Binary Representation

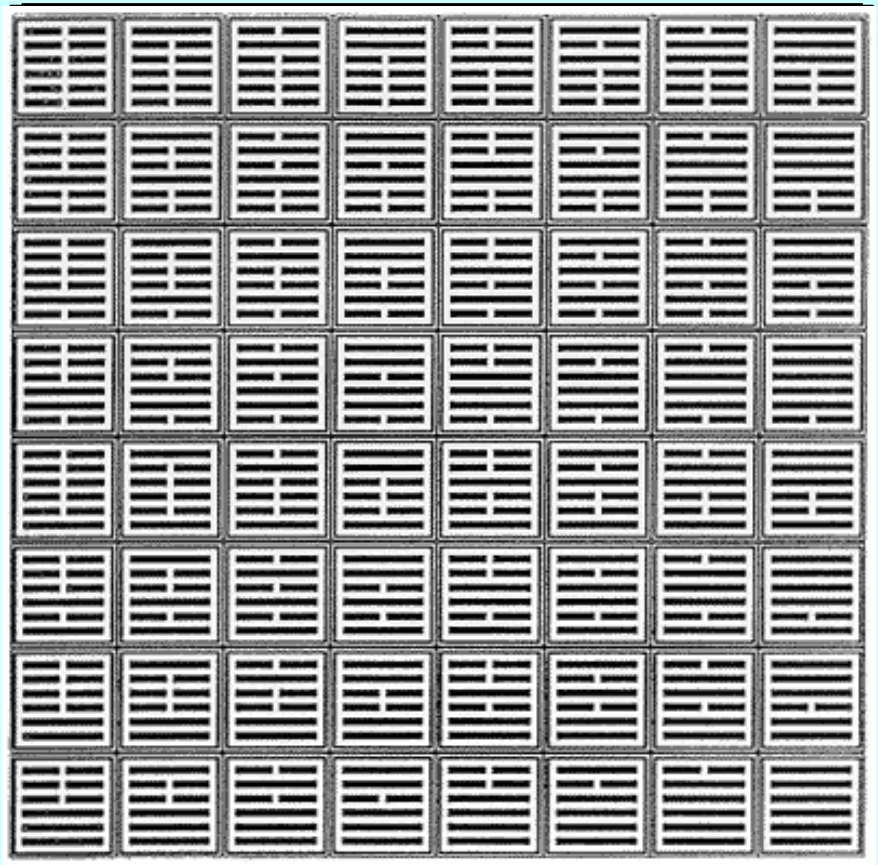
Jerry Cain
CS 106AX
October 18, 2023
slides leveraged from those constructed by Eric Roberts

The Power of Bits

- The fundamental unit of memory inside a computer is called a ***bit***—a term introduced in a paper by Claude Shannon as a contraction of the words *binary digit*.
- An individual bit exists in one of two states, usually denoted as **0** and **1**.
- More sophisticated data can be represented by combining larger numbers of bits:
  - Two bits can represent four (2 × 2) values.
  - Three bits can represent eight (2 × 2 × 2) values.
  - Four bits can represent 16 (2^4) values, and so on.
- This laptop has 64GB of main memory and can therefore exist in $2^{549,755,813,888}$ states, because 64GB is actually 2^{39} = 549,755,813,888 independent bits.

Leibniz and Binary Notation

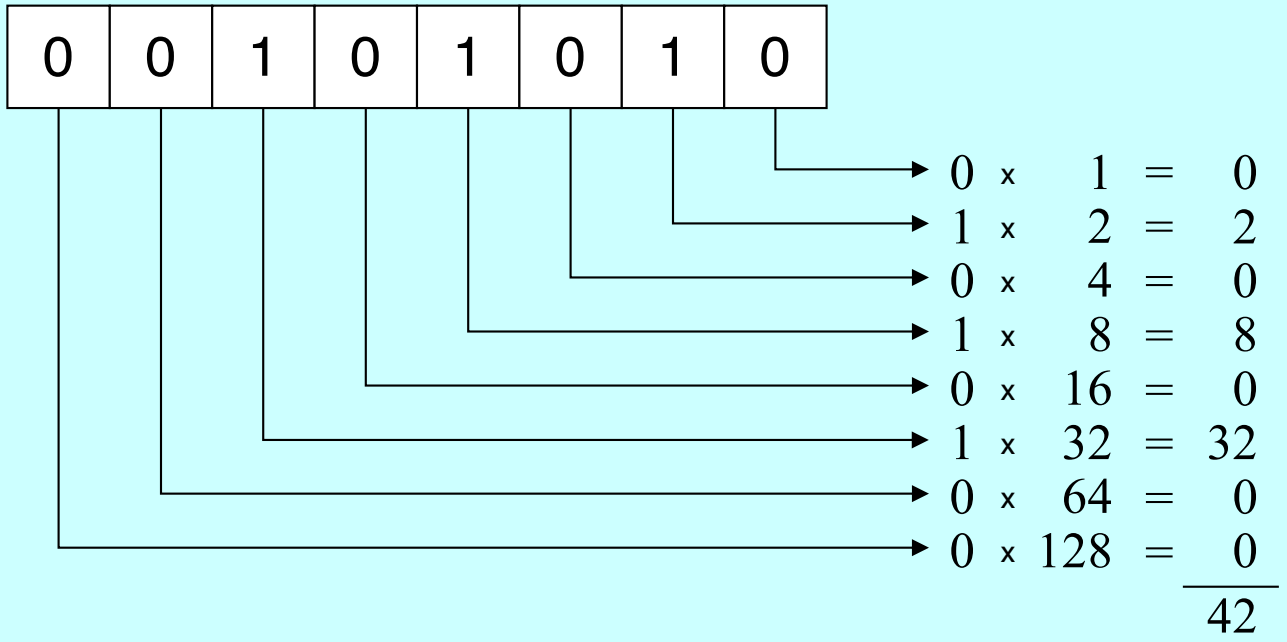
- Binary notation is an old idea. It was described back in 1703 by the German mathematician Gottfried Wilhelm von Leibniz.
- Writing in the proceedings of the French Royal Academy of Science, Leibniz describes his use of binary notation in a simple, easy-to-follow style.
- Leibniz's paper further suggests that the Chinese were clearly familiar with binary arithmetic thousands of years prior, as evidenced by the patterns of lines found in the *I Ching*.



qu'elle ſert à la perfection de la ſcience des Nombres. Ainſi je n'y employe point d'autres caracteres que 0 & 1, & puis allant à deux, je recommence. C'eſt pourquoi *deux* s'écrit ici par 10, & deux fois deux ou *quatre* par 100; & deux fois quatre ou *huit* par 1000; & deux fois huit ou *ſeize* par 10000, & ainſi de ſuite. Voici *la Table des Nombres* de cette façon, qu'on peut continuer tant que l'on voudra.

Using Bits to Represent Integers

- Binary notation is like decimal notation but uses a different ***base***. Decimal numbers use 10 as their base, which means that each digit counts for ten times as much as the digit to its right. Binary notation uses base 2, which means that each position counts for twice as much instead of 10, as follows:



Numbers and Bases

- The calculation at the end of the preceding slide makes it clear that the binary representation 00101010 is equivalent to the number 42. When it is important to distinguish the base, the text uses a small subscript, like this:

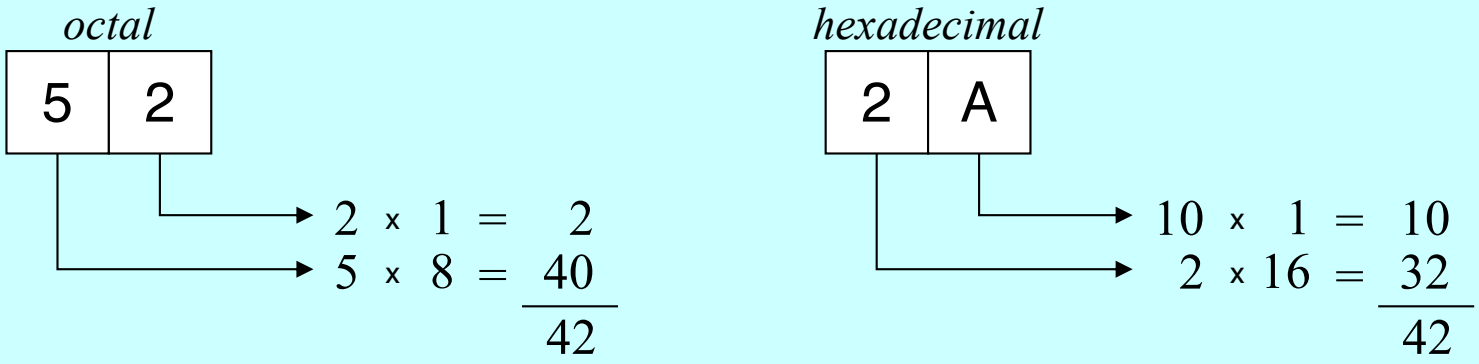
$$00101010_2 = 42_{10}$$

- Although it is useful to be able to convert a number from one base to another, it is important to remember that the number remains the same. What changes is how you write it down.
- The number 42 is what you get if you count how many stars are in the pattern at the right. The number is the same whether you write it in English as *forty-two,* in decimal as 42, or in binary as 00101010.

- Numbers do not have bases. Representations do.

Octal and Hexadecimal Notation

- Because binary notation tends to get rather long, computer scientists often prefer ***octal*** (base 8) or ***hexadecimal*** (base 16) notation instead. Octal notation uses eight digits: 0 to 7. Hexadecimal notation uses sixteen digits: 0 to 9, followed by the letters `A` through `F` to indicate the values 10 to 15.
- The following diagrams show how the number forty-two appears in both octal and hexadecimal notation:



- The advantage of using either octal or hexadecimal notation is that doing so makes it easy to translate the number back to individual bits because you can convert each digit separately.

Exercises: Number Bases

- What is the decimal value for each of the following numbers?

10001_2 — 17

177_8 — 127

AD_{16} — 173

- As part of a code to identify the file type, every Java class file begins with the following sixteen bits:

1	1	0	0	1	0	1	0	1	1	1	1	1	1	1	0

How would you express that number in hexadecimal notation?

1	1	0	0	1	0	1	0	1	1	1	1	1	1	1	0

$CAFE_{16}$

Exercise: Perfect Numbers in Binary

- Greek mathematicians took a special interest in numbers that are equal to the sum of their ***proper divisors***, which are simply those divisors less than the number itself. They called such numbers ***perfect numbers***.
- For example, 6 is a perfect number because it is the sum of 1, 2, and 3, which are the integers less than 6 that divide into 6 evenly. The next three perfect numbers—all of which were known to Greeks mathematicians—are 28, 496, and 8128.
- Convert each of these numbers into its binary representation:

$$6 = 110_2$$
$$28 = 11100_2$$
$$496 = 111110000_2$$
$$8128 = 1111111000000_2$$

Bits and Representation

- Sequences of bits have no intrinsic meaning except for whatever meaning that we assign to them, both by convention and by building specific operations into the hardware.
- As an example, a 32-bit word represents an integer only because we have designed hardware that can manipulate those figures arithmetically, applying operations such as addition, subtraction, and comparison.
- By choosing an appropriate representation, you can use bits to represent any value you can imagine:
  - Characters are represented using numeric character codes.
  - Floating-point representation supports real numbers.
  - Two-dimensional arrays of bits represent images.
  - Sequences of images represent video.
  - And so on . . .

Representing Characters

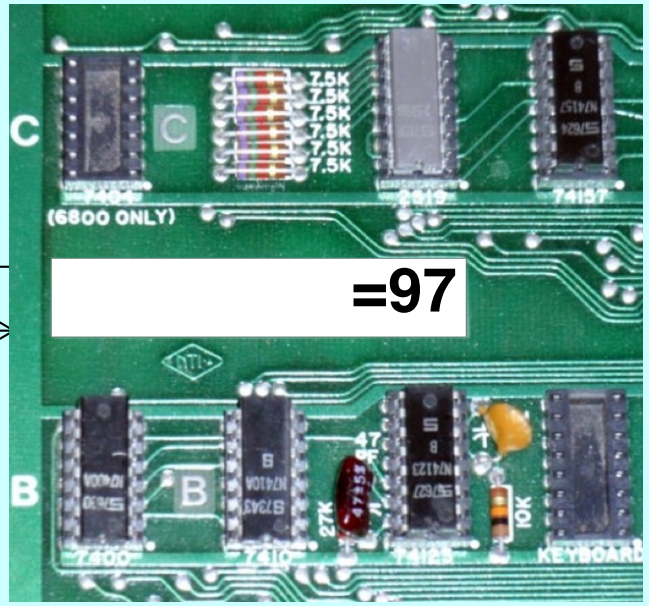
- Computers use numeric encodings to represent character data inside the memory of the machine, in which each character is assigned an integral value.
- Character codes, however, are not very useful unless they are standardized. When different computer manufacturers use different coding sequence (as was indeed the case in the early years), it is harder to share such data across machines.
- The first widely adopted character encoding was ASCII (*American Standard Code for Information Interchange*).
- With only 256 possible characters, the ASCII system proved inadequate to represent the many alphabets in use throughout the world. It has therefore been superseded by Unicode, which allows for a much larger number of characters.

The ASCII Subset of Unicode

The table below shows the first 128 characters in the Unicode character set, which are the same as those in the older ASCII set:

	0	1	2	3	4	5	6	7
00*x*	\000	\001	\002	\003	\004	\005	\006	\007
01*x*	\b	\t	\n	\011	\f	\r	\016	\017
02*x*	\020	\021	\022	\023	\024	\025	\026	\027
03*x*	\030	\031	\032	\033	\034	\035	\036	\037
04*x*	*space*	!	"	#	$	%	&	'
05*x*	(	)	*	+	,	-	.	/
06*x*	0	1	2	3	4	5	6	7
07*x*	8	9	:	;	<	=	>	?
10*x*	@	A	B	C	D	E	F	G
11*x*	H	I	J	K	L	M	N	O
12*x*	P	Q	R	S	T	U	V	W
13*x*	X	Y	Z	[	\	]	^	_
14*x*	`	a	b	c	d	e	f	g
15*x*	h	i	j	k	l	m	n	o
16*x*	p	q	r	s	t	u	v	w
17*x*	x	y	z	{	|	}	~	\177

Hardware Support for Characters



Unicode `String` Methods

`String.fromCharCode(`*code*`)`** Returns the one-character string whose Unicode value is *code*.
`charCodeAt(`*index*`)`** Returns the Unicode value of the character at the specified index.

- These two methods allow us to alternate between characters and their internal numeric representations.
- While it's rarely important to know what numbers back each character, it **is** important to remember that neighboring lowercase letters have consecutive codes, so that `"b".charCodeAt(0) - "a".charCodeAt(0)` is always 1.
- The uppercase letters are also laid out contiguously, as are the ten digit characters.
- String algorithms can and often do leverage these guarantees.

Exercise: Implementing `toUpperCase`

- Pretend the `toUpperCase` method doesn't exist. Implement a `toUpperCase` function that returns the same result.

```
/*
 * Function: toUpperCase
 * ---------------------
 * Accepts the provided string and returns the same string
 * where all alphabetic letters are uppercase.
 */
function toUpperCase(str) {
   let result = "";
   for (let i = 0; i < str.length; i++) {
      let ch = str.charAt(i);
      if (ch >= "a" && ch <= "z") {
         let offset = ch.charCodeAt(0) - "a".charCodeAt(0);
         ch = String.fromCharCode("A".charCodeAt(0) + offset);
      }
      result += ch;
   }
   return result;
}
```

The End