

Composición de una computadora y representación de la información

Aldo Sayeg Pasos Trejo

Algoritmos Computacionales
Facultad de Ciencias
Universidad Nacional Autónoma de México

14 de julio de 2020

Modelos teóricos: Máquinas de Turing

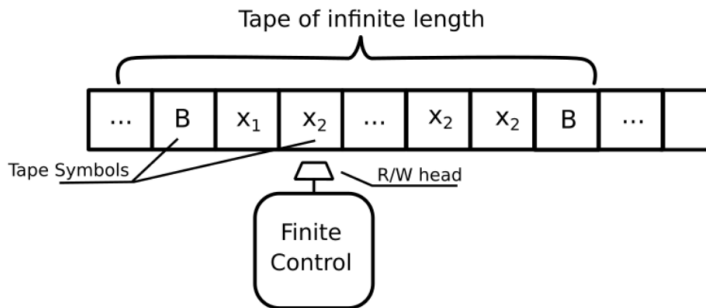


Figura 1.1: Máquina de Turing

Formalización de algoritmo: La tesis de Church-Turing

Tesis de Church-Turing

Un procedimiento es algorítmico si y solo si puede ser realizado por una máquina de Turing o es definible en cálculo lambda

OBS

Las máquinas de Turing y el cálculo lambda son equivalentes. Es decir, para cada máquina de Turing existe una función lambda que computa lo mismo dado el mismo input y viceversa.

Modelos de computabilidad

Turing-completez

Un autómatas o modelo de computación es *Turing-completo* si y solo si toda máquina de Turing es representable en dicho modelo.

¿Existen otros modelos Turing-completos además de las máquinas de Turing y el cálculo lambda? Por supuesto

- μ -funciones recursivas
- Lógica combinatoria
- Autómatas celulares
- Cualquier lenguaje de programación moderno: Python, Julia, C, Javascript, ...

¿Qué modelo utiliza una computadora moderna?

Las computadoras actuales utilizan un modelo de computabilidad llamado **máquina de acceso aleatorio**, que puede resumirse en una máquina de Turing que puede acceder a una celda en cualquier posición.

Es posible demostrar (de manera no trivial) la existencia de una equivalencia entre una máquina de Turing M arbitraria con una máquina de acceso aleatorio M^*

Random Access Machine (RAM)

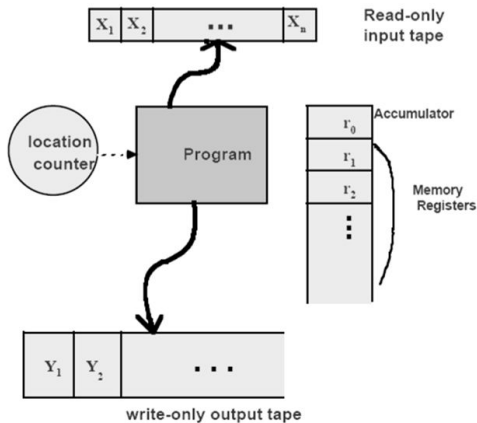


Figura 1.2: Máquina de acceso aleatorio

Diseño de computadoras: la arquitectura de Turing

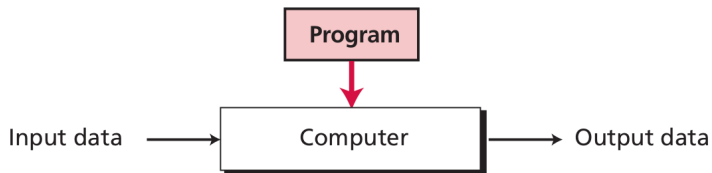


Figura 1.3: Arquitectura de Turing [1]

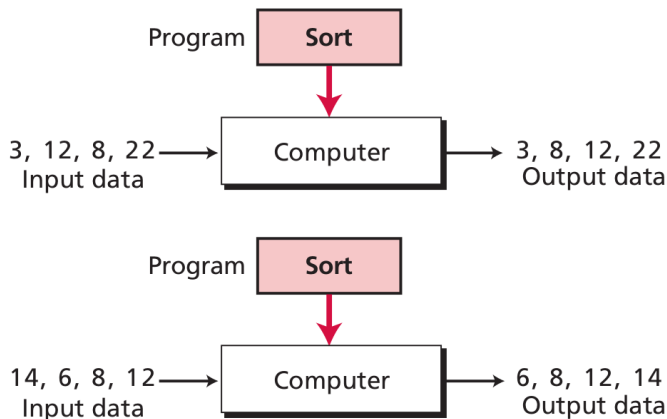


Figura 1.4: Computadora con programación fija para disintos inputs [1]

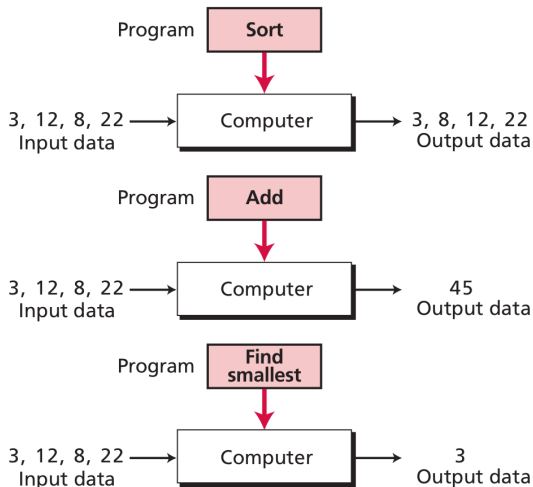


Figura 1.5: Computadora con distintas programaciones para input fijo [1]

Máquinas de Turing universales

Es posible construir una máquina de Turing M con una configuración tal que dentro su input venga especificada su configuración. Es decir, esta máquina puede cambiar su configuración para simular a cualquier máquina otra máquina M^* .

Es una máquina **universal**, es decir, puede **programarse** para hacer cualquier algoritmo.

Más aún, esta máquina almacenará en su memoria tanto su programación como su input y output.

Arquitectura de Von Neumann

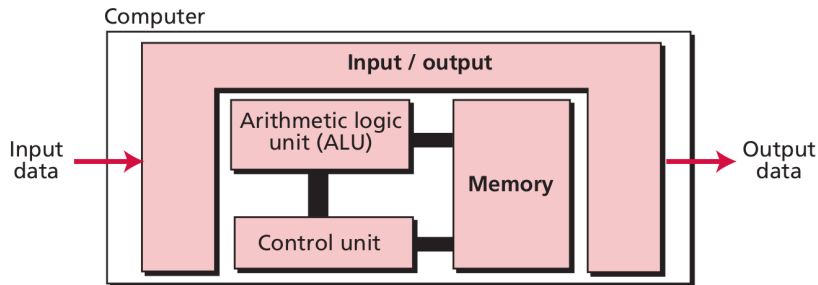


Figura 1.6: Arquitectura de Von Neumann [1]

Descripción de componentes

Memoria: lugar donde se almacena el programa y los datos usados por él, incluidos el input o el output.

Unidad Aritmética Lógica (ALU): lugar donde se realizan operaciones entre objetos almacenados en la memoria.

Unidad de control: coordina las operaciones de la memoria, ALU y el sistema de entrada y salida.

Sistema de entrada y salida: mecanismo que acepta datos como input y envía algunos como output.

Libertad en el modelo

La arquitectura de von Neumann solo es conceptual y nos permite definir como estará diseñada la memoria, como accederemos a ella, que operaciones puede realizar el ALU, como coordina las cosas la unidad de control, y muchas otras cosas.

Aunque no es exactamente el mismo modelo, la arquitectura de Von Neumann es similar a la arquitectura de las computadoras actuales

- Memoria: memoria RAM.
- ALU y unidad de Control: CPU
- Sistema de entrada y salida: periféricos y CPU.

¿Cómo realizar un cómputo en la arquitectura de von Neumann?

- 1 Especificar el programa, es decir, las operaciones a realizar en el input en términos de las operaciones posibles en el ALU.
- 2 Especificar el input en la memoria.

Especificación del input: representación de datos

Representar el input en la memoria depende de la estructura de la memoria. Por simplicidad y consistencia con el modelo de máquina de acceso aleatoria para computar, la mayoría de las memorias tienen una estructura **binaria**: muchos contenedores ordenados y distinguidos que pueden estar en dos estados.

Bit

Definimos un **bit** como un contenedor en una memoria tal que el contenedor solo puede tener dos valores distintos, que denotaremos 0 y 1.

Ya que la mayoría de operaciones que queremos realizar en una computadora no son sobre objetos binarios, si no sobre objetos más complejos (números enteros, números reales, palabras), debemos **codificar** esas estructuras en algo binario.

Representacion de números

Sistemas numéricos posicionales

Queremos utilizar sucesiones finitas de símbolos para representar un número arbitrario. Sean $b, k, l \in \mathbb{N}$, y, para todo $i \in \mathbb{N}$ tal que $-l \leq i \leq k$, sea $S_i \in \{0, \dots, b-1\}$.

Entonces la sucesión $\pm(S_k S_{k-1} \dots S_1 S_0 . S_{-1} \dots S_{-l+1} S_{-l})_b$ representa al número real $n \in \mathbb{R}$ tal que

$$\begin{aligned} n &= \pm(S_k \cdot b^k + S_{k-1} \cdot b^{k-1} + \dots + S_0 \cdot b^0 + \dots S_{-l+1} \cdot b^{-l+1} + S_{-l} \cdot b^{-l}) \\ &= \pm \sum_{j=-l}^k S_j \cdot b^j \end{aligned}$$

Al número b le llamamos **base**, a la subsucesión $S_k S_{k-1} \dots S_1 S_0$ la **parte entera**, $S_{-1} \dots S_{-l+1} S_{-l}$ la **parte fraccionaria** y al símbolo $+$, $-$ que lo antecede lo llamamos **signo**

Ejemplos

Notación decimal

$$b = 10$$

$$S_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$n = (132.59)_{10} = 132.59$$

Notación binaria

$$b = 2$$

$$S_i \in \{0, 1\}$$

$$n = -(100101.0110)_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3}$$

Notación hexadecimal

$$b = 16$$

$$S_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10(A), 11(B), 12(C), 13(D), 14(E), 15(F)\}$$

$$n = (A97B.0F8)_{16} = 10 \cdot 16^3 + 9 \cdot 16^2 + 7 \cdot 16^1 + 11 \cdot 16^0 \\ + 0 \cdot 16^{-1} + 15 \cdot 16^{-2} + 8 \cdot 16^{-2}$$

Representación de números enteros en notación binaria

Es claro que, si la memoria solo puede tener dos símbolos distintos, la notación más compatible es la notación binaria. Supongamos ahora que tenemos n bits de memoria los cuales queremos que representen a un entero.

¿Cuántos posibles valores podemos representar? Hay 2^n combinaciones distintas, por lo cual podemos representar 2^n números distintos.

Si los números enteros que nos interesan son todos positivos, no debemos preocuparnos por el signo y podemos representar cualquier entero m entre 1 y 2^n . Como es importante representar el cero, normalmente tomamos cualquier entero m con $0 \leq m \leq 2^n - 1$

Convenciones para representar enteros

Si el número tiene signo, es decir, puede ser positivo o negativo, debemos usar un bit para el signo y los $n - 1$ restantes para el valor absoluto del número. Esto implica que podemos guardar cualquier número entero m con valor absoluto entre 0 y $2^{n-1} - 1$, es decir, todos los enteros m con $-(2^{n-1} - 1) \leq m \leq 2^{n-1} - 1$.

Por cuestiones de construcción y diseño, no siempre es posible utilizar cantidades arbitrarias de bits para representar un entero, por lo que existen convenciones sobre cuantos bits se utilizan. Por lo general deben de ser múltiplos de 8 bits. A un conjunto de 8 bits le llamamos un **byte**. A una estructura computacional diseñada para almacenar un dato le llamamos **tipo**.

Bytes	Bits	Signo	Nombres	Límites
2	16	No	UInt16, unsigned short	0 – 65,535
2	16	Si	Int16, short	–32,767 – 32,767
4	32	No	UInt32, unsigned int	0 – $4.294 \cdot 10^9$
4	32	Si	Int32, int	$-2.147 \cdot 10^9$ – $2.147 \cdot 10^9$
8	64	No	UInt64, unsigned long	0 – $18.446 \cdot 10^{18}$
8	64	Si	Int64, long	$-9.223 \cdot 10^{18}$ – $9.223 \cdot 10^{18}$

Cuadro 1.1: Tipos para números enteros

Representación de valores binarios

Un bit puede codificar de manera trivial dos valores distintos, los cuales pueden representar los valores de verdad de una proposición (verdadero 1 y falso 0).

Los valores binarios pueden ser muy útiles para realizar operaciones con lógica por lo que a las computadoras les interesa. A estos valores de verdad se les asigna el tipo `bool` o `boolean`

Representación de números reales

En principio, todo número real tiene una cola infinita de dígitos decimales, por lo que no podemos representar una cantidad infinita de dígitos. Solo podemos representar números con una cantidad finita de decimales, es decir, un conjunto todavía más pequeño que los racionales. Intuitivamente, podemos utilizar un sistema numérico posicional binario para representar dichos números.

Esto **NO** es lo que normalmente se hace. Los sistemas posicionales pierden información sobre los números.

Ejemplo

Usando notación binaria, si tenemos 4 bits para la parte entera y 3 para la parte decimal, podemos representar un número como $(11.01)_2$ de manera precisa como $(0011.010)_2$, pero a la hora de querer representar el número $(0.00011)_2$ este se volará $(0000.000)_2$, lo cual pierde toda la información sobre el número original.

Sistemas de punto flotante

Es preferible utilizar un sistema en el cual la posición del punto decimal no esté fija, es decir, podemos usar los bits para representar dígitos en posiciones arbitrariamente lejanas del punto decimal. A esta clase de sistemas le llamamos sistemas de punto flotante. ¿Como son estos sistemas?

Ejemplo no binario: notación científica

En la notación científica, podemos representar un número como 0.0026 como $2.6 \cdot 10^{-3}$, al igual que 59000.0 como $5.9 \cdot 10^4$

Ejemplo binario

Para representar un número como $(0.00011)_2$, podemos representarlo como $(1.1)_2 \cdot 2^{-4}$. También $(11.01)_2 = (1.101)_2 \cdot 2^1$.

Números flotantes

Esto implica que, para una base fija $b \in \mathbb{N}$ y dos enteros $x = \pm(S_n \dots S_0)_{10}$ y $y = \pm(T_m \dots T_0)_{10}$, podemos definir un nuevo número $z = x \cdot b^y$. Al número x se le llama **mantissa** y a y se le llama **exponente**

Números flotantes

Bytes	Bits	Bits Exponente	Bits Mantissa	Nombres	Límites
2	16	5	11	Float16, half	$-6.5 \cdot 10^4 - 6.5 \cdot 10^4$
4	32	8	24	Float32, float	$-3.4 \cdot 10^{38} - 3.4 \cdot 10^{38}$
8	64	11	53	Float64, double	$-1.7 \cdot 10^{308} - 1.7 \cdot 10^{308}$

Cuadro 1.2: Tipos para números flotantes

¿Cualquier número con un número finito de dígitos es representable en punto flotante?

Notemos que, para un tipo de número flotante, hay una cantidad finita de números que podemos representar con ese tipo, por lo que no es posible representar todos los números. ¿Qué sucede si queremos representar a uno que no está ahí? La computadora lo **redondea** al más cercano.

Esto provoca que puedan existir errores en la representación de valores y, si además operamos con un número redondeado, esos errores se pueden hacer más grandes. La rama que estudia estas limitaciones de los sistemas se llama **aritmética de punto flotante**

Representación de símbolos

Si ahora queremos representar un símbolo tomado de un conjunto finito S de posibles símbolos Γ , la manera más simple es asignar a todo elemento $t \in \Gamma$ un número entero y usar dicho número para representarlo en la memoria.

A este proceso se le llama **codificación**. Existen diversos estándares de codificación de símbolos.

Codificación ASCII

La codificación ASCII (American Standard Code for Information Interchange) fue una de las primeras utilizadas en computación. Se usa para codificar $128 = 2^7$ símbolos distintos utilizados en el idioma inglés. Para representar cada símbolo, se utiliza entonces un entero de 7 bits.

A los símbolos codificados por ASCII la computadora les reserva el tipo `char`

0	<u>NUL</u>	16	<u>DLE</u>	32	<u>SP</u>	48	0	64	@	80	P	96	`	112	p
1	<u>SOH</u>	17	<u>DC1</u>	33	!	49	1	65	A	81	Q	97	a	113	q
2	<u>STX</u>	18	<u>DC2</u>	34	"	50	2	66	B	82	R	98	b	114	r
3	<u>ETX</u>	19	<u>DC3</u>	35	#	51	3	67	C	83	S	99	c	115	s
4	<u>EOT</u>	20	<u>DC4</u>	36	\$	52	4	68	D	84	T	100	d	116	t
5	<u>ENQ</u>	21	<u>NAK</u>	37	%	53	5	69	E	85	U	101	e	117	u
6	<u>ACK</u>	22	<u>SYN</u>	38	&	54	6	70	F	86	V	102	f	118	v
7	<u>BEL</u>	23	<u>ETB</u>	39	'	55	7	71	G	87	W	103	g	119	w
8	<u>BS</u>	24	<u>CAN</u>	40	(56	8	72	H	88	X	104	h	120	x
9	<u>HT</u>	25	<u>EM</u>	41)	57	9	73	I	89	Y	105	i	121	y
10	<u>LF</u>	26	<u>SUB</u>	42	*	58	:	74	J	90	Z	106	j	122	z
11	<u>VT</u>	27	<u>ESC</u>	43	+	59	;	75	K	91	[107	k	123	{
12	<u>FF</u>	28	<u>FS</u>	44	,	60	<	76	L	92	\	108	l	124	
13	<u>CR</u>	29	<u>GS</u>	45	-	61	=	77	M	93]	109	m	125	}
14	<u>SO</u>	30	<u>RS</u>	46	.	62	>	78	N	94	^	110	n	126	~
15	<u>SI</u>	31	<u>US</u>	47	/	63	?	79	O	95	_	111	o	127	<u>DEL</u>

Figura 1.7: Codificación ASCII

Codificación Unicode

La codificación Unicode pretende ser más universal al incluir símbolos de todos los idiomas utilizados en el mundo.

En su modalidad UTF-8, es compresible y dinámico: utiliza entre 1 y 4 bytes para representar cualquier símbolo, usando menos bytes entre más común es el símbolo (los primeros 128 símbolos corresponden a los símbolos de ASCII).



































Count	Code	Browser	B&W*	Apple	Andr	Twit	Wind	GMail
1	U+1F600							
2	U+1F601							
3	U+1F602							
4	U+1F603							
5	U+1F604							
6	U+1F605							

Figura 1.8: Codificación Unicode

Representación de palabras

Para representar una palabra, podemos simplemente verla como una sucesión finita de símbolos y representarla utilizando

A las palabras o sucesiones finitas de símbolos se les asigna el tipo `string`

Sucesiones finitas: arreglos

Si quisiera no almacenar no solo un número entero, real, símbolo o palabra, ¿Cómo puedo agrupar en la computadora una sucesión finita de estos objetos?

Podemos definir sucesiones finitas de enteros, reales, símbolos o palabras. A una sucesión finita de objetos del mismo tipo se le llama un **arreglo** o “**array**”.


También podemos tener una sucesión finita de objetos de distinto tipo. A esas sucesiones normalmente se les llama **listas**.

Ejemplos

- $A = [1, 2, 3, 5, 7, 8]$ es un arreglo de enteros.
- $B = [c, a, !]$ es un arreglo de símbolos.
- $C = [c, 5, 10.695]$ es una lista.

Notemos que todo arreglo es una lista pero no toda lista es un arreglo. Normalmente $A[i]$ denota al i -ésimo elemento de una lista o arreglo. Dependiendo del contexto, los elementos se pueden empezar a contar desde 0 o 1. Es decir, $C[2] = 5$ o $C[2] = 10.695$

Referencias

 Behrouz A Forouzan and Firouz Mosharraf.
Foundations of Computer Science.
Cengage Learning EMEA, 2007.