COMPUTACIÓN Y PROGRAMACIÓN FUNCIONAL

CAMILO CHACÓN SARTORI

Marcombo

COMPUTACIÓN Y PROGRAMACIÓN FUNCIONAL

Introducción al cálculo lambda y la programación funcional usando Racket y Python

Camilo Chacón Sartori

COMPUTACIÓN Y PROGRAMACIÓN FUNCIONAL

Introducción al cálculo lambda y la programación funcional usando Racket y Python

Camilo Chacón Sartori

Computación y programación funcional

Primera edición, 2021

www.marcombo.com

Diseño de cubierta: ENEDENÚ DISEÑO GRÁFICO

Corrección: Manel Fernández y Haizea Beitia

Maquetación: D. Márquez

Directora de producción: M.ª Rosa Castillo

ISBN: 78-84-26732-84-2

Producción del ePub: booqlab

A mi padre y mi abuelo,
José Chacón y Carlos Sartori

ÍNDICE

PRÓLOGO

El cálculo automático es una invención con fecha antigua. Hace 2000 años ya se había construido el «mecanismo de Anticitera», un dispositivo que permitía calcular los eclipses lunares. Más tarde, en el siglo XVII, el matemático Wilhelm Schickard desarrolló la primera calculadora mecánica.

En el siglo XX aparecieron casi concurrentemente los modelos teóricos de la computación y los desarrollos tecnológicos asociados. Muchos de estos últimos estaban basados en la intuición; otros, fundamentados en bases formales. Aparecen los modelos de Church y Turing en el área de la teoría de la computación, y de Chomsky en el área de los lenguajes formales.

Los tres enfoques, complementados con las ideas y los estudios de muchos científicos y profesionales, fueron el punto de partida de lo que hoy denominamos «Ciencia de la Computación».

A la par de estos nuevos conocimientos, surgió la necesidad de expresarlos de manera inteligible para el ser humano, y procesables por un sistema automatizado. Esto dio origen a los lenguajes de programación, cada uno con sus enfoques y sus sabores sintácticos. Los procesos involucrados en el análisis de esos lenguajes se fundamentan en los aportes de Chomsky y Backus: las teorías de Chomsky (1956) relativas a los lenguajes formales y las gramáticas y los aportes de Backus relativos a cómo expresar las reglas gramaticales. Backus creó un lenguaje recursivo, que fue denominado «BNF» (Backus Normal Form, 1959) en su honor.

Los primeros lenguajes de programación surgieron para satisfacer diferentes necesidades: Fortran (1954), para el cálculo numérico; APL (1957) incorporó la computación funcional; Algol (1958) muestra el camino hacia la programación estructurada; Lisp (1958), basado en el cálculo lambda, derivó en el primer lenguaje para inteligencia artificial; Cobol (1959), para el procesamiento masivo de registros, y Simula (1962) introduce el concepto de «clase».

Sin temor a equivocarnos, podemos afirmar que la mayoría de los paradigmas computacionales actualmente en boga tienen su semilla en las décadas de los 50 y los 60: la programación orientada a objetos, la inteligencia artificial, la computación funcional, el cálculo lambda, la programación estocástica y tantas otras ideas. Las sucesivas encantaciones de dichas creaciones primigenias devienen, gracias a los nuevos aportes, interacciones y experiencias, en el mundo digital que vivimos hoy.

Los paradigmas reaparecen adaptados y evolucionados para enfrentar necesidades nuevas, tales como los crecientes volúmenes de información y los tamaños de los sistemas computacionales, que en muchos casos superan los millones de líneas de código. Así, por ejemplo, en su momento florece el concepto de programación estructurada como consecuencia del creciente tamaño de los programas y la necesidad de garantizar un código computacional libre de errores. También aparecen nuevos enfoques de administración de datos, con la creación de lenguajes especializados como lo fue SQL (1970), fundamentado en el cálculo relacional propuesto por C. J. Date.

Nacen lenguajes que favorecen la programación orientada a objetos basados en el concepto de clase ofrecido por el lenguaje Simula. Estos lenguajes evolucionaron de diversas formas, de ellas solo mencionaremos Python (1991) y Java (1995) dadas sus notorias diferencias conceptuales: Java exige clasificar los objetos durante la fase de compilación, mientras que Python delega la clasificación de los objetos en el momento de ejecución. El primero busca garantizar la correcta interacción entre objetos, mientras que el segundo busca

una forma más simple de especificación. En el segundo caso, se reduce la necesidad de tener un conocimiento detallado de los objetos utilizados, pero se aumenta la carga de trabajo durante la ejecución.

En 1994, junto al Internet abierto y público, aparece el procesamiento distribuido. Ahora sus componentes ya no forman parte de sistemas localizados, donde los diversos módulos que interactúan son conocidos, sino que deben interactuar con objetos y procedimientos remotos cuyo detalle se desconoce. Esto conlleva la necesidad de definir interfaces con contexto laxo para simplificar su interacción. Es aquí donde aparecen necesidades conceptuales y lingüísticas que deben ser provistas por nuevas herramientas que permitan la implementación de las interfaces necesarias.

Lo descrito presenta el entorno donde se ubica la computación funcional y el cálculo lambda contenidos en este libro.

El libro nos presenta un sólido análisis teórico y conceptual de los tópicos vertidos aquí, y describe detalladamente la manera en la que estas ideas se hacen disponibles en los lenguajes Haskell (1980), Python (1991) y Racket (1995). La lectura y el estudio detallado de su contenido proveerán al lector de los conocimientos necesarios que le permitirán comprender, resolver y extender los problemas asociados al desarrollo de programas computacionales conforme las tendencias actuales.

Quisiera terminar con un pensamiento que el matemático D.H. Lehmer escribió en 1966:

> Hay dos tipos de actividades en la investigación matemática: (a) la mejora de las carreteras entre las partes bien establecidas de las matemáticas y los puestos avanzados de su dominio, y (b) el establecimiento de nuevos puestos de avanzada.
>
> Tomando la segunda actividad primero, parece haber dos escuelas de pensamiento sobre la cuestión de cómo descubrir mejor nuevos puestos de avanzada. La escuela más popular de nuestros

días favorece la extensión de la prueba a situaciones más generales. Este procedimiento tiende a debilitar las hipótesis más que a fortalecer las conclusiones. Favorece la proliferación de teoremas de existencia y es psicológicamente reconfortante porque es menos probable que uno se encuentre con teoremas que no puede probar.

Bajo este régimen, las matemáticas se convertirían en un universo en expansión de generalidad y abstracción, extendiéndose sobre un paisaje multidimensional sin rasgos distintivos en el que cada piedra se convierte en una pepita de oro, por definición.[1]

Carlos Lauterbach R.
PhD en Ciencias de la Computación
UCLA, 1976.

---

1 Lehmer D.H. «Mechanized Mathematics», *Bulletin of the American Mathematical Society,* September 1966, Vol 72, pp 739-750.

ACERCA DEL LIBRO

La popularidad de la programación funcional ha crecido en los últimos años debido a sus principales ventajas a la hora de construir software: reducción de errores, manejo eficiente de datos en entornos concurrentes que deben escalar y un gran respaldo teórico. Sin embargo, muchos programadores fracasan en su intento de adentrase en ella por ir directamente a aprenderla usando un lenguaje de programación (tecnología), omitiendo así la teoría y el contexto histórico que le dio origen.

Por eso este libro presenta una tesis muy simple: «la programación funcional no puede reducirse solo al uso de la tecnología».

Para sustentarla, ofrecemos una introducción a lo qué es la computación y la programación, en pos de delimitar su campo de acción. En segundo lugar, presentamos el cálculo lambda, que es el modelo de computación que influenció a la programación funcional en los años en los que ni siquiera existían los lenguajes de programación ni mucho menos los ordenadores digitales. Y para concluir, usamos los lenguajes de programación Racket y Python para enseñar las diversas características de la programación funcional, sus fortalezas y debilidades y como ellas pueden combinarse con otros paradigmas.

Este libro está dividido en tres partes fundamentales:

I. **Introducción a la computación y la programación**. En los primeros tres capítulos abordaremos los fundamentos de la computación, la programación y los lenguajes de

programación. Incluso, y no menos importante, trataremos sus implicaciones filosóficas.

II. **Cálculo lambda**. Los capítulos del 4 al 11 serán una introducción a este modelo de computación, que es la base de la programación funcional. En ellos encontraremos diversos ejemplos para su mayor comprensión.
III. **Programación funcional**. En los capítulos restantes entraremos a la parte aplicada del libro, donde veremos cuáles son los conceptos principales de este paradigma usando lenguajes de programación.

Asimismo, este libro no renuncia a una visión teórica de la computación, ni mucho menos a la parte práctica. Se deduce así que la computación es una conversación entre estas dos cuestiones y que una no puede reducir o absorber a la otra.

Por otro lado, y no menos importante, todo el libro es un recorrido a través del cálculo lambda no tipado y, por consiguiente, significa que haremos uso de lenguajes de programación que son dinámicos y libres de tipos. No obstante, la importancia de los sistemas de tipos en la programación funcional es innegable y fundamental. Por ello, se dedica un capítulo para conocer el cálculo lambda tipado con una breve introducción a Haskell.

En resumen, usted aprenderá lo siguiente:

- Qué es la computación, la programación y los lenguajes de programación. Así tendrá una visión general del área que le permitirá entender el porqué de la existencia de diferentes paradigmas de programación y cómo ellos se entrelazan para construir software, con un énfasis en el modelo funcional.
- Los fundamentos que subyacen a la programación funcional, a saber: el cálculo lambda.
- Cómo aplicar estos fundamentos en un lenguaje de programación de origen funcional como lo es Racket, y en otro de uso masivo, como Python.

- A diseñar y construir un pequeño lenguaje de programación usando el enfoque funcional.

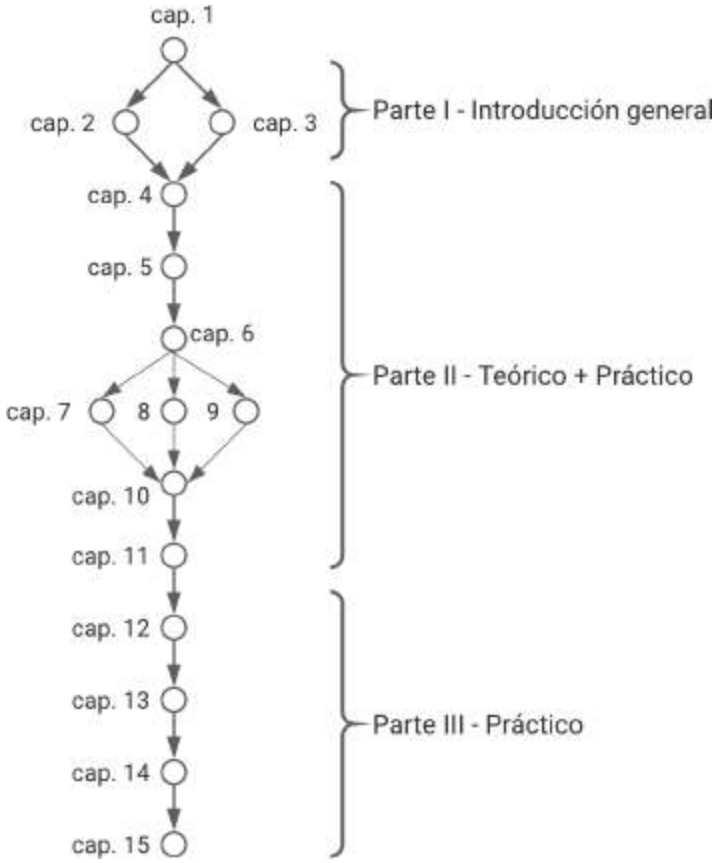
Al final del libro se incorpora una lista de lecturas recomendadas y un glosario de términos que sirve de soporte y ampliación a lo exhibido en esta obra.

A quién va dirigido el libro

Es para cualquier persona que tenga algún conocimiento básico en programación. Este libro incorpora temas que no han sido tratados en la literatura disponible en español de manera conjunta, por ende, puede ser interesante para todo aquel que quiera incorporar estos conocimientos a su ambiente laboral, ya sea un programador o programadora con varios años de experiencia o que recién este dando sus primeros pasos en el desarrollo de software.

Será de primordial interés para una persona que quiera comprender la programación funcional desde un contexto histórico y teórico antes de enfocarse en la tecnología. Alguien a quien le gusta la computación independientemente de la herramienta de moda. Igualmente, para esa persona que no desprecia la filosofía, la teoría, ni los múltiples paradigmas de la programación y, por ello, mantiene su curiosidad en el transcurso de la vida y suele evadir centrarse en una sola herramienta sin un contexto. En resumen, es para quien ama aprender.

Un camino visual de cómo leer el libro se ve en la siguiente figura:



Cada nodo del grafo es un capítulo del libro y las flechas (aristas) representan la prioridad de lectura. Por ejemplo, para leer el capítulo 4 podría leer los capítulos 1 y 2 o 1 y 3. O si usted prefiere, puede leer los capítulos 1, 2 y 3. Nada lo imposibilita. Asimismo, se añade a la derecha el enfoque de cada parte del libro.

Notas

De principio a fin de este libro, aparecen contenidos en un cuadro como este. Se trata de información extra, aclaraciones y comentarios que son relevantes como complemento a lo que se está tratando en dicho apartado. Por lo cual, le rogamos no omitirlo.

Ejercicios

Al final de cada capítulo de las partes II y III se encontrará con una sección de ejercicios. Cada uno de ellos tiene asociado el signo «*» que representa la dificultad del problema:

- (*) Fácil
- (**) Intermedio
- (***) Difícil

Cualquier otro signo indica que el autor de este libro le está proponiendo un desafío intelectual o, quizá, una broma.

Material

Toda la información está disponible en:

www.camilochacon.com

Ahí se encuentra el enlace al repositorio de GitHub que contiene lo siguiente:

- Cada código utilizado en la parte III.
- Cada figura en una mayor resolución y a color.
- Enlace para instalar Racket, Python y Haskell. (Usamos las últimas versiones de cada uno.)
- Enlace al intérprete web. Para este libro se ha creado un entorno que permite evaluar expresiones del cálculo lambda, por ello, usted podrá ejecutar cada ejemplo de la parte II y saber y verificar si ha podido resolver exitosamente cada ejercicio. Será su asistente en el aprendizaje.

PARTE I

INTRODUCCIÓN A LA COMPUTACIÓN Y LA PROGRAMACIÓN

1. **¿Qué es la computación?**
2. **¿Qué es la programación?**
3. **Lenguajes de programación**

La computación es esencial en nuestra vida. Y la programación la hace posible. Ambas se combinan a través de los lenguajes de programación, que son piezas vitales para entender su rol, contexto y relevancia.

Capítulo 1
¿QUÉ ES LA COMPUTACIÓN?

> Si alguien me preguntara qué consejo le daría a un joven [...] Creo que una de las cosas que le diría es que no porque algo esté de moda significa que sea bueno. Probablemente iría más allá: si encuentro demasiada gente adoptando una cierta idea, probablemente pensaría que está mal o, ya sabes, si mi trabajo se hubiera vuelto demasiado popular, probablemente pensaría que tengo que cambiar.
>
> Donald Knuth

Esta pregunta puede parecer simple en su formulación, pero nos puede llevar a confines que van desde aspectos técnicos hasta filosóficos. A su vez, la misma se ha vuelto relevante con el paso de los años y tuvo su mayor auge a comienzo del siglo XX, en particular, hacia la década de los treinta, cuando un grupo de científicos, matemáticos y lógicos buscaban comprender qué era la computación como tal, ya sea desde un punto de vista teórico o práctico.

Así como también sus posibilidades y limitaciones para tratar con diversos problemas. De esta manera, y producto de sus esfuerzos, fue como nació el concepto de «modelos de computación». Este concepto apareció antes de la existencia del primer ordenador digital (de nombre ENIAC, construido durante la segunda guerra mundial [1943]).

Pero ¿qué es un modelo de computación? Esta pregunta usa dos palabras relevantes: «modelo» y «computación». Primero, un «modelo» es una representación del funcionamiento de algo en particular (ya sea abstracto o concreto) que puede ser reproducible. Por otra parte, «computación» refiere a la capacidad de hacer tareas repetitivas que se puedan automatizar sin la intervención humana, de lo cual se desprenden dos cuestiones: (1) la acción de hacer estas tareas se llama **computar** y (2) si es posible realizar dichas tareas, entonces se suele decir que son **computables**.

Así, un «modelo de computación» es una representación abstracta que representa qué tareas son computables o no a través de computar operaciones. Dado que estos modelos son abstractos, podemos decir que un modelo de computación es equivalente a un modelo matemático. Con la salvedad de que el primero se enfoca en el contexto de la computación y hace uso de sus ventajas y desventajas.

Por ello la computación es en sí —al menos en su origen— es un área de las matemáticas. Por lo tanto, la computación es un conjunto de modelos de computación.

Sin embargo, la computación misma ha sufrido cambios en las últimas décadas, y decir que solo se trata de un conjunto de modelos de computación parece caer en un reduccionismo teórico. Más bien, tomaremos partido por ampliar esta definición a la siguiente: la computación es un conjunto de modelos de computación que se pueden expresar a través de artefactos abstractos (software) y concretos (hardware), mediante el uso de una metodología científica o ingenieril, teórica o práctica.

Objetivos de este capítulo:

- Conocer los principales modelos de computación.
- Conocer qué es la tesis de Church-Turing y cuáles son sus implicaciones en la computación.
- Comprender qué es la filosofía de la ciencia de la computación y cómo nos puede ayudar a ser mejores conocedores de

nuestra propia área.

Nota:

Aún hoy, existe un debate sobre la naturaleza de la computación. ¿Es una ciencia? ¿Una ingeniería? ¿O algo totalmente nuevo? Son preguntas difíciles de responder que han suscitado libros, artículos y amplios debates en las últimas décadas. Pero tomaremos una postura. Creemos, como dicen Peter J. Denning y Peter A. Freeman en su interesante artículo «Computing's Paradigm» («Paradigma de la computación») de 2009, que la computación es un «nuevo paradigma» que combina cuestiones de diversas disciplinas del saber: ciencias, ingeniería y matemáticas. Además, ellos dicen que la computación no trata sobre ordenadores, sino más bien sobre el procesamiento de la información, y los ordenadores son máquinas que implementan esos procesos.

Y no faltan motivos para creerlo, ya que la computación puede ser una herramienta muy útil para una innumerable cantidad de áreas, desde humanidades hasta ciencias, aparte de estudiarse a sí misma (procesos computacionales y sus límites teóricos). Algo que no se compara con nada hasta la fecha. Por tanto, deberíamos tratarla como algo nuevo, en desarrollo y en búsqueda de su propio significado.

Este libro no intentará entrar en ese debate. Pero daremos una introducción clásica y teórica de lo que es la computación, con algunas pinceladas filosóficas. Para más información le recomendamos leer el libro Computational thinking *(«Pensamiento computacional») de Peter J. Denning y Matti Tedre.*

1.1 MODELOS DE COMPUTACIÓN

Un modelo de computación es un modelo matemático, es decir, un modelo formal con una sintaxis definida, no ambigua, y que nos permite representar un proceso computable, al cual conocemos como **algoritmo**.

Problema de decisión

¿De dónde nace el concepto de modelos de computación? En primer lugar, debemos volver al principio del siglo XX y fijarnos en una persona que es relevante en la historia de las matemáticas: David Hilbert (véase figura 1.1). Fue un eminente matemático que a comienzos del siglo XX dio una famosa conferencia[2] en la que presentó veintitrés problemas matemáticos que serían importantes e influyentes para el desarrollo de las matemáticas y, por eso mismo, él esperaba que se encontrara una solución para cada uno de ellos. (Hasta el día de hoy hay muchos de ellos que siguen sin solución.) Luego, en 1928, junto a William Ackermann (su alumno de doctorado), publicó un libro titulado *Grundzüge der theoretischen Logik* («Fundamentos de la lógica teórica») donde postularon lo que se conoce como **problema de decisión** (originalmente conocido como *Entscheidungsproblem*, en alemán), el cual se define como sigue:

> Encontrar un procedimiento *P* que **siempre** encuentre la solución a un problema *A* según sus premisas y conclusiones.

A este «procedimiento» se le conoce, hoy en día, como «algoritmo». Por ejemplo, este problema plantea la pregunta de si es posible encontrar un algoritmo que siempre dé una respuesta («sí» o «no») a preguntas tales como: «dado un número natural, ¿es posible saber si es miembro de un conjunto?» o «dado un número natural, ¿puede responder si es un número primo o no?». Esto lleva a la siguiente cuestión, un problema de decisión es un problema de **decidibilidad** que busca responder a la pregunta de si existe un método efectivo (algoritmo) que demuestre si un objeto matemático (por ejemplo, un número) es miembro de un conjunto.

Un problema que demostró que no es posible encontrar un algoritmo para resolver un problema de decisión, es el conocido **problema de la parada** (*halting problem* en inglés). Este se puede definir de la siguiente forma:

> Si existe un programa *P* que recibe como argumento un programa *K* con datos de entrada *X*; *P* debe devolver 1 si *K* con la entrada *X* termina en un número finito de pasos, mientras que devuelve un 0 si *K* con *X* cae en un bucle infinito.

Esto desembocó, casi en paralelo y de manera independiente, en que Alan Turing, con su máquina de Turing,[3] y Alonzo Church, con el cálculo lambda, probaran que no existe un algoritmo para resolver el problema de la parada para cualquier valor de entrada (es decir, es «indecidible»), derrumbando, así, el sueño de Hilbert.

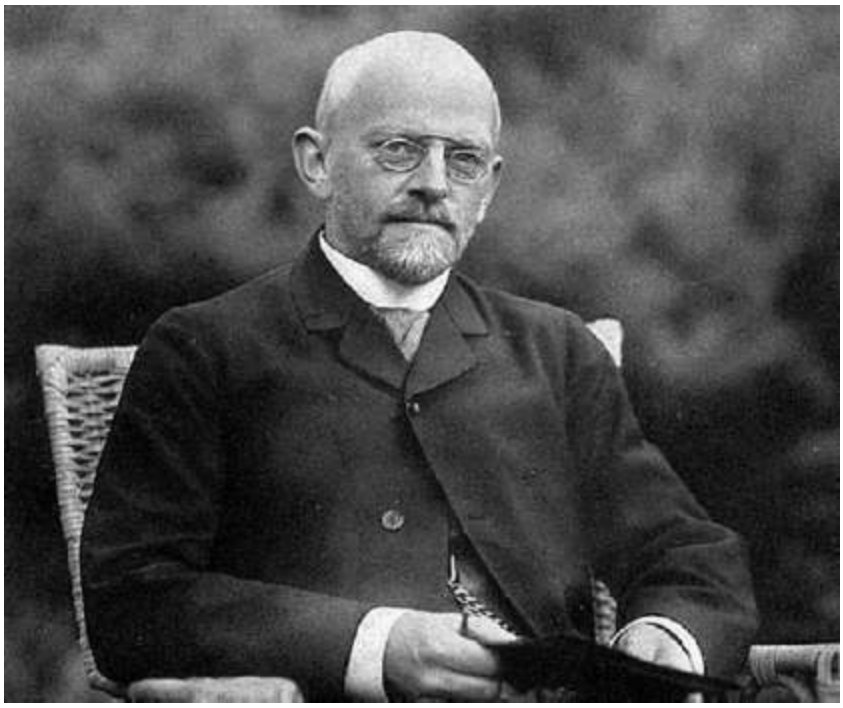


Figura 1.1 David Hilbert.

1.1.1 Máquina de Turing

En 1936, Alan Turing formularía en su artículo «On Computable Numbers, with an Application to the Entscheidungsproblem» («Sobre los números computables, con una aplicación al problema de la decisión») la prueba de que no es posible dar respuesta al problema de decisión.

En este trabajo Turing presentaría lo que hoy en día llamamos **máquina de Turing**. Una abstracción matemática que representa la computación o, para ser más precisos, lo que es computable o no.

Michael Sipser, importante teórico de la computación diría en su libro *Introduction to the Theory of Computation* («Introducción a la teoría de la computación») lo siguiente:

> Una máquina de Turing puede hacer todo lo que un ordenador de propósito general puede hacer. Sin embargo, hay algunos problemas que ni la máquina de Turing puede resolver. En concreto, estos problemas van más allá de los límites teóricos de la computación. (Sipser, 1997)

Esto quiere decir que, de hecho, una máquina de Turing puede hacer todo lo que un ordenador digital puede hacer (teóricamente) y que, a su vez, posee las mismas limitaciones. Por ello, queda claro que existe una influencia en su nivel más subyacente y esencial. Los ordenadores digitales que usamos en la actualidad corresponden a la arquitectura propuesta por John von Neumann en su borrador sobre EDVAC, en el que presenta la organización y el mecanismo de cómo debería funcionar este. Pero ya volveremos a esto.

Algunas características de una máquina de Turing son las que se presentan a continuación:

- Existe una cinta infinita (que puede entenderse como una memoria).
- Existe un cabezal que puede leer y escribir sobre cada celda de la cinta.
- El cabezal se puede mover de izquierda a derecha o viceversa.
- Existe una tabla de instrucciones; haciendo uso de ella, la máquina sabe qué valores puede reemplazar en cada operación y también cuándo detenerse.

Intuitivamente podría ser vista como se muestra en la figura 1.2:

Figura 1.2 Una representación visual de una máquina de Turing, donde el cabezal se posiciona en el símbolo 0. El símbolo al final (derecha) «#» significa que el procedimiento se va a parar (comúnmente conocido como *halt*, en inglés).

Ejemplo

Imaginemos que necesitamos realizar una operación muy simple: reemplazar cada aparición de cierto carácter en una secuencia de texto dada, por ejemplo, pasar de «00011010» a «11111111», o sea, reemplazar el carácter 0 por 1. Para ello, es necesario crear una máquina de Turing.

Para esto necesitamos una tabla de instrucciones que contenga la mecánica de cómo se realizará cada cambio de estado:

Estado actual	Símbolo de entrada	Siguiente estado	Símbolo de salida	Movimiento del cabezal
E_0	0	E_0	1	Derecha
E_0	1	E_0	1	Derecha
E_0	#	H (Se detiene)	-	-

Tabla 1.1 Tabla de instrucciones de lo que debe realizar la máquina.

Considere el siguiente texto de entrada:

00011010#

La secuencia comenzaría de izquierda a derecha. Se agrega el símbolo «*» para indicar la posición del cabezal junto al carácter

actual, que estará a la izquierda. Así pues, los cambios de estado serían los siguientes:

1. 0*0011010#
2. 10*011010#
3. 110*11010#
4. 1111*1010#
5. 11111*010#
6. 111111*10#
7. 1111111*0#
8. 11111111*#
9. 11111111#*
10. 11111111

En el primer paso, por ejemplo, el cabezal encuentra el carácter «0». Entonces, si vemos la tabla (primera fila), cuando el estado actual es E_0 y tiene el símbolo de entrada «0», se mantiene en el estado E_0 y lo cambia por el símbolo de salida «1», y el cabezal se mueve a la derecha. Esto ocurre sucesivamente hasta el tercer paso.

Ahora bien, si vamos al cuarto paso, que es donde encontramos un «1» en el cabezal, esto nos lleva a la segunda fila de la tabla. Si el estado actual es E_0 y el símbolo de entrada es «1», entonces se mantienen el estado actual y el valor «1», y se mueve el cabezal a la derecha.

El proceso sigue ocurriendo hasta que se encuentra el estado E_0 y el símbolo de entrada «#» (última fila de la tabla) devuelve un símbolo vacío «-» y detiene el procesamiento de la máquina de Turing (estado H). Por consiguiente, ya no hay ningún tipo de movimiento más por realizar. La computación se ha terminado.

Máquina de Turing universal

Una máquina de Turing universal U puede simular otra máquina de Turing T como una entrada. ¿Cómo lo logra? Leyendo (1) una descripción o tabla de instrucciones D_t de la máquina E_t a ser

simulada y (2) los valores de entrada de dicha máquina E_t. Pues bien, con esto se puede, con una sola máquina, lograr «ejecutar» cualquier otra máquina de Turing.

Esto lo representamos con la figura 1.3, donde una máquina de Turing universal *U* puede simular otra máquina de Turing E_t con (1) descripción, (2) contenido y (3) estados. Esto conlleva la idea de que una máquina de Turing universal puede simular cualquier otra máquina de Turing dándole la información requerida en otra «cinta».

Figura 1.3 Una máquina de Turing universal.

Una máquina de Turing universal, concretamente, según dice Martin Davis, inspiró a John von Neumann para crear la primera arquitectura de computadoras en 1945, en su documento «First Draft of a Report on EDVAC» («Primer borrador de un informe sobre EDVAC»). En este documento, von Neumann presenta las ventajas del sistema binario sobre el decimal para las operaciones aritméticas elementales (+, -, ×, ÷), y según sus propias palabras, esto se explica por lo siguiente:

> La principal virtud del sistema binario, en contraposición con el decimal, reside en la mayor simplicidad y velocidad con

> que se pueden ejecutar las operaciones elementales. (von Neumann, 1945)

Igualmente, se incorporan los registros especiales de memoria cuyo objetivo es, antes que nada, tener un conjunto de funciones ya definidas en memoria (por ejemplo: √, ||, log_{10}, log_2, *ln*, *sin*, etc.). Esto significa que no es necesario redefinir dichas funciones y, por ello, como es de suponer, el tiempo de cómputo se vería disminuido.

Por otro lado, el trabajo de Turing tuvo notables implicaciones en los fundamentos de la teoría de la complejidad computacional. Por su misma naturaleza, la capacidad explícita de secuencialidad de cada instrucción hace más simple poder clasificar problemas computacionales de acuerdo con su dificultad. Por ejemplo, las diversas clases de complejidad.

Antes de eso, primero debemos diferenciar dos cuestiones fundamentales: tiempo polinomial y tiempo exponencial. El tiempo polinomial, o mejor dicho, el tiempo polinomial **de un algoritmo**, es aquel en el que el tiempo de cómputo crece según la cantidad de datos de entrada *n*, los cuales no son exponenciales. Esto significa que la computación es más rápida. En la tabla 1.2 se presentan varios algoritmos[4] clásicos, que usan la notación Big O[5] (en el peor de los casos):

Tiempo polinomial		**Tiempo exponencial**	
Búsqueda lineal	$O(n)$	Knapsack 0/1	$O(2^n)$
Búsqueda binaria	$O(log\ n)$	Travelling salesman problem	$O(2^n)$
Quicksort	$O(n^2)$	Graph coloring	$O(2^n)$
Merge sort	$O(n\ log\ n)$	Hamylton cycle	$O(2^n)$

Tabla 1.2 Comparación de complejidad algorítmica entre tiempo polinomial y tiempo exponencial.

Cualquier algoritmo que tenga una complejidad elevada a la enésima potencia (como los que aparecen en la columna de la

derecha) es computacionalmente demasiado complejo de solucionar en un tiempo de cómputo óptimo (polinomial). Esto se traduce en que se tarda demasiado tiempo en terminar la computación.

Pasemos a ver las principales clases de complejidad:

- **P (*polynomial* en inglés)**. Es la clase de problema que es posible tratar, es decir, hay un algoritmo determinístico que lo resuelve de manera eficiente en un tiempo polinomial. Por ejemplo, los algoritmos de ordenamiento y de búsqueda de una subcadena dentro de un texto o del camino más corto entre dos nodos dentro de un grafo, entre otros.
- **NP (*non-deterministically polynomial* en inglés)**. La clase de problemas que pueden ser resueltos en un tiempo polinomial por un **algoritmo no determinístico**. (Esto último significa que son algoritmos que tienen un componente de selección aleatorio en su comportamiento, es decir, cada ejecución con los mismos argumentos no asegura el mismo valor de devolución. Por ejemplo, el algoritmo de colonia de hormigas, algoritmos genéticos, etc.) Los problemas NP son problemas de decisión. De esta clase nace la siguiente pregunta: ¿estos problemas pueden resolverse de manera determinista en un tiempo polinomial? Es decir, ¿es $P = NP$? Esto no ha sido probado y es uno de los problemas del siglo que aún sigue abierto. Ejemplos de problemas de esta clase: la factorización de números enteros y el isomorfismo de grafos.
- **NP-Completo**. Problemas más difíciles que los NP y, obviamente, que los P. Son problemas de decisión que se diferencian de los NP en lo siguiente: representan el conjunto de todos los problemas X en NP que se pueden reducir a cualquier otro problema NP en tiempo polinomial, es decir, intuitivamente lo entendemos así: si tenemos la solución para el problema X, entonces podríamos encontrar rápidamente la solución al problema Z. Ejemplos de problemas de esta clase son: 3-SAT, el problema del vendedor viajero, camino hamiltoniano, etc.