Algoritmos y Programas

Ing. Luis Roberto Ojeda Ch.

Introducción

Este artículo presenta un tema fundamental en ciencias de la computación: la posible existencia de algoritmos para realizar cualquier tarea. Se trata de precisar esta posibilidad siguiendo esquemas de raciocinio satisfactorios aunque no excesivamente rigurosos, a fin de motivar a una amplia audiencia alrededor de estos temas.

I. Computabilidad

Existen algoritmos para tejer una cobija, para preparar un pastel, para hacer un vestido.

Los computadores pueden controlar semáforos, líneas de producción y plantas siderúrgicas.

Hay algoritmos básicos conocidos por todos: sumar, restar, multiplicar, dividir, sacar raíces cuadradas.

¿Existe algún trabajo que el computador no pueda realizar? ¿Algún trabajo para el que no exista un algoritmo?

La idea de poder disponer de un algoritmo o receta para efectuar alguna tarea es milenaria.

Durante mucho tiempo se creyó que si cualquier problema podía enunciarse de manera precisa, entonces con suficiente esfuerzo sería posible encontrar una solución con el tiempo o en su defecto una prueba de la no existencia de solución.

En otros términos se creía que no había problema que fuera intrínsecamente tan difícil que en principio nunca pudiera resolverse en el sentido de encontrarle la solución o de establecer la imposibilidad de hallarla.

David Hilbert (1862-1943) fue el propulsor principal de esta apreciación. La meta de Hilbert era crear un sistema matemático formal en el que todos los sistemas pudieran plantearse con proposiciones que solo serían o verdaderas o falsas.

Su idea era encontrar un algoritmo que dada una proposición cualquiera en el sistema formal, determinara si esa proposición era o no verdadera.

Si Hilbert hubiera logrado su objetivo, cualquier problema que estuviera bien definido podría haberse resuelto simplemente al ejecutar el algoritmo.

Hilbert consideró el problema de decidir la verdad de una proposición dada en el sistema formal como un problema fundamental de las matemáticas y lo denominó **Entscheidungsproblem.**

En la década de 1930 se produjó un conjunto de investigaciones que mostraron que el Entscheidungsproblem no es computable, es decir, que no puede existir un algoritmo como el que buscaba Hilbert.

En 1931 Kurt Godel publicó el terorema de incompletitud. Este teorema mostró entre otras cosas que no existe algoritmo cuya entrada sea cualquier proposición sobre los números enteros y cuya salida indique si la proposición es o no verdadera.

Otros matemáticos como Alonzo Church, Stephen Kleene, Emil Post y Alan Turing encontraron más problemas carentes de solución algorítmica. Esto en la década del 30, antes de que se hubieran construido los primeros computadores.

II. Conjuntos Contables

Para precisar algunos conceptos sobre computabilidad requerimos introducir la idea de conjunto contable.

Un conjunto es contable si cada uno de sus elementos puede ser ligado a un entero positivo. Un conjunto es incontable cuando esto no es posible.

La mecánica de asignación de los elementos de un conjunto a los enteros positivos puede verse así:

Existe una hilera infinita de casillas postales rotuladas 1,2,3,4,5,...

* Ingeniero de Sistemas, Profesor Asociado, Departamento de Ingeniería de Sistemas Universidad Nacional de Colombia

Tomamos un conjunto y comenzamos a colocar elementos del conjunto en las casillas, alojando como máximo un elemento en cada casilla. Si encontramos una forma de asignación el conjunto es contable.

Por ejemplo consideremos el conjunto de todos los números enteros pares: [2,4,6,8,...]

La regla de asignación puede ser: Asignar el entero i a la casilla i/2.

Se observa que el conjunto es contable.

III. Los Programas y Los Conjuntos Contables

¿Podemos acaso imaginar el conjunto de todos los programas de computador que puedan existir?

En tal caso; será este un conjunto contable?

Todo programa puede verse como una cadena de caracteres (incluyendo puntuación, espacios, y caracteres especiales).

Supongamos que el número (siempre finito) de caracteres que pueden usarse para escribir cualquier programa es *P* (en ASCII P está limitado a 128).

Si el número total de cadenas que se puede construir con *P* es contable, el número de programas tendrá también que ser contable (incluso, muchas de tales cadenas no serán realmente programas).

Podemos asignar cadenas a las casillas como sigue:

- Asignar la cadena vacía a la primera casilla.
- Ahora tomar todas las cadenas de un carácter y asignarlas a las siguientes *P* casillas (en algún orden, p.e. alfabético).
- Ahora tomar todas las cadenas de dos caracteres y asignarlas a las siguientes *P**2* casillas (nuevamente en un orden, p.e. alfabético, como en un directorio).
- Proceder así con todas las cadenas de tres caracteres, de cuatro caracteres, etc.

De esta forma una cadena de cualquier longitud puede ser asociada con una casilla. El conjunto de cadenas es contable y por tanto el número de programas que ser escritos también lo es.

IV. El Número De Funciones Matemáticas Y Los Conjuntos Contables

Para simplificar consideremos las funciones que toman un solo entero positivo como entrada y que generan un solo entero positivo como salida.

Denominemos a tales **funciones enteras.**

Queremos ver si es posible asignar todas las funciones enteras al conjunto infinito de casillas.

Si esto es posible, el número de funciones enteras será contable. Si el conjunto de funciones enteras no resulta contable, habrán funciones enteras que no podrán asociarse con ningún programa. Recuérdese que el casillero está ligado en cada una de las casillas con un programa (o mejor, con una cadena de caracteres en *P*) y que por tanto cada casilla es equivalente a una cadena y se trata de que también cada casilla sea equivalente a una función entera.

V. Como Describir Las Funciones Enteras Para Asignarlas A Las Casillas

Argumento	Valor
Infinito numero de filas	

Visualización de una función

Cada función entera será una tabla de dos columnas, la primera para el argumento y la segunda para el valor de la función, y habrá un número infinito de filas para cubrir todo par (argumento, valor).

Podemos idear que:

- En la casilla 1 va la tabla de la función que siempre retorna 1 para cualquier argumento.
- En la casilla 2 va la tabla de la función que retorna exactamente el mismo valor del argumento, para cualquier argumento.
- En el casillero puede ir cualquier función (lo mismo que hemos hecho en las casillas 1 y 2) p.e. *y=2*x+5,* etc.

¿Puede asignarse cada función a una casilla?

Supongamos que el matemático Pérez presenta una forma de asignación que presuntamente cubre la totalidad de funciones enteras.

Siempre podremos a partir de las funciones encasilladas, generar una nueva función que no ha sido asignada, así:

- Vamos a la casilla 1 y tomamos el valor de salida para el valor de entrada 1. Sea este F1(*1*).
- Sumamos 1 a F1(*1*) y tomamos este resultado como la salida de la nueva función para la entrada 1.
- Sin considerar el resto de salidas en la nueva función, ella será diferente de la asignada a la casilla 1, porque su salida para la entrada 1es F1(*1*)+1 y no F1(*1*).
- Ahora tomamos la salida para la función en la casilla 2 y para el valor de entrada 2. este será F2(*2*). Adicionamos 1

a esta salida y digamos que F2(2)+1 es la salida de la nueva función para la entrada 2.

De esta forma la nueva función difiere ya de la función en la casilla 1 y también de la función en la casilla 2.

- En general y para que la nueva función difiera de todas las encasilladas, hacemos:

Fnueva(i)=F(i)+1

O sea:

Fnueva(1) = F1(1)+1
Fnueva(2) = F2(2)+1
Fnueva(3) = F3(3)+1
etc.

Así tenemos una función nueva que no estaría contemplada en el método propuesto por Pérez. Y cualquier método propuesto será igualmente refutado.

El significado de esto es que el conjunto de funciones enteras es incontable y no puede ligarse con el conjunto de programas que si lo es.

Nunca podremos construir un programa para cada función o sea que hay funciones que no son definibles mediante programas, pero que si pueden expresarse mediante una tabla.

VI. Conjuntos Infinitos De Diferente Tamaño

Alguien podría argumentar que, al fin de cuentas, tanto el conjunto de todos los programas como el conjunto de todas las funciones enteras son infinitos y que siendo así debe existir alguna correspondencia biunívoca, de algún modo.

Los conjuntos con un número infinito de elementos no contienen necesariamente la misma cantidad de elementos y hay conjuntos infinitos que son más grandes que otros conjuntos infinitos. Existen varios niveles de infinitud, donde algunos conjuntos infinitos son mayores que otros.

Se habla de cardinalidad de los conjuntos. Cuando se trata de conjuntos finitos la cardinalidad corresponde al concepto tradicional de número de elementos en el conjunto, pero la cardinalidad de un conjunto infinito no es un número en el sentido habitual. La cardinalidad de un conjunto A se representa como |A|.

Es necesario establecer una forma de comparación de las cardinalidades de dos conjuntos diferentes.

$$|X| <= |Y|$$

Si y solo puede establecerse una correspondencia de cada elemento de X con un elemento único y distinto de Y.

En otras palabras $|X|<=|Y|$ si y solo si hay una función uno a uno de X a Y.

Si $|X|<=|Y|$ *y* $|Y|<=|X|=|Y|$ *entonces*
$|X|=|Y|$

Un ejemplo claro de conjuntos infinitos de diferente cardinalidad se puede construir a partir de la noción de **conjunto potencia de un conjunto.**

Dado un conjunto X de cardinalidad $|X|P(X)$ es el conjunto potencia de *X* si corresponde al conjunto de todos los subconjuntos de *X*.

Para todos los conjuntos finitos *X* con tamaño mayor o igual que uno se tiene:

$|P(X)|=2**|X|$

Por ejemplo, si X={1}el conjunto P(X) es {{},{1}}

Si X={1,2} P(X) es: {{},{1},{2},{1,2}}

A partir de aquí se puede formular y probar el teorema que dice que:

Si *X* es un conjunto cualquiera entonces $|X|<|P(X)|$

Y así podemos apoyar la afirmación de que los conjuntos infinitos pueden tener cardinalidades distintas.

Otro teorema que se puede probar es que la cardinalidad del conjunto de los números naturales (que es el conjunto que nos ha permitido rotular las casillas y enumerar los programas posibles en un lenguaje) es menor o igual que la cardinalidad de cualquier otro conjunto infinito.

VII. El Problema De Detención

Un problema común al usar un computador para correr un programa es que la persona que escribió el programa cometió errores que impiden que el programa termine.

Se dice que el programa entra en un ciclo infinito.

Sería muy grato si los programadores pudieran detectar la existencia de ciclos infinitos en sus programas antes de ejecutarlos. Que existiera un algoritmo para la tarea, en lugar de que cada caso sea un problema distinto, tan distinto como puede ser un ordenamiento de burbuja y el método de rumbo y distancia en topografía.

En otras palabras se trataría de: *determinar si un programa cualquiera se detiene o no, mediante un algoritmo. A esto se le conoce como el problema de detención (Halting problem).*

La solución es un algoritmo que, dado un programa cualquiera *P* y sus datos de entrada *D* pueda indicarnos si con

el tiempo *P* se detendrá al ejecutarse con sus datos *D*.

Desafortunadamente se ha demostrado que no puede existir el mencionado algoritmo.

VIII. Análisis Del Problema De Detención

Vamos a suponer que si existe el algoritmo que toma un programa y sus datos y que anuncia si el programa se detiene o no.

HIPÓTESIS: la existencia de un algoritmo cuya forma es:

Algoritmo pruebadetención (P, D)
Comienzo
Si P se detiene con los datos D
entonces reportar SI
sino reportar NO
fin

contando con el algoritmo anterior se puede escribir el algoritmo nuevapruebadetencion así:

algoritmo nuevapruebadetención (P)
comienzo
pruebadetención(P,P)
fin

prueba detención estará aquí verificando si *P* se detiene cuando sus datos son el mismo *P*.

Un caso que muestra que efectivamente un programa *P* puede aceptar a *P* como sus datos puede ser aquel en que *P* es un traductor de un lenguaje de programación a otro (o al lenguaje de máquina). De esta manera, *P*, escrito en un lenguaje, traduce a *P* a otro lenguaje. Conviene anotar que este caso si es viable algorítmicamente.

Ahora escribimos el algoritmo sorprendente así:

algoritmo sorprendente (*P*)
comienzo
Si nuevapruebadetención (P) encuentra que P se detiene
entonces ejecutar un ciclo infinito
sino detenerse
fin

Vamos a ejecutar paso a paso a sorprendente aplicado sobre sorprendente. sorprendente (sorprendente)

Primero evaluamos nuevapruebadetencion (sorprendente)

Esto requiere evaluar pruebadetencion (sorprendente, sorprendente)

Pruebadetencion va a establecer si sorprendente se detiene o no.

O sea, si se detiene PRECISAMENTE el proceso que EN ESTE MOMENTO estamos siguiendo paso a paso y que no hemos terminado de seguir.

Puede suceder que:

- Pruebadetencion (sorprendente, sorprendente) encuentre que si hay detención. En tal caso, continuando con el seguimiento paso a paso de sorprendente, debemos entrar en un ciclo infinito.

  Esto quiere decir: si se diagnosticó que sorprendente se detiene, a continuación sorprendente entro en un ciclo infinito (sorprendente no se detiene, contrariando al diagnóstico).

- Pruebatencion (sorprendente, sorprendente) encuentre que no hay detencion.

Ahora, regresando al seguimiento paso a paso, sorprendente se detiene, en contra de lo previsto.

La contradicción solo puede resolverse admitiendo que sorprendente no puede existir.

La única consideración que se hizo para construir sorprendente fue la existencia de pruebadetencion.

Por tanto pruebadetencion no puede existir.

Hay un número de problemas suficientemente simples como para que se pueda determinar si se detienen o no (antes de ejecutarlos).

Pero existen otros complicados.

IX. El Último Teorema De Fermat

Este teorema enuncia que **no existen** tres enteros positivos a, b y c tales que:

Si m=a elevado a la potencia n
p=b elevado a la potencia n
q=c elevado a la potencia n

(cuando n>2)

m+p=q

No se conoce prueba a favor o en contra de este teorema.

El siguiente procedimiento es interesante:

Procedimiento fermat(n)

repetir para a=1,2,3,4,.....,n
repetir para b=1,2,3,,,a
repetir para c=1,2,3,,,,...a+b

Si a**n+b**n=c**n
Entonces dar salida a: a,b,c y n y
detenerse

Si el algoritmo efectivamente se detiene para una entrada específica con n>2 se habrá probado que el último teorema de Fermat es falso.

Cuando a=1, b=1 y c=2 hay detención siendo n=1
Cuando a=3, b=4 y c=5 hay detención para n=2
Cuando n=3 se dice que se requiere ingenio y paciencia para encontrar que nunca se detendrá.

También fue demostrado que tampoco se detiene cuando n= 37

X. Otro Ejemplo: La Secuencia De Collatz

Verifiquemos el siguiente código:

```
while (n>1)
      if (ODD(n))
            n=3*n+1;
            else n=n/2
```

La secuencia de valores asignada a *n* es la llamada la secuencia de Collatz. No se ha logrado establecer si el código siempre se detiene.

Conclusiones

Se ha presentado una panorámica rápida de un tema vital en ciencias de la computación: la existencia de algoritmos para ejecutar tareas. Los métodos expuestos para enfrentar el asunto no son los más rigurosos pero se ajustan y conducen a los hallazgos aceptados universalmente en este terreno y requieren poco esfuerzo para ser comprendidos.

Se dice popularmente que un problema bien planteado equivale a un problema casi resuelto. Esto se encaja dentro del esquema más básico propuesto por Hilbert. El caso del problema de detención es un caso bien planteado y efectivamente queda resuelto: No hay posibilidad de algoritmo que establezca de antemano si un algoritmo se detiene o no. La idea de relacionar la totalidad de los programas posibles en cualquier lenguaje con la totalidad de las funciones enteras es también claramente planteada y la conclusión en tal caso también queda establecida. No hemos tocado el punto de cómo se puede plantear el proyecto de Hilbert (la posibilidad de un algoritmo al que se someta cualquier problema y que le trace a este el camino de solución o de negación de la misma) de modo que se llegue a la conclusión de que tal algoritmo no puede existir. Es un reto interesante hallar una forma racionalmente sencilla de percibir la prueba, y lo dejamos como tal.

Se espera que este trabajo atraiga la atención de personas que se inician en el estudio profundo de la computación y que tal vez motive a plantear exposiciones rigurosas entre quienes ya se encuentran interesados en el campo.

Bibliografía

1. GOLDSCHLAGER, Les y LISTER, Andrew. ***Introducción Moderna a la Ciencia de la Computación con un Enfoque Algoritmico***. Prentice Hall, 1986.

2. SHAFFER, Clifford A.. ***Data Structures and Algorithm Analysis***. Prentice Hall, 1997

3. VAN LE, T. ***Techniques of PROLOG programming***. *John Wiley, 1993.*