador de textos:

- Todas las líneas de texto tienen la misma longitud, indicada por el usuario.

IAGP

3

Problema

Un cambio de línea debe ocurrir sólo después de una palabra, a menos que el usuario pida explícitamente la división por sílabas.

La inconsistencia resulta al preguntarse ¿Si la palabra es más larga que la línea? Entonces si el usuario no pide explícitamente la división por sílabas, esta línea tendrá mayor longitud.

Ejemplo de incompletitud: Por incompletitud entendemos que no todos los términos estén bien definidos o que no estén comprendidos todos los requerimientos del problema.

IAGP

4

Problema

En un manual de un procesador de textos encontramos la frase “Seleccionar es el proceso de designar las áreas de su documento sobre las cuales desea trabajar” ¿qué significa designar, colocar el “ratón” dónde? ¿qué significa área? ¿cómo deben ser las áreas? ¿es una sola área?

En otras palabras, se trata de encontrar una representación del problema donde todo esté dicho, sea mediante gráficos, o utilizando otro lenguaje distinto al natural. En esta etapa interviene el proceso de abstracción, que permite simplificar el problema, buscando modelos abstractos equivalentes y eliminando informaciones superfluas.

IAGP

5

Problema

Un problema no estará del todo bien comprendido si no hemos encontrado una representación en la cual todos los elementos que intervienen se representen sin redundancia, sin ambigüedad y sin inconsistencias.

Entonces el universo de búsqueda de la solución estará bien delimitado y con frecuencia surgirá en forma más clara la dificultad principal del problema.

El problema se convierte en más abstracto y más puro. Hablamos entonces de un enunciado cerrado.

IAGP

6

Problema

Por ejemplo, si queremos determinar la ruta más corta para ir de Murcia a Bogotá en avión, podemos tomar un mapa mundial y modelarlo a través de un grafo, donde los nodos o vértices serían las ciudades y los arcos las vías de comunicación aérea existentes entre ambas ciudades. A cada arco asociamos el número de kilómetros de la vía que representa y luego intentamos buscar la solución en el grafo.

En matemáticas la forma más general de enunciado de un problema se puede escribir como: "Encontrar en un conjunto X dado, los elementos x que satisfacen un conjunto dado de restricciones K(x)".

IAGP

7

Solución de un problema

Lo indicado aquí se centra en la enseñanza de la programación "en pequeño". La habilidad para hacer programas pequeños es necesaria para desarrollar programas grandes, aunque puede no ser suficiente, pues el desarrollo de programas grandes requiere de otras técnicas que no serán tratadas aquí.

Los pasos generales para resolver un problema mediante un algoritmo son:

- Especificación del problema.
- Diseño y construcción del algoritmo.

IAGP

8

Solución de un problema

La especificación de un problema consiste en precisar la información del problema, la cual la constituyen los datos de entrada y los resultados que se desea obtener, y formular las relaciones existentes entre los datos de entrada y los resultados. Para ello será necesario:

- Identificar los objetos involucrados y darles un nombre (una variable que los identifique).

- Reconocer el tipo de cada uno, que es conocer el conjunto de valores que los objetos pueden poseer y las operaciones que pueden efectuarse sobre estos.

- Describir lo que se desea obtener en términos de relaciones entre los objetos involucrados.

IAGP

9

Algoritmos

Algoritmo es un método descrito paso a paso para resolver algún problema.

Se conocen desde la antigua Babilonia.

La palabra surge del nombre de un matemático árabe del siglo XI, llamado al-Khowarizmi.

Los algoritmos son fundamentales en las matemáticas y en la informática.

Para resolver un problema en ordenador, hay que describir la solución como una serie de pasos precisos.

IAGP

10

Características

Las características de una algoritmo son las siguientes:

- ü Precisión. Los pasos se enuncian con precisión.
- ü Unicidad. Los resultados intermedios quedan definidos de manera única.
- ü Carácter finito. Se detiene después de ejecutar un número finito de instrucciones.
- ü Entrada. Recibe una entrada.
- ü Salida. Produce una salida.
- ü Generalidad. Se aplica a un conjunto de entradas.

IAGP

11

Ejemplo de algoritmo

Ejemplo de algoritmo:

1. x:=a
2. Si b>x, entonces x:=b
3. Si c>x, entonces x:=c

sirve para determinar el máximo de tres números.

Procede mediante la comparación de los números uno por uno y copia el mayor en la variable x.

Caso real, a=1, b=5, c=3. Se asigna x a 1, se asigna x a b (5), pues 5 es mayor que 1 y para terminar 3>5 es falso, luego no se hace asignación.

IAGP

12

Pseudocódigo

Aunque a veces el lenguaje común es adecuado para expresar un algoritmo, los informáticos utilizan un pseudocódigo, por su precisión, estructura y universalidad.

Se asemeja al código real de Pascal y C.

Hay muchas versiones de pseudocódigo, cualquiera es válida siempre que no existan ambigüedades.

IAGP

13

Diseño descendente

Entre las estrategias generales para resolver problemas se encuentra el **diseño descendente**.

Consiste en expresar la solución del problema como una composición de las soluciones de problemas más "sencillos" de resolver.

Una **máquina** es un mecanismo capaz de ejecutar un algoritmo expresado en función de las acciones elementales que ésta pueda ejecutar.

Cuando resolvemos mediante diseño descendente, vamos refinando la solución en términos de algoritmos cuyas acciones elementales pueden ser ejecutadas por máquinas cada vez menos abstractas

IAGP

14

Ejemplo diseño descendente

Cuando un algoritmo puede ser ejecutado por una máquina real (ordenador) decimos que el algoritmo es un programa. Un programa es un algoritmo destinado a comandar una máquina real.

Ejemplo de aplicación de diseño descendente:

Problema: Estoy en casa, deseo leer el periódico y no lo tengo. Una solución sería:

- Salir de casa.
- Ir al puesto de venta del periódico y comprarlo.
- Regresar a casa.
- Leer el periódico.

IAGP

15

Máximo tres números

Máximo de los números a, b y c

Entrada: tres números a, b, c
Salida: el máximo, x

1. Procedimiento max(a,b,c)
2. x:=a
3. Si b>x entonces
4. x:=b
5. Si c>x entonces
6. x:=c
7. Retorna (x)
8. Fin max

IAGP

16

Estructura si p entonces

En general en la estructura

Si *p* entonces *acción*

indica que si es verdadera la condición p, se ejecuta la acción y el control pasa a la proposición siguiente. Si la condición p es falsa, el control pasa directamente a la proposición posterior a acción.

Alternativamente se puede escribir,

Si *p* entonces
acción1
sino
acción2

IAGP

17

Número máximo

El siguiente algoritmo determina el número máximo en la sucesión $s_1, s_2, \ldots s_n$

Entrada: la sucesión y su longitud
Salida: mayor, el máximo de la sucesión.

1. Procedimiento *encuentra*(s,n)
2. *mayor*:=*s1*
3. *i*:=2
4. Mientras que *i*<=*n* desde
5. inicio
6. Si *si*>*mayor* entonces
7. *mayor*:=*si*
8. *i*:=*i*+1
9. fin
10. Retorna(*mayor*)
11. Fin *encuentra*(s,n)

IAGP

18

Número máximo

El siguiente algoritmo determina de otra forma el número máximo en la sucesión $s_1, s_2, \ldots s_n$

Entrada: la sucesión y su longitud
Salida: mayor, el máximo de la sucesión.

```
1. Procedimiento encuentra(s,n)
2.     mayor:=s1
3.     para  i:=2 hasta n desde
4.         Si si>mayor entonces
5.             mayor:=si.
6.     Retorna(mayor)
7. Fin encuentra(s,n)
```

IAGP

19

Número primo

Verifica si un entero positivo es primo.

Entrada: m, un entero positivo.
Salida: verdadero si es primo, falso si no lo es.

```
1. Procedimiento es_primo(m)
2.     Para i:=1 hasta m-1 desde
3.         Si m mod i = 0 entonces
4.             Retorna(falso)
5.     Retorna(verdadero)
6. fin es_primo
```

IAGP

20

Primo mayor que entero

Primo mayor que un entero

Entrada: n, un número entero positivo
Salida: m, el menor primo mayor que n

1. procedimiento *mayor_primo*(n)
2. *m*:=*n*+1
3. mientras que no *es_primo(m)* desde
4. *m*:=*m*+1
5. retorna(m)
6. fin *mayor_primo*

IAGP

21

Algoritmo de Euclides

Es un antiguo y famoso algoritmo para determinar el máximo común divisor de dos enteros no nulos. Es el máximo entero positivo que divide a ambos.

El concepto se usa para verificar que una fracción de dos enteros está reducida a su mínima expresión. Si el m.c.d. de la fracción de ambos números es 1, entonces está reducida a su mínima expresión.

Si a, b y q son enteros, b distinto de 0 y satisfacen a=bq, se dice que b divide a a y se escribe b/a, siendo q el cociente y b el divisor.

IAGP

22

Algoritmo de Euclides

Entrada: a y b, enteros no negativos
Salida: Máximo común divisor de a y b

1. Procedimiento *mcd*(*a*,*b*)
2. // Hacemos que a sea el mayor
3. Si *a*<*b* entonces
4. intercambiar (a,b)
5. // temp:=a, a:=b, b:=temp
6. Mientras *b* <>0 desde
7. Inicio
8. Dividir *a* entre *b* para obtener *a*=*b*q+r, 0<=r<b
9. *a*:=*b*
10. *b*:=*r*
11. Fin
12. Retorna(*a*)
13 fin *mcd*

IAGP

23

Algoritmo de Euclides

Ejemplo:
a=504, b=396

504 = 396 * 1 + 108
396 = 108 * 3 + 72
108 = 72 * 1 + 36
72 = 36 * 2 + 0

Resulta 36 como el máximo común divisor de 396 y 504.

IAGP

24

Algoritmos recursivos

IAGP

La recursividad es una técnica de programación que permite diseñar programas de una forma eficiente y elegante.

La utilización indiscriminada de la recursividad no es una buena estrategia.

Algunas aplicaciones importantes de la recursividad:

- Tipos recursivos de datos.
- Backtracking o vuelta atrás

25

Formas de ejecución recursivas

IAGP

La ejecución de un programa lleva consigo la invocación o activación de uno o varios subprogramas.

Para comprender mejor en qué orden se producen tales activaciones y cómo se desarrollan sus necesidades de memoria, se utilizan los siguientes conceptos:

- Árbol de activación: representación gráfica que muestra en qué orden se ejecutan un programa y los subprogramas que él contiene.

26

Noción de recursividad

IAGP

- Pila de control y registro de activación: zona de memoria utilizada por los subprogramas durante su ejecución.

La recursividad es una técnica de programación que permite transformar un problema de gran dificultad en otros subproblemas más simples del mismo tipo.

Esta característica es fundamental en la recursividad, ya que, al ser el problema original y los subproblemas del mismo tipo, se puede utilizar un único algoritmo para resolverlos a todos, modificando solamente los argumentos de entrada.

27

Noción de recursividad

IAGP

Desde el punto de vista de la programación, se dice que un algoritmo es recursivo si al ejecutarse se invoca a sí mismo directa o indirectamente. Debido a ello la recursividad se puede clasificar como:

- Directa: El algoritmo contiene una llamada a sí mismo.

- Indirecta: El algoritmo P contiene una llamada a otro algoritmo Q que, a su vez, contiene una llamada directa o indirecta a P.

28

Iteración

IAGP

Se puede concebir la recursividad como una alternativa a la iteración. El concepto de recursividad va ligado al de repetición.

Son recursivos aquellos algoritmos que, estando encapsulados dentro de una función, son llamados desde ella misma una y otra vez, en contraposición a los algoritmos iterativos, que hacen uso de bucles mientras, repetir, para, etc.

Algo es recursivo si se define en términos de sí mismo (cuando para definirse hace mención a sí mismo).

29

Factorial

IAGP

Para que una definición recursiva sea válida, la referencia a sí misma debe ser relativamente más sencilla que el caso considerado.

La función factorial es un ejemplo recursivo:

```
FUNCION Factorial (n:ENTERO): ENTERO
    SI n=0
        DEVOLVER 1
    SI NO
        DEVOLVER n*Factorial(n-1)
    FIN SI
FIN Factorial
```

30

Fibonacci

Otro ejemplo es la serie de Fibonacci:

```
FUNCION Fibonacci (n:ENTERO): ENTERO
    SI n=1 O n=2
        DEVOLVER 1
    SI NO
        DEVOLVER Fibonacci(n-
        1)+Fibonacci(n-2)
    FIN SI
FIN Fibonacci
```

IAGP

31

Torres de Hanoi

Otro problema muy conocido es Torres de Hanoi:

```
Algoritmo
Hanoi(num_discos,origen,destino,auxiliar)
    INICIO
    SI num_discos=1
        ESCRIBIR origen, '->', destino
    SI NO
        Hanoi(num_discos-1,origen,auxiliar,destino)
        ESCRIBIR origen, '->', destino
        Hanoi(num_discos-1,auxiliar,destino,origen)
    FIN SI
FIN
```

IAGP

32

Cuando usar la recursividad

IAGP

La recursividad debe ser comprendida para poder aplicarla en problemas que posean una solución algorítmica recursiva más eficiente y/o elegante.

Los programadores que comienzan a usar la recursividad para el diseño de algoritmos suelen atravesar las siguientes fases:

- Un fuerte rechazo inicial, ya que no comprenden la filosofía que conlleva.
- Una vez que "creen" haber comprendido su esencia, la utilizan indiscriminadamente siempre.

33

Cuando usar la recursividad

IAGP

Los algoritmos recursivos son apropiados cuando el problema a resolver o los datos a tratar se definen de forma recursiva. Sin embargo, esto no garantiza que la mejor forma de resolver el problema sea así.

Debe evitarse el uso de la recursividad cuando haya una solución obvia por iteración .

Si un problema se puede solucionar de forma recursiva e iterativa, generalmente la mejor solución será la iterativa. Las razones son:

- Ahorrar memoria de pila
- Evitar calcular dos veces el mismo valor

34

Eficiencia y complejidad

IAGP

Un mismo problema informático se puede resolver mediante varios algoritmos diferentes. Para elegir entre ellos hay que evaluar su eficiencia.

Un algoritmo es más eficiente, o lo que es igual, menos complejo, cuando usa menos recursos.

Los recursos con los que se trabaja en un algoritmo, y por tanto los que vamos a tener en cuenta para medir su complejidad son:

35

Eficiencia y complejidad

IAGP

- El espacio utilizado en memoria: determina la complejidad espacial. Es la suma del tamaño de todas las variables y espacios de almacenamiento usados. Es fácil de calcular para las variables, pero no tanto cuando se usa memoria dinámica

- El tiempo de ejecución: determina la complejidad temporal del algoritmo.

Generalmente estos dos recursos se convierten en objetivos contrapuestos: los algoritmos con baja complejidad temporal suelen tener una complejidad espacial alta y viceversa.

36

Factores que influyen

IAGP

En el cálculo de la complejidad hay tres factores:

Ø El tamaño de los datos de entrada: el número de bits que ocupan, aunque se usa el número de elementos lógicos de que constan (ej., número de elementos en un vector a ordenar). Cuando se trata de problemas numéricos (ej., el factorial) se usa, en cambio, el valor del dato.

Ø El contenido de los datos de entrada: a partir de algoritmos medianamente complejos, los valores de los datos de entrada pueden causar la activación o no de estructuras condicionales, cambiar el número de iteraciones de los bucles.

37

Factores que influyen

IAGP

Ø El compilador y la máquina: el tiempo y/o el espacio necesitados por un algoritmo pueden verse modificados según la calidad de las optimizaciones del compilador y según la potencia de la máquina en que se ejecute. Sin embargo, no queremos particularizar, sino obtener la eficiencia abstracta.

Hay tres posibles enfoques para el estudio de la eficiencia de un algoritmo:

Ø Empírico o a posteriori: se implementa el algoritmo en un programa y se realizan pruebas con datos de distinto tamaño y distintos valores

38

Enfoques estudio

IAGP

- Ø Teórico o a priori: se determina matemáticamente la cantidad de recursos necesarios para ejecutar el algoritmo y se obtiene una función f(n), que indica la cantidad de recursos y donde n es el tamaño de los datos de entrada.

- Ø Híbrido: se obtiene la función de eficiencia de forma empírica y luego se obtienen los parámetros adicionales para calcular la eficiencia real de una implementación determinada.

39

Conclusión

IAGP

Lo importante no es que un algoritmo tenga una 'buena' eficiencia, sino que tenga la mejor de entre los posibles. Se debe considerar la eficiencia de forma relativa.

Esta consideración relativa debe tener en cuenta que el estudio de eficiencia es asintótico, y sólo válido para valores de n suficientemente grandes.

Por tanto, un algoritmo con eficiencia peor que la de otro puede ser mejor que éste para tamaños de n pequeños.

40