¿Qué es una computadora?

Manrique Mata-Montero*
Virginia Berrón Lara**

Resumen

En este artículo nos planteamos la pregunta: ¿qué es una computadora? Para responderla adoptamos una perspectiva evolutiva considerando sus orígenes y desarrollo. Proponemos y discutimos algunos aspectos del tema que nos parecen relevantes para adquirir una mejor comprensión de lo que es una computadora.

Abstrait

Dans cet article nous nous demandans: ¿qu´est-ce qu´un ordinateur? Por lui répondre nous adoptons une perspective evolutive en considirant ses origines et développement. Nous proposons et nous diswtons quelques aspects du théme qui nous semble le plus importants pour avoin une meilleur comprehention de ce qu´est un ordinateur.

Abstract

In this article we look at the question: What is a computer? To answer it we adopt an evolutionary perspective which examines its origin and development. We raise and discuss those aspects which we consider relevant towards acquiring a better understanding of what a computer is.

1. Introducción

Presumiblemente cualquiera que intente dar una respuesta a la pregunta: ¿qué es una computadora?, debería saber lo que es una computadora. Pero, puesto que existen muchos enfoques para responder a esta pregunta, ¿cómo decidimos cuál es la definición acertada?.

Para ciertos objetos una descripción precisa de su morfología y su composición física y/o química indica lo que son. Cualquiera que conozca su descripción sabe lo que son esos objetos. Desafortunadamente la morfología y la constitución física de una computadora son casi irrelevantes.

Para los objetos "abstractos", tales como los números reales, la sola definición matemática no parece bastar, puesto que parece adecuado argüir que para conocer los números reales, uno debería conocer también todas sus propiedades, p.j., $x \cdot x \geq 0$ para cualquier número real x. Esto nos lleva a la pregunta de si hay alguien que conozca los números reales, ya que hay un número infinito de hechos o propiedades de los mismos. Los matemáticos han atacado esta situación creando lo que se conoce como un "sistema formal", dentro del cual todas las propiedades de los números reales (¿todas realmente?) se podrían inferir.

Entonces se podría decir que conocer lo que los números reales son comprende el conocimiento de su definición matemática y la del sistema formal que rige su manipulación. Lo malo de esta interpretación del concepto de conocimiento, aunque fuera adecuado, es que tal sistema formal no puede existir (Hofstadter 1980).

Así las cosas ¿qué significa conocer los números reales?: ¿conocer una metateoría de los números reales? o ¿una metateoría de una metateoría de los números reales?

Cuando investigamos la naturaleza de la computadora, tenemos la misma dificultad que cuando consideramos a los números reales unido al hecho de que la computadora no sólo es un objeto (abstracto) matemático, sino uno que además tiene una materialización en un objeto que es casi autónomo, i.e., un objeto que puede funcionar casi independientemente de la intervención humana. Así, parece adecuado argumentar que si alguien conoce lo que es una computadora, debería saber: su definición matemática, sus propiedades, la clase de tareas que puede llevar a cabo y cómo, y las maneras en las que este objeto abstracto puede ser materializado.

Entonces, de acuerdo con esta definición, no hay ser humano (hasta donde sabemos) que pueda afirmar a ciencia cierta que conoce lo que es realmente una computa-

* *Profesor visitante en la División de Posgrado de la Universidad Tecnológica de la Mixteca.*
** *Profesor-Investigador de la División de Posgrado de la Universidad Tecnológica de la Mixteca.*

dora. Pedimos a nuestros lectores expectativas más modestas y que acepten leer el planteamiento acerca de las computadoras de quienes tienen un conocimiento parcial acerca de las mismas.

En este artículo nos planteamos cada uno de los aspectos que caracterizan a la computadora y esperamos con eso ganar un conocimiento parcial acerca de su naturaleza. Una "invención" es un acontecimiento "cultural"; como muchos autores sugieren (Dennet 1996), " si Newton no hubiera inventado el cálculo, otro lo hubiera hecho"[1]. Esto no intenta minimizar el mérito dado a los que desde siempre han sido considerados como los inventores de la computadora[2]. Es sólo el reconocimiento del hecho de que las ideas (o los inventos) obedecen las leyes de la evolución y que su posibilidad puede ser establecida de acuerdo en qué parte del árbol de la evolución (árbol de la vida) estemos viviendo[3].

Nuestro uso (deliberado) de la expresión "la computadora" más que la de "las computadoras" está basado en el hecho de que hay fundamentalmente una computadora en el sentido matemático, y que la pluralidad que nosotros vemos es meramente un aspecto tecnológico o de ingeniería[4]. Esto puede parecer una sorpresa, pero veremos que podemos hacer una evaluación del poder de la computadora independientemente de su aparente "variedad". Más aún, mostraremos que hay sustancialmente más problemas matemáticamente *insolubles* que *solubles*, y que de entre aquellos matemáticamente o lógicamente solubles muchisímos (en realidad, "los interesantes") son prácticamente insolubles. A estos les llamamos *intratables*. Finalmente, hablaremos de cómo están construidas las computadoras usando circuitos electrónicos y cómo nos comunicamos con ellas.

2. Las raíces de la idea de una computadora

A pesar de haber adoptado la posición de que la "invención" de la computadora no fue un hecho aislado de una persona o un grupo de personas, en un espacio y tiempo particulares, separamos su desarrollo en antes y después del año 1900. La idea de construir una máquina computadora autónoma se ha difundido e intentado llevar a cabo en nuestra cultura desde tiempos tan tempranos como mediados del siglo XVII con el Calculador de Blas Pascal, en 1801 con el Telar de J. Jacquard, en 1821 con la Máquina de Diferencias de C. Babbage y más tarde con su Máquina Analítica.

Sin embargo, no es sino hasta 1900 que las preguntas adecuadas (correctas), y poco después las respuestas correctas, fueron claramente establecidas y encontradas con respecto a la posibilidad de tal artefacto autónomo. Por preguntas y respuestas adecuadas entendemos aquellas que clara y precisamente establecidas son relevantes para la existencia de un mecanismo de computación autónomo.

La primera persona que públicamente planteó problemas que ahora consideramos como parte de los fundamentos de la teoría de los calculadores artificiales (artificiales porque en aquellos tiempos se entendía calculador como calculista, es decir un ser humano que realizaba cálculos) fue el matemático alemán David Hilbert en 1900. En 1928 Hilbert presentó su proyecto de nuevo, esta vez a partir del punto de vista de los "sistemas formales".

En el congreso de 1928 Hilbert precisó sus planteamientos:

- Primero: ¿se puede probar si las matemáticas son completas? La completitud se entiende en este caso como la propiedad de que toda proposición (en este caso matemática) puede ser probada o refutada.

1 En realidad, Leibniz afirma haberlo hecho antes que Newton. Existen comunicaciones escritas con argumentaciones serias entre ellos acerca de esta controversia.

2 Incluyendo a Konrad Zuse (1910-1995), su inventor alemán. Para mayor información acerca de su vida y obra: http://hjs.geol.uib.no/zuse/zusez1z3.htm.

3 Para una discusión detallada acerca de la "posibilidad" desde un punto de vista evolutivo vea Richard Dawkins. *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe without Design*. Norton 1986; Daniel C. Dennet. *Darwin's Dangerous Idea: Evolution and the Meanings of Life*. Touchstone Books, 1996.

4 Este aspecto es también muy interesante como lo sabe cualquiera que por lo menos lea con regularidad la sección de los periódicos correspondiente.

- Segundo: ¿son las matemáticas consistentes? En el sentido de que ninguna afirmación que nosotros hemos mostrado anteriormente como falsa pueda también demostrarse como verdadera por medio de una sucesión de pasos lógicamente válidos (resultando verdadera y falsa a la vez).
- Tercero: ¿son las matemáticas decidibles? Es decir, ¿existe un método definitivo que podría, en principio, ser aplicado a cualquier afirmación para producir una decisión correcta acerca de si la afirmación es verdadera? (Hodges 1983)

Las preguntas de Hilbert amalgamaron temas provenientes del trabajo en lógica iniciado por Frege y continuado por Sir B. Russell y A.N. Whitehead. Hilbert conjeturó y defendió vehementemente una respuesta afirmativa a estos tres cuestionamientos en aquel congreso de 1928. Sorpresivamente, en el mismo congreso su antiguo estudiante Kurt Gödel demostró que, en relación a la consistencia y completitud de cualquier formalización de las matemáticas, la única respuesta afirmativa corresponde a la pregunta acerca de la consistencia, i.e., lo mejor que se podría alcanzar es una formalización de las matemáticas consistente pero incompleta.

Es digno de notarse que en los primeros años del siglo XX no era claro lo que Hilbert entendía por "método definitivo (claro, determinado)" o "proceso mecánico". Así, implícito en la respuesta a la tercera pregunta debería de haber una definición satisfactoria de lo que eso significaba.

La ingeniosidad de Gödel le permitió codificar las afirmaciones acerca de los números naturales junto con sus respectivas pruebas como ¡números naturales![5] De esta manera se codificó de una manera efectiva toda la teoría de la aritmética dentro de la aritmética.

Él logró probar que cualquier teoría capaz de formalizar a la aritmética no puede ser a la vez consistente y completa. En el mejor de los casos, puede ser consistente e incompleta o exclusivamente completa pero inconsistente. Como lo mencionamos antes, en caso de que una teoría pueda ser consistente, su consistencia no puede ser probada dentro de la teoría misma. Su brillante idea consistió en codificar dentro de su formalismo una proposición que básicamente dice: *Esta proposición no puede ser probada*[6].

Claramente, cualquier prueba de falsedad o verdad de esta proposición conduce a una contradicción con ella misma. Gödel dejó abierta la pregunta de si podría haber un "proceso mecánico" con el que proposiciones indemostrables (dentro de un formalismo dado) podrían ser encontradas[7]. La respuesta a esta pregunta tuvo que esperar al "invento" de Turing en 1936.

3. La máquina de Turing, algoritmos y problemas indecidibles

Después de estudiar los resultados de Gödel, Turing dio no sólo una definición satisfactoria de lo que sensatamente podría llamarse "proceso mecánico", también probó que hay afirmaciones en matemáticas que son indecidibles o insolubles.

Debería recordarse que si bien la definición de Turing no contempla el "proceso mecánico", sí pareció (y todavía parece) ser tan apropiada que dio lugar a lo que comúnmente se conoce como la *Tesis Church-Turing*[8]. Esta tesis establece básicamente que todo proceso que razo-

5 Fundamental en el trabajo de Gödel y de Turing es la idea de "codificación" (esta idea había sido ya usada en los 1800's para representar números reales como sucesiones infinitas de dígitos (su codificación fue formalizada por Dedekind en 1872)). Una codificación es una representación preservadora de significado de una sucesión de símbolos por otra. La correspondencia entre sucesiones es arbitraria (normalmente descrita por medio de un homomorfismo) pero esta correspondencia debe ser tal que el significado de ambas sucesiones es el mismo, i.e., lo que entendemos por una sucesión deberá ser entendida por otra. Una codificación es un cambio de forma, p.j., el número entero siete cuya codificación decimal es 7 y en binaria es 111.

6 Nótese el carácter autoreferencial de esta proposición.

7 Note que esta pregunta es una variación de la pregunta original establecida en términos de verdad y no demostrabilidad.

8 En honor de Alonzo Church (1903-1995), asesor de doctorado de Alan Turing, creador del l-cálculo, estableció la Hipótesis que lleva su nombre acerca de la noción intuitiva de algoritmo. Vea http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Church.html

nablemente puede ser llamado mecánico puede ser ejecutado por una máquina de Turing.

Casi por definición esta tesis no admite una prueba empírica, porque no sabemos qué es exactamente "proceso mecánico", a lo más podemos probarla con aquellos procesos que consideramos mecánicos. Claramente, este punto de vista nunca nos llevará a una comprobación de esta tesis, pero sí a refutarla. La Tesis Church-Turing se ha convertido en una de las hipótesis metafísicas de las ciencias de la computación y sus campos afines[9], tales como la psicología cognoscitiva, la inteligencia artificial y, más recientemente, la vida artificial.

La Máquina de Turing es un aparato sencillo, lo que la hace particularmente llamativa y elegante. Consiste de una cinta de tamaño arbitrario dividida en celdas o cuadros, cada uno de ellos capaz de almacenar o contener un símbolo de un alfabeto predeterminado, un control finito que puede estar en un estado perteneciente a un conjunto predeterminado de estados, uno de los cuales será identificado como el "estado inicial", una cabeza lectora/escritora que puede moverse de izquierda a derecha, una celda de la cinta a la vez para escribir o leer en la cinta, y para cada estado y posible símbolo de entrada una tabla que especifica cómo los estados de control cambian, qué símbolo escribe la máquina en la cinta y la dirección del movimiento de la cabeza lectora/escritora. Esta tabla puede ser vista en realidad como un conjunto de instrucciones u órdenes que determinan el comportamiento del aparato.

La máquina funciona en intervalos de tiempo discretos, ejecutando una instrucción a la vez, si hay una instrucción aplicable al estado en que la máquina está y al símbolo de la cinta va a ser leído. Por convención, al comienzo del cómputo, la secuencia de longitud finita (la secuencia de entrada)[10] que forma los datos de entrada está ya colocada en la cinta, la cabeza lectora/escritora está colocada sobre el símbolo de la extrema izquierda de la secuencia de entrada y el control está en el estado inicial. También suponemos que si la cabeza de la máquina se mueve hacia la derecha del símbolo situado a la extrema derecha de la secuencia almacenada en la cinta, encontrará una celda con un símbolo de "espacio en blanco".

Un cómputo de la máquina de Turing a partir de una secuencia de entrada de largo finito es una sucesión de configuraciones, cada una de las cuales pasa a la siguiente en la sucesión de acuerdo con la tabla de instrucciones hasta que se alcanza la última configuración. La última configuración se alcanza cuando ninguna orden o regla en la tabla es aplicable a la configuración de la máquina. Una configuración consiste de: el estado de la máquina, la secuencia de símbolos que en ese momento están almacenados en la cinta y de la posición de la cabeza lectora/escritora sobre la cinta. La respuesta de la máquina consiste de los símbolos que aparecen escritos en la cinta, cuando la máquina para. En algunos casos puede no haber una última configuración, como en aquellos casos en que la máquina entra en un ciclo infinito y nosotros decimos que la salida de la máquina está indefinida.

Hay muchas variaciones de este modelo básico, todas ellas equivalentes en el sentido de que todo lo que puede hacerse con un modelo puede hacerse con el otro. Además, cualquier otro modelo de cómputo inventado hasta ahora ha resultado ser equivalente a un modelo de la máquina de Turing (MT), lo cual incrementa nuestra confianza en que la MT sea un modelo de cómputo adecuado.

Una descripción total de una máquina de Turing puede ser dada por una secuencia finita de símbolos provenientes de un alfabeto adecuado. Para hacer este hecho evidente incluimos su definición matemática.

Una Máquina de Turing M es una 6-tupla $(\Sigma,\Gamma,\delta,B,s_0,Q)$ tal que:

9 La idea de que la computación es la computación hecha con una máquina de Turing está actualmente tan integrada en la cultura científica popular que invariablemente la definición de computación es equiparada con la definición de computación con una máquina de Turing.

10 A partir de ahora llamaremos indistintamente a los datos de entrada "entrada" y a los de salida "salida".

- Σ es un alfabeto finito de símbolos de entrada. Se puede probar que en realidad nos basta considerar un alfabeto con dos símbolos generalmente denotados por 0 y 1.
- Γ es un alfabeto finito de símbolos en la cinta, aquellos que la máquina puede escribir en la cinta, más el símbolo B. Todos los símbolos de Σ pertenecen a Γ.
- B representa el símbolo en blanco el cual está en Γ y aparece en toda celda que la cabeza lectora/escritora alcanza después de moverse a la derecha del símbolo en la parte extrema derecha de la secuencia almacenada en la cinta.
- Q es un conjunto finito de estados.
- s_0 es el estado de inicio, elemento del conjunto Q, y
- δ es una función descrita como una tabla con argumentos en $Q \times \Gamma$ que toman como valores algunas tripletas pertenecientes a $Q \times \Gamma \times \{L(eft), R(ight)\}$.

Note que la descripción completa de esta tabla es finita ya que ambos el dominio y el contradominio de la función son finitos.

3.1 Un ejemplo de una Máquina de Turing

Definamos una máquina con los siguientes valores para sus parámetros:

- el alfabeto de entrada incluye los símbolos *a* y *b*,
- el alfabeto de la cinta incluye *a*, *b*, *x*, *Y* y *N*,
- el símbolo en blanco está representado por *x*,
- el conjunto de estados contiene *estado*1, *estado*2, *estado*3,
- estado inicial: *estado*1, y
- función de transición dada en la tabla 1.

Símbolo de entrada			
Estado	a	b	x
*estado*1	(*estado*2, *x*, *R*)	(*estado*1, *x*, *R*)	(*estado*3, *Y*, *R*)
*estado*2	(*estado*1, *x*, *R*)	(*estado*2, *x*, *R*)	(*estado*3, *N*, *R*)
*estado*3			

TABLA 1:
FUNCIÓN DE TRANSICIÓN DE LA MÁQUINA DE TURING DE ESTE EJEMPLO

Un cómputo de esta máquina con entrada *abba* comienza con la configuración inicial, seguida de una sucesión de configuraciones obtenidas de acuerdo a la tabla de transición. Un cómputo está dado a continuación:

$$a_1bba \mapsto xb_2ba \mapsto xxb_2a \mapsto xxxa_2 \mapsto xxxx_1 \mapsto xxxxYx_3$$

Usamos la notación *configuración* $\mapsto$ *configuración* para indicar que la configuración del lado derecho es obtenida de la configuración del lado izquierdo, de acuerdo con la tabla de transición.

El subíndice que aparece en cada configuración indica el estado de la máquina y está colocado sobre el símbolo donde la cabeza lectora/escritora está situada.

En este caso, la secuencia $xxxxYx_3$ [11] representa una configuración de paro, ya que ninguna regla de la tabla de transición puede ser utilizada. La salida de la máquina es Y. Nosotros afirmamos que esta máquina tiene salida Y si el número de *a*'s es par y *N* de otra manera. Note que una lista de todos los elementos incluídos en la descripción de la máquina incluye sólo un número finito de símbolos y que esta secuencia de símbolos de largo finito puede ser codificada como una secuencia finita de ceros y unos.[12]

Basado en la definición de máquina de cómputo, Turing definió como algorítmico cualquier proceso que pueda ser especificado usando una de sus máquinas con la propiedad de que no importa cual secuencia esté dada como entrada, la máquina siempre para y nos da una respuesta. Así, para resolver un problema en una manera algorítmica, creamos una máquina de Turing que recibe instancias del problema y produce como salidas soluciones a las correspondientes instancias, e.g., para resolver algorítmicamente el problema de sumar dos números,

11 Esta configuración indica que la máquina está en el *estado 3*, el contenido de la cinta (excluyendo los espacios en blanco) es *Y* y la cabeza lectora/escritora está colocada en el espacio en blanco que aparece después del símbolo *Y*.

12 Podemos escoger un esquema de codificación que asocia secuencias de cinco ceros y unos a cada uno de los símbolos de la descripción de acuerdo a una cierta regla, p.j., 00000 podría estar asociada al símbolo (, 00001 con *Y* y así sucesivamente.

escribimos una máquina de Turing que acepta dos números cualesquiera como entrada (una instancia del problema, e.g., 3, 7) y produce su suma (en este caso 10). La máquina del ejemplo dado anteriormente es un algoritmo para resolver el problema de reconocer las secuencias de a's y b's que incluyen un número par de a's.

Usando el hecho de que esta máquina podría ser representada por una sucesión finita de símbolos, Turing pasó a definir a la computadora en una forma que recuerda las afirmaciones autoreferenciales usadas por Gödel para mostrar la incompletitud de las matemáticas. Él definió a la computadora como una máquina de Turing, llamémosla U, cuya entrada consistía de una codificación para cualquier máquina de Turing, digamos M, seguida de una entrada para M, a la que llamaremos w. La máquina U lee la codificación de la máquina M y de su entrada w y entonces *simula* las acciones que M llevaría a cabo si tuviera a w como entrada. Él dio una descripción matemática de la máquina U y la llamó la Máquina Universal, nosotros la llamamos la Máquina Universal de Turing (MUT).

4. ¿Qué puede hacer la computadora?

Hasta aquí hemos discutido la definición de computadora. En esta sección, hablaremos acerca de la existencia de problemas no computables, mencionaremos un ejemplo de un problema no computable "natural" y daremos una afirmación que intente generalizar algunas de las propiedades y determinar el poder de la computadora. ¿Qué es lo que consideramos cuándo nos preguntamos acerca del poder de la computadora? ¿Nos preguntamos lo que es posible computacionalmente desde el punto de vista lógico? ¿Deseamos también considerar lo que es factible y lo que no lo es?

En justicia, parece que ambos, los aspectos lógicos y los prácticos, son importantes para establecer su poder. Nosotros los trataremos por separado.

Las ciencias de la computación estudian el poder de la computadora desde el punto de vista lógico en la teoría de la computabilidad y del práctico en la teoría de la complejidad computacional.

Alan Turing nos mostró que hay problemas que con respecto a su aparato de cómputo no son computables, lo que hizo que el problema de Hilbert concerniente a la decidibilidad de las matemáticas "colapsara por su propio peso". Las matemáticas son indecidibles.

Mientras que las proposiciones formuladas por Gödel para obtener sus resultados tienen, para algunos una forma artificiosa, los problemas no computables de Turing son muy naturales.

Para argumentar la no computabilidad de problemas necesitamos un poco de lógica y álgebra elemental. Primeramente discutiremos la existencia de tales problemas y después mostraremos la no decidibilidad de un problema natural y práctico.

Consideramos un problema como una función con dominio igual a todas las instancias de las que el problema consiste y cuyo codominio contiene las respuestas a las instancias del dominio. La solución de un problema puede verse como el cómputo de esta función. Por ejemplo, podemos considerar el problema de sumar dos números como una función cuyo dominio incluye todos los pares de números enteros (las instancias del problema) y un codominio que contiene todos los números que resultan de sumar dos enteros (las respuestas). Una máquina de Turing para resolver este problema recibiría como entrada dos números enteros (una instancia del problema, un elemento del dominio de su respectiva función) y daría como salida su suma (un elemento del codominio, la respuesta correspondiente a la entrada dada).

Ahora podemos dar un argumento de conteo sencillo para mostrar la existencia de funciones no computables, i.e., funciones que no pueden ser calculadas por ninguna máquina de Turing. El plan es el siguiente: nos restringimos a una clase de funciones simples, las contamos y encontramos que hay más de ellas que (posibles) máquinas de Turing. De aquí se sigue que deben

existir algunas funciones que no pueden ser calculadas por una máquina de Turing, ya que cada máquina computa una función.

Fijémonos en las funciones cuyo dominio son los números naturales y cuyo codominio incluye las palabras *sí o no*. Cada función asocia números pertenecientes a un conjunto particular de números naturales con la palabra *sí* y los números restantes que no están en el conjunto, con la palabra *no*. Por ejemplo, dos funciones en esta clase son: la función que le asocia la palabra *sí* a los números en el conjunto que consta solamente del 1 y la palabra no a aquellos en el conjunto de los números naturales excepto el 1; y la función que le asocia a los elementos en el conjunto de todos los números impares la palabra *sí* y a los elementos restantes, los números pares, la palabra *no*. ¿Cuántas de estas funciones hay? Primeramente observemos que existen tantas como posibles subconjuntos de números naturales, de los que se sabe que forman un conjunto no numerable.[13]

Ahora, ¿cuántas máquinas de Turing hay? Ya sabemos que existe una secuencia de largo finito que describe a cada máquina de Turing, entonces consideremos sus codificaciones en hileras de ceros y unos (justo como en nuestros días las computadoras codifican su información en sucesiones binarias). Entonces podemos interpretar estas secuencias como números naturales codificados en binario. Este punto de vista nos permite inferir que hay tantas máquinas de Turing como números naturales: un número numerable de ellas. Claramente no puede haber una correspondencia biyectiva entre las máquinas de Turing y las funciones descritas anteriormente, es decir un apareamiento entre las funciones y las máquinas de Turing que las computan. Luego existen algunas funciones que no pueden ser computadas por ninguna máquina de Turing.

Podríamos pensar que las funciones no computables no son de relevancia práctica, pero después de todo resulta que sí. Consideremos el siguiente problema (¿a qué función podría corresponder?). El *problema del paro:* Dada una máquina de Turing (o un programa), llamémosle M, y una secuencia de entrada, llamémosle w, ¿parará la máquina en algún momento después de que le dimos los datos de entrada w, o permanecerá trabajando para siempre (en un ciclo infinito)?

En estos tiempos de las computadoras omnipresentes, ¿qué pregunta podría ser más natural?

Siempre que usamos una computadora, estamos manejando una máquina universal de Turing con un programa (otra máquina de Turing) y un conjunto de datos. A nosotros no nos gustaría que la MUT nos mandara un mensaje como "¡Tendrá que esperar mucho tiempo por la respuesta, porque el cómputo que estamos simulando para usted contiene un ciclo infinito!", pero sí nos gustaría uno del tenor "¡Los resultados estarán listos en 5 minutos!". Resulta que el problema del paro no es computable. A continuación expondremos un argumento que prueba nuestra afirmación.[14]

Estructuraremos la prueba como sigue: primeramente suponemos que el problema es en realidad computable, esto es, que existe una máquina de Turing que computa su función asociada, llamémosla D, que será un "decididor ". La máquina D cuando le damos una entrada (o la codificación de una máquina), digamos M, y una secuencia de datos, digamos w, nos informa si la máquina M parará cuando termine de correr su programa con entrada w o no (note que los datos de entrada para la máquina D son de la forma $(!M!, w)$, donde $!M!$ denota una codificación de la máquina M). En segundo lugar, usando esta suposición, derivaremos una contradicción que nos indicará que nuestra suposición fue incorrecta, es decir que no hay ese decididor D para el problema de paro. Volveremos a encontrarnos en el proceso para llegar a

11 Recordemos que hay (infinitamente) muchos más números reales que números naturales, y que los naturales son numerables y los reales no numerables.

14 Lo que nosotros probamos es que su función asociada no es computable. Esta es la función que asocia a cada programa M y a cada secuencia de entrada para M, digamos w, con la palabra *sí*, si el programa M eventualmente para cuando le damos w como entrada, o con la palabra *no*, si M ejecuta un ciclo infinito cuando recibe la entrada w.

esta contradicción con una idea que nos recordará a un argumento de Gödel, es decir, otra vez emplearemos la idea de la autoreferencia.

Llevaremos a cabo la segunda idea como sigue: construiremos una máquina *P* (de problema) que recibe como entrada una codificación de cualquier MT (máquina de Turing), digamos *M*. La máquina *P* es similar a la máquina universal de Turing (*MUT*), ya que nos servirá para simular la máquina *D*, cuya existencia se supuso, con datos que consisten en una descripción de la máquina, !*M*!, y la secuencia de los datos para ella serán también !*M*!.

Si la simulación de la máquina *D* nos da un *sí*, la máquina *P* se cicla para siempre sin producir nunca una respuesta. Si *D* nos da un *no*, la máquina se para y nos entrega un *sí*. Podemos resumir la conducta de la máquina *P* como sigue:

- recibe como (datos de) entrada a la codificación de una máquina, digamos !*M*!,
- obtenemos como resultado un *sí* y se para, si el decidor *D* nos da un *no*, i.e., cuando *D* nos informa que la máquina *M* no parará de hacer los cómputos sobre una secuencia de datos que es una codificación de *M* misma,
- entra en un ciclo infinito, si el decidor *D* nos reporta un *sí* como resultado, i.e., cuando *D* nos informa que la máquina *M* parará (en algún momento) de hacer cómputos sobre la secuencia que es una codificación de ella misma.

Igual que con cualquier otra máquina de Turing, la máquina *P* tiene una representación en una secuencia de largo finito. Consideremos que pasa cuando la máquina recibe como entrada una codificación de sí misma, esto es, nosotros sustituimos !*P*! por !*M*! en la descripción previa[15]. Acerca de *P* podemos decir ahora lo siguiente:

- recibe como datos de entrada una codificación de la máquina *P*, esto es !*P*!,
- nos da un *sí* como salida y para, si el decidor *D* nos reporta como salida un *no*, i.e., cuando *D* nos informa que la máquina P no parará de hacer cómputos a partir de la secuencia de entrada que es una codificación de sí misma,
- entra en un ciclo infinito si el decidor *D* nos da un *sí*, i.e., *D* nos dice que la máquina *P* parará de hacer cómputos en algún momento si la entrada es una codificación de sí misma.

Claramente la conducta de la máquina *P* es contradictoria, ya que nos da un sí como salida y para, si entra en un ciclo infinito[16] (de acuerdo con el resultado del decidor) y ¡se cicla para siempre, si para!. En consecuencia, el supuesto decidor *D* no puede existir.[17]

Este resultado obtenido por Turing en 1936 nos da una visión panorámica acerca del poder de la computadora, o mejor dicho, de la debilidad de la computadora.[18]

Es un hecho que la computadora no puede decidir si un cómputo de un programa produce una salida en particular, o si para un programa dado y una salida dada hay una entrada que lo produzca.

Generalmente podemos decir que cualquier propiedad "no trivial" de los programas son indecidibles (no computables) por un computador.[19]

Concluimos esta sección replanteándonos una de las preguntas originales: ¿qué es lo que puede hacer una

15 No hay nada en nuestro planteamiento que nos lo impida. ¿No es cierto?

16 Esto es, la máquina no parará de hacer cómputos.

17 Esta es una adaptación de un argumento estándar para mostrar la no decibilidad del problema de paro, vea por ejemplo [9].

18 On Computable Numbers, with an application to the Entscheidungsproblem, Proc. Lond. Math. Soc. (2) 42 pp 230-265 (1936); correction ibid. 43, pp 544-546 (1937). Lo interesante del caso es que antes de que se construyera la primera computadora ya se había pensado en sus limitaciones, ya que Konrad Zuse construyó su primera computadora, la Z1, de 1936 a 1938 y la primera computadora hecha en los EUA se empezó a construir en 1938. No sabemos si Zuse estaba al tanto de los trabajos de Turing, lo que sí sabemos es que Turing se interesó en la construcción de una computadora ("Proposed Electronic Calculator", un original mecanografiado de principios de 1946 está en la oficina Public Record Office en el archivo DSIR 10/385).

19 La definición de propiedad no trivial es técnica y no añade mucho a la comprensión de nuestros argumentos. No estaremos muy errados si pensamos que el término quiere decir lo que nuestra intuición nos sugiere.

computadora? Aunque hay muchos resultados en la teoría de la computabilidad y (como veremos en la próxima sección) en la teoría de la complejidad computacional que establecen muchos hechos particulares a este respecto, quizás la respuesta más práctica está dada por la tesis Church-Turing: "Una computadora puede llevar a cabo cualquier proceso que pueda ser entendido como mecánico."

5. ¿Qué puede hacer realmente una computadora?

Hemos visto en la sección previa que hay muchos más problemas que soluciones computacionales y que algunos de los problemas sin solución computacional son de importancia práctica. Pero, ¿qué pasa con los problemas para los cuales hay en principio una solución algorítmica? ¿Pueden ser resueltos en términos prácticos, más concretamente, podemos obtener soluciones para todos los problemas solubles aún cuando impongamos restricciones de tiempo y espacio? En primer lugar, tenemos que decidir qué tipo de restricciones impondremos y después estableceremos si hay o no problemas cuyas soluciones no satisfacen estas restriccciones.

Centrémonos únicamente en el tiempo que una computadora necesita para producir la solución correspondiente a una instancia genérica del problema. Vagamente tenemos la idea de que no podemos esperar por la respuesta más que un lapso de tiempo igual al promedio de vida de un ser humano o el tiempo de vida correspondiente a 10 generaciones de seres humanos, cualquiera que sea su duración en años para el lector.

Indicaremos que hay un número infinito de problemas solubles que no satisfacen restricciones razonables de tiempo y que muchos de ellos son de importancia práctica, lo que los hace particularmente interesantes.

La teoría de la complejidad computacional intenta clasificar los problemas de acuerdo a qué tan "caras" son sus soluciones. Hay muchos tipos de criterios, pero por simplicidad consideremos únicamente el tiempo. Definiremos la viabilidad de una solución algorítmica para un problema con respecto a sus requerimientos de tiempo[20] como sigue: dada una solución algorítmica, mediremos sus requerimientos de tiempo en términos de (como función de) el número de caracteres que forman las instancias del problema. Llamaremos a los requerimientos en tiempo de un algoritmo su complejidad de tiempo. Por ejemplo, el algoritmo acostumbrado[21] para sumar dos números, digamos *m* y *r*, nos tomaría una cantidad proporcional al número de dígitos del número más largo de los dados en la entrada[22]. Las unidades que usamos para medir el tiempo pueden ser cualesquiera, digamos cualquier fracción de un segundo, un nanosegundo, un micosegundo o un milisegundo, etc. Lo que es relevante es la caracterización de los problemas con respecto a los requerimientos de tiempo necesarios para producir las soluciones.

Se puede mostrar que para casi cualquier complejidad de tiempo que podamos imaginar hay problemas cuyas soluciones algorítmicas tienen precisamente esos requerimientos de tiempo (Balcázar, Díaz y Gabarró 1988; Hopcroft y Ullman 1979).

La tabla 2, que tomamos de Garey y Johnson (1978), nos da una idea de la importancia de la complejidad de tiempo de los algoritmos. Esta tabla nos muestra, para diversas complejidades de tiempo (enlistadas en la primera columna), la cantidad de tiempo que tendríamos que esperar para recibir los datos de salida de un cómputo que tuviera secuencias de entrada de diferentes tamaños (las enlistadas en el primer renglón). Podemos suponer que los tiempos de espera se consideran con respecto a la computadora más rápida que, según los pronósticos, tendremos en el año 2000.

Parece obvio que un problema cuya mejor solución algorítmica tiene una complejidad en el tiempo de o ma-

20 Recordemos que una solución algorítmica para un problema es una máquina de Turing que acepta como entrada las instancias del problema y sus datos de salida son las respectivas respuestas a las instancias.

21 ¡El que usamos para sumar dos números sin usar la caculadora!

22 En este caso particular, ya que el tiempo necesitado por el algoritmo es proporcional a la longitud del número más largo, decimos que su complejidad en el tiempo es $c.n$, donde c es una constante y n es la longitud del número de entrada más largo.

yor no puede ser resuelto en la práctica, a pesar de ser soluble[23].

Nuestros lectores seguramente se están planteando la cuestión de sí hay problemas solubles que aparezcan en la práctica de una manera natural y cuya solución resulte ser intratable. La solución es *sí*, al menos en lo que respecta a nuestro conocimiento de las ciencias de la computación y las matemáticas. Sin embargo, nótese que algunos problemas, aunque no se ha probado todavía matemáticamente que son intratables, son considerados por los investigadores en el área como tales, pues se considera que hay suficiente "evidencia" para eso (Garey y Johnson 1979; Balcázar, Díaz y Gabarró 1988).

Tamaño n de la entrada						
Complejidad en el tiempo	10	20	30	40	50	60
n	.00001 segundo	.00002 segundos	.00003 segundos	.00004 segundos	.00005 segundos	.00006 segundos
n^2	.0001 segundo	.0004 segundos	.0009 segundos	.0016 segundos	.0025 segundos	.0036 segundos
n^3	.001 segundo	.008 segundos	.027 segundos	.064 segundos	.125 segundos	.216 segundos
n^5	.01 segundo	3.2 segundos	24.3 segundos	1.7 minutos	5.2 minutos	13.0 segundos
2^n	.001 segundo	1.0 segundo	17.9 minutos	12.7 días	35.7 años	366 siglos
3^n	.059 segundo	58 minutos	6.5 años	3855 siglos	2×10^8 siglos	1.3×10^{13} siglos

TABLA 2

TABLA DE COMPARACIÓN DE LOS TIEMPOS DE ESPERA PARA ALGUNAS COMPLEJIDADES DE TIEMPO DADAS CONTRA TAMAÑOS DE ENTRADAS

Hemos mencionado que los problemas más interesantes resultan ser problemas intratables. ¿Cuáles de esos problemas? No intentaremos enumerarlos todos, sólo diremos que ya conocemos algunos miles de ellos (Garey y Johnson 1979). Hablaremos en esta ocasión sobre tres de ellos:

- El primero de ellos es el *problema de la tautología,* este problema consiste en la determinación del valor de verdad de una sentencia dada con respecto a un sistema formal al menos lo suficientemente poderoso como para poder incluir el cálculo proposicional.[24] El hecho de que este problema sea intratable es lo que hace que los matemáticos tengan trabajo, porque la mayoría de los sistemas que los matemáticos usan son al menos tan poderosos como el cálculo proposicional.
- Otro problema es el que tiene que ver con la calendarización de tareas, por ejemplo, el problema de la calendarización de los exámenes finales en una universidad dada tal que sus estudiantes no tengan que presentar más de un examen diario dentro de un período de exámenes mínimo (o con otra condición parecida). Aún no se sabe si puede ser resuelto por algún algoritmo con una complejidad de tiempo menor que la muy burda de 2" pasos.[25]
- Otro problema práctico es el relativo a la "eficiencia" de la ida al supermercado: supongamos que cuando vamos al supermercado de compras deseamos maximizar la utilidad de las cosas que vamos a comprar con respecto al dinero, el espacio y cualquier otra cantidad de restricciones razonables y que tanto estas restricciones como la utilidad pueden ser expresada por medio de desigualdades lineales (la clase más simple de desigualdades) o ecuaciones. Este problema es llamado por los técnicos un problema de *programación lineal entera,* el cual también pertenece a la categoría de los dos previos.

En general, si imaginamos un problema "interesante", muy probablemente será intratable. Uno de los campos de la computación, la inteligencia artificial, se detuvo frente a la barrera de la intratabilidad. Mientras que muchas teorías se podrían usar en principio para la creación de una inteligencia artificial, las más prometedoras han re-

23 Note que estamos usando la palabra "soluble" en el sentido técnico de las matemáticas. En este caso queremos decir que la solución existe pero que emplearemos demasiado tiempo en calcularla.

24 Recordemos que si la potencia del sistema formal es tan grande que la aritmética puede ser expresada en él, este problema resulta ser indecidible, en cuyo caso no habría un algoritmo para solucionarlo.

25 Por eso no es de sorprenderse que haya tantos problemas con esto en MUN.

sultado ser intratables o, peor aún, no decidibles. Sin embargo, tenemos que decir que hay nuevos enfoques en la inteligencia artificial (que quizás deba ser llamada más propiamente vida artificial) que podrían horadar la barrera de la intratabilidad.

Mencionaremos además que los problemas intratables permanecen intratables incluso si los intentamos resolver con una computadora con cualquier número razonable pero variable de elementos de computación (una computadora paralela) que varien con respecto al tamaño de la entrada, i.e., al tamaño de las instancias del problema a resolver.

Hemos discutido ya diferentes aspectos del poder de la computadora. Hemos visto sus serias limitaciones lógicas y prácticas. Sin embargo para muchos de nosotros es un artefacto extremadamente útil y las implicaciones de este hecho podrían ser el tema de otro artículo.

Hemos afirmado que cualquier computadora es la materialización de una Máquina Universal de Turing. En la siguiente sección daremos una idea de cómo este concepto abstracto puede ser llevado a la práctica por medio de componentes electrónicos.

6. El aparato llamado computadora

Si hay un aparato (concreto) capaz de hacer lo que una Máquina Universal de Turing (MUT) hace... eso es una computadora. Recordemos que la MUT simula a otras máquinas para hacer cómputos a partir de los datos de entrada. Para lograrlo, primero fijemos un esquema de codificación que usaremos para especificar las entradas de la MUT y sus salidas. Sin pérdida de generalidad, podemos asumir que tal esquema de codificación produce secuencias de ceros y unos. Así, podemos ver que la tarea de la MUT es calcular una función con un dominio y un codominio que incluye sucesiones de ceros y unos. A esta clase de funciones las llamamos funciones booleanas, porque pueden ser expresadas usando variables booleanas (variables que sólo pueden tomar dos valores, a saber 0 y 1) y las tres operaciones bien conocidas de OR (o), AND (y) y NOT (no). Usando esta terminología podemos decir que la función de la MUT es la de calcular funciones booleanas. El próximo paso en la construcción de un aparato que funcione como una máquina MUT abstracta es asumir el hecho de que las funciones booleanas pueden ser consideradas como un modelo abstracto de un circuito con compuertas OR, AND y NOT. Las compuertas son aparatos físicos hechos de transistores que simulan las correspondientes operaciones booleanas. Esto es un modelo real muy simplificado de la MUT, ya que cualquier circuito tiene un número fijo de entradas y como tal sólo puede procesar entradas de cierta longitud, mientras que la longitud de las entradas de la MUT no está acotada, pero es siempre finita. Los ingenieros y científicos de la computación han encontrado algunas formas de soslayar este problema utilizando lo que popularmente se conoce como "memoria". Los detalles de la construcción de una computadora son mucho más complicados que esto y van más allá del objetivo de nuestra discusión. Únicamente mostraremos como se resuelve un pequeño problema que involucra funciones booleanas para transmitir un poco del "sabor" de este procedimiento. Supongamos que deseamos construir un circuito que simule la asociación entre entradas y salidas dada en la tabla 3.

El primer paso es separar la información dada en la tabla 3 en dos tablas, cada una de ellas relaciona a dos entradas con una salida como se muestra en las tablas 4 y 5. Siguiendo un sencillo procedimiento podemos extraer las expresiones booleanas que describen las relaciones mostradas en las tablas 4 y 5. Estas expresiones están dadas a continuación:

$s = (NOT(x) AND y) OR (x AND NOT(y))$ y

$c = (x AND y)$

Entradas		Salidas	
x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

TABLA 3: ASOCIACIÓN DE DOS ENTRADAS CON DOS SALIDAS

Entradas		Salida
x	*y*	*s*
0	0	0
0	1	1
1	0	1
1	1	0

TABLA 4: ASOCIACIÓN DE DOS ENTRADAS A LA PRIMERA SALIDA

Entradas		Salida
x	*y*	*c*
0	0	0
0	1	0
1	0	0
1	1	1

TABLA 5: ASOCIACIÓN DE DOS ENTRADAS A LA SEGUNDA SALIDA

Además, siguiendo un procedimiento sencillo podemos construir el circuito para computar la relación dada en la tabla 3 a partir de estas dos expresiones. El circuito resultante es mostrado en la figura 1.

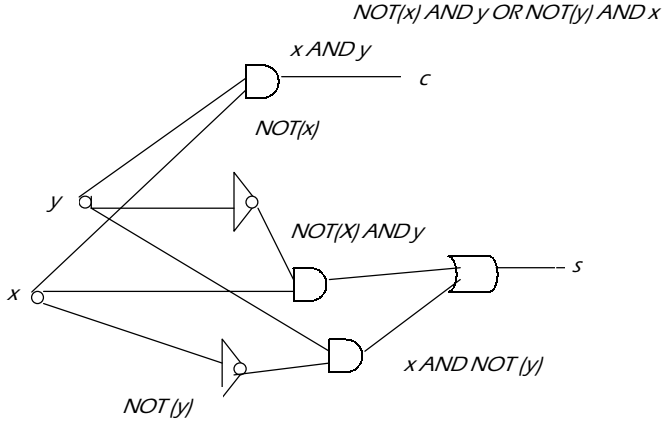

FIGURA1

CIRCUITO PARA IMPLEMENTAR LA SUMA DE DOS DÍGITOS BINARIOS

Si miramos con atención encontramos que el circuito mostrado en la figura 1 simula la suma de dos dígitos binarios, la suma está dada por *s* y lo que llevamos por *c*.

La "construcción concreta" de la MUT[26] es mucho más compleja pero los principios seguido por los productores de hardware son similares.

7. Software: la especificación de los algoritmos

Podemos pensar que la computadora como objeto físico, el hardware, es un modelo de la Máquina Universal de Turing y que los datos de entrada de esta máquina son pares de la forma (!M!, w), donde !M! es la codificación de una Máquina de Turing (que para), un algoritmo, y es la entrada para ese algoritmo. Los algoritmos pueden ser largos y complicados y su codificación puede requerir de un proceso muy detallado y complicado. Las primeras codificaciones para algoritmos que se usaron en la década de 1950 eran muy "primitivas", reflejaban las operaciones básicas de la computadora muy cuidadosamente. Cuando codificaciones nuevas y más expresivas fueron diseñadas, las especificaciones de los algoritmos se volvieron más sencillas. Estos esquemas de codificación más poderosos son llamados lenguajes (de programación). Así, usamos lenguajes de programación para codificar Máquinas de Turing.

Todos los lenguajes de programación permiten la codificación de los mismos algoritmos, la razón de la existencia de tantos de ellos son las ventajas que ofrecen para codificar diferentes tipos de algoritmos. Hasta ahora ningún lenguaje de programación ha podido ser definido para que permita la codificación de todos los algoritmos con la misma simplicidad, algo así como un lenguaje de programación universal (sencillo). En consecuencia, hay lenguajes de programación que facilitan el proceso de codificar algoritmos para el procesamiento de textos, mientras que otros facilitan la escritura de programas para cálculos numéricos. Puesto que hay muchos lenguajes de

26 Las computadoras actuales *no* corresponden a una MUT, sino a máquinas RAM universales. La máquina RAM es otro modelo de computación y como tal no es más poderoso que una MUT. Este modelo es el que se usa actualmente en las computadoras porque su operación es más eficiente que la de la MUT. No se usa para el análisis de computabilidad y complejidad porque su definición matemática no es tan simple y elegante como la de la máquina de Turing.

programación y el hardware de la máquina está dado, ¿cómo decodifica una máquina las codificaciones en los diferentes lenguajes de programación de, digamos, un algoritmo dado?

Para no tener que construir una computadora para cada lenguaje de programación, lo que sería una solución particularmente costosa y difícil, se crea un programa que traduce la codificación de un algoritmo que ya está codificado con un esquema de codificación en una codificación equivalente que la computadora pueda decodificar. Por lo tanto, hay un programa para cada lenguaje de programación (esquema de codificación). Estos programas son llamados compiladores y tienen, o deberían tener, la propiedad de que la traducción *preserva el significado,* i.e., lo que significa el programa original es lo que debe significar el programa en la codificación que el hardware puede decodificar. Esta clase de solución nos lleva a plantearnos la pregunta: ¿qué significa *preservación del significado?*

La ciencia de la computación ha tenido que crear formalismos para especificar el significado de las oraciones en un lenguaje. Hay muchos formalismos pero su discusión cae fuera del ámbito de este trabajo.

Podemos entender la *preservación del significado* con respecto a dos codificaciones de un algoritmo: dos codificaciones de un algoritmo tienen el mismo significado si, dada una entrada arbitraria, las salidas producidas por los dos algoritmos son las mismas cuando las interpretamos de acuerdo al significado de sus respectivos lenguajes.

El significado de una oración en un lenguaje puede ser dado de muchas formas, una de ellas es describir la conducta de la máquina subyacente[27] cuando se le presenta una oración en ese lenguaje. Cuando describimos el significado de las oraciones en un lenguaje en términos de la conducta de su máquina subyacente decimos que damos la "semántica operacional" del lenguaje. Quizás la semántica operacional de un lenguaje es la manera más intuitiva de especificar el significado de las sentencias en un lenguaje de programación, pero tal vez no es la más conveniente para todos los casos.

Una solución alternativa a la traduccción entre codificaciones diferentes es imaginar que hay una máquina que puede decodificar las oraciones en un lenguaje particular, digamos el lenguaje Scheme, y que los algoritmos están escritos para ser decodificados y llevados a cabo por la máquina del mismo nombre: la máquina virtual de Scheme. Ya que las máquinas virtuales no existen entonces se escribe un programa para simular esa máquina virtual en un lenguaje de programación que puede ser decodificado y ejecutado por el hardware existente.

Cuando nosotros deseamos correr un programa escrito en Scheme, solicitamos al hardware que ejecute el simulador que ejecuta (simula) programas (máquinas) escritas en Scheme.

Cuando empleamos un procesador de palabras (un programa, una máquina de Turing) le indicamos a la Máquina Universal de Turing (el hardware) que simule el procesador de palabras. El texto que nosotros escribimos es la entrada al procesador de palabras y el documento final es la salida de la simulación. El desarrollo de algoritmos y sus codificaciones es una tarea compleja y es la principal ocupación de los programadores.

Los usuarios de computadoras no necesitan saber cómo programar porque ellos generalmente usan productos desarrollados por programadores: el software. Frecuentemente los usuarios solamente necesitan familiarizarse con productos de software particulares (Máquinas de Turing), aprender como pedirle a la MUT que simule el producto de software que quieren usar y, finalmente, proveer los datos para la simulación.

8. Conclusiones

Hemos discutido la definición, construcción y la capacidad de la computadora. El lector puede tener la impre-

27 Implícito en esta afirmación está el hecho de que los lenguajes de computadora tienen máquinas que llevan a cabo las sentencias escritas en esos lenguajes. Estas máquinas pueden no existir en la realidad en cuyo caso son llamadas máquinas virtuales.

sión de que el tono general de nuestro discurso fue establecer las limitaciones de la computadora. Ninguna afirmación fue hecha acerca de lo que puede hacer, aparte de la afirmación general conocida como la Tesis Church-Turing. Creemos que la capacidad de la computadora es suficiente para modificar la manera cómo nuestra cultura maneja los problemas. Problemas monetarios, domésticos, científicos y académicos se resuelven ahora de una manera diferente usando la computadora. En un artículo posterior nos referiremos a algunas de las ideas que están revolucionando la manera en la que entendemos a la naturaleza y a nosotros mismos. Nosotros sostenemos que la influencia de la computadora es tan grande que está causando un cambio de paradigma en la ciencia y en un futuro cercano en las humanidades.

El lector puede notar que hay muy poca o ninguna mención de la tecnología de la computación. Fue una omisión deliberada. Creemos que a pesar de la gran inventiva, la astuta ingeniosidad y las grandes inversiones monetarias de los fabricantes de hardware, no hay ninguna diferencia sustancial o fundamental entre el primer computador que fue construido y los últimos modelos. El lector interesado puede consultar cualquier libro de divulgación que describa el funcionamiento interno de la computadora

Bibliografía

Balcázar, José Luis, Josep Díaz y Joaquín Gabarró

1988 *Structural Complexity I.* Springer Verlag,.

Dawkins, Richard

1986 *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe without Design.* Norton.

Dennet, Daniel C.

1996 *Darwin's Dangerous Idea: Evolution and the Meanings of Life.*Touchstone Books.

Garey, Michael R. y David S.

1979 Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Co., San Francisco.

Hodges, Andrew

1983. *Alan Turing: The Enigma.* Simon and Schuster.

Hofstadter, Douglas

1980 *Gödel, Escher Bach : An Eternal Golden Braid.* Vintage.

Hofstadter, Douglas

1982 (Tr. Mario Arnaldo Usabiaga Brandizzi.). *Gödel, Escher Bach: Una Eterna Trenza Dorada.* Conacyt.

Hopcroft, John E. y Jeffrey Ullman.

1979 *Introduction to Automata Theory, Languages and Computations.* Addison-Wesley.

Christos H.

1994 Papandimitriou *Computational Complexity.* Addison-Wesley.

Sipser, Michael

1996 *Introduction to the Theory of Computation.* PWS Publishing Company,

Williams, Michael R.

1985 *A History of Computing Technology.* Prentice Hall.

Una excelente página en la red para conocer más acerca de Alan Turing es:
www.turing.org.uk/turing/scrapbook/