Bits and Bytes

A *binary digit* or *bit* has a value of either 0 or 1; these are the values we can store in hardware devices.

A *byte* is a sequence of 8 bits.

A byte is also the fundamental unit of storage in memory.

A *nybble* is a sequence of 4 bits (half of a byte).

Consider the table at right:

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Impact of Hardware Limitations

Any storage system will have only a finite number of storage devices.

Whatever scheme we use to represent integer values, we can only allocate a finite number of storage devices to the task.

Put differently, we can only represent a (small) finite number of bits for any integer value.

This means that computations, even those involving only integers, are inherently different on a computer than in mathematics.

Example: 32-bit Integers

As an example, suppose that we decide to provide support for integer values represented by 32 bits.

There are 2^32 or precisely 4,294,967,296 different patterns of 32 bits.

So we can only represent that many different integer values.

Which integer values we actually represent will depend on how we interpret the 32 bits:

1 bit for sign, 31 for magnitude (abs value):	-2147483647 to +2147483647
32 bits for magnitude (no negatives):	0 to +4294967295
2's complement representation:	-2147483648 to +2147483647

Integer Data Types

data type	a collection of values together with the definitions of a number of operations that can be performed on those values

We need to provide support for a variety of data types.

For integer values, we need to provide a variety of types that allow the user to choose based upon memory considerations and range of representation.

For contemporary programming languages, we would expect:

- signed integers and unsigned integers
- 8-, 16-, 32- and (perhaps) 64-bit representations
- the common arithmetic operations (addition, subtraction, multiplication, division, etc.)
- sensible handling of issues related to limited ranges of representation
- sensible handling of computational errors resulting from abuse of operations

Unsigned Integers: Pure Base-2

We store the number in base-2, using a total of n bits to represent its value.

Common values for n include 8, 16, 32 and 64, although any positive number of bits would work.

The range of represented values will extend from 0 to 2^n – 1.

Signed Integers: 2's Complement Form

For non-negative integers, represent the value in base-2, using up to n – 1 bits, and pad to 32 bits with leading 0's:

```
 42:   101010 --> 0010 1010
```

For negative integers, take the base-2 representation of the value (ignoring the sign) pad with 0's to n – 1 bits, invert the bits and add 1:

```
-42:   101010 --> 0010 1010
              --> 1101 0101
              --> 1101 0110
```

Weird! What's the point? Well, we've represented -42 in such a way that if we use the usual add/carry algorithm we'll find that 42 + -42 yields 0 (obviously desirable):

```
 42:   0010 1010
-42:   1101 0110
sum:   0000 0000 (ignore carry-out)
```

2's Complement Observations

To negate an integer, with one exception*, just invert the bits and add 1.

```
   25985:  0110 0101 1000 0001

  -25985:  1001 1010 0111 1111

 --25985:  0110 0101 1000 0001
```

The sign of the integer is indicated by the leading bit.

There is only one representation of the value 0.

The range of representation is asymmetrical about zero:

minimum -2^{n-1}

maximum $2^{n-1} - 1$

* QTP

2's Complement Shortcut

To negate an integer, with one exception*, find the right-most bit that equals 1 and then invert all of the bits to its left:

```
 3328:   0000 1101 0000 0000

-3328:   1111 0011 0000 0000
```

Why does this work?

2's Complement to base-10

If the integer is non-negative, just expand the positional representation:

```
0000 1101 0000 0000   =   2^11 + 2^10 + 2^8
                      =   3328
```

If the integer is negative, take its negation (in 2's complement), expand the positional representation for that, and then take the negation of the result (in base-10).

base-10 to 2's Complement

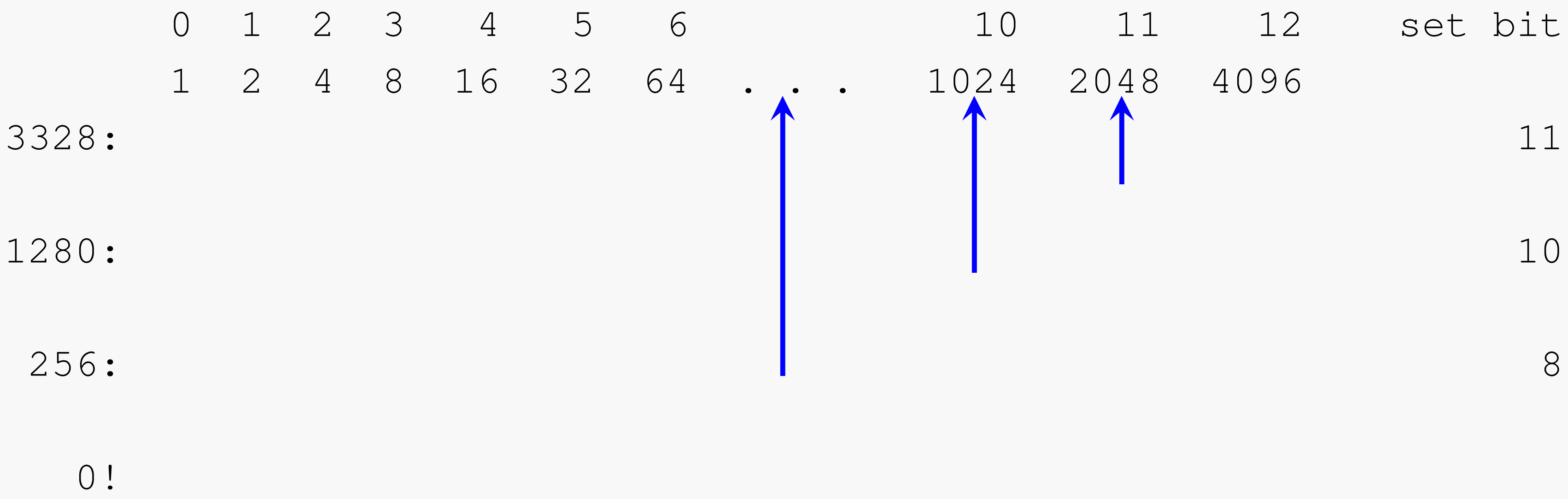
Obvious method:

- apply the division-by-2 algorithm discussed earlier to the magnitude of the number
- if value is negative, negate the result

base-10 to 2's Complement

Alternate method:

- find the largest power of 2 that's less than the magnitude of the number
- subtract it from the magnitude of the number and set that bit-position to 1
- repeat until the magnitude equals 0
- if value is negative, negate the result



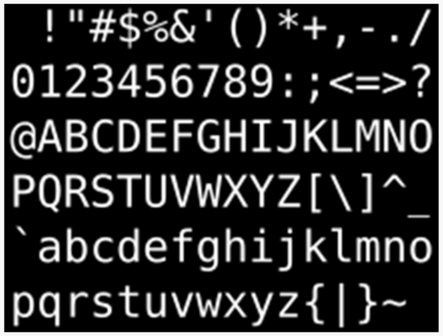
```
        0  1  2  3   4   5   6           10    11    12      set bit
        1  2  4  8  16  32  64   . . .  1024  2048  4096
3328:                                                            11

1280:                                                            10

 256:                                                             8

   0!
```

ASCII

The American Standard Code for Information Interchange maps a set of 128 characters into the set of integers from 0 to 127, requiring 7 bits for each numeric code:

95 of the characters are "printable" and are mapped into the codes 32 to 126:

The remainder are special control codes (e.g., WRU, RU, tab, line feed, etc.).



```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
```

Since the fundamental unit of data storage was quickly standardized as an 8-bit byte, the high bit was generally either set to 0 or used as a *parity-check bit*.

Some ASCII Features

The decimal digits '0' through '9' are assigned sequential codes.

Therefore, the numeric value of a digit can be obtained by subtraction: '7' – '0' = 7

The upper-case characters 'A' through 'Z' are also assigned sequential codes, as are the lower-case characters 'a' through 'z'.

This aids in sorting of character strings, but note that upper-case characters have lower-valued codes than do upper-case characters.

There are no new operations, but since ASCII codes are numeric values, it is often possible to perform arithmetic on them to achieve useful results…

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
```

ASCII Table

It's easy to find ASCII tables online (including some that are clearer than this one):

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Extended ASCII

For good or ill, the ASCII codes are 7-bit codes, and that leads to temptation.

There exist 8-bit character encodings that extend the ASCII codes to provide for 256 different characters (e.g., ISO_8859-1:1987).

Unfortunately, none of these has achieved the status of a practical Standard in use.

ASCII Art

```
       ____
   .-'&    '-.
  /           \
 :   o    o    ;
(      (_       )
 :             ;
  \    __     /
   `-._____.-'
     /`"""`\
    /    ,  \
   /|/\/\/\ _\
  (_|/\/\/\\__)
    |_______|
   __)_ |_ (__
  (_____|_____)
```

Just for fun…

Boolean Logical Values

We must represent two values, TRUE and FALSE, so a single bit suffices.

We will represent TRUE by 1 and FALSE by 0.

Thus, a sequence of bits can be viewed as a sequence of logical values.

Note: this is not the view typically taken in high-level languages!

Boolean Logical Operations

Given two Boolean logical values, there are a number of operations we can perform:

A	NOT A
0	1
1	0

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1