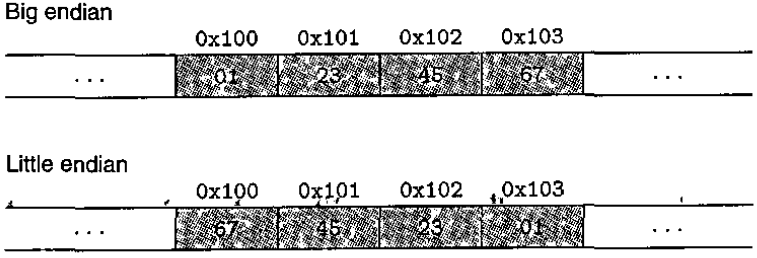
on the numeric ranges of the different data types, as will be covered later, but there are no upper bounds (except with the fixed-size types). With 32-bit machines and 32-bit programs being the dominant combination from around 1980 until around 2010, many programs have been written assuming the allocations listed for 32-bit programs in Figure 2.3. With the transition to 64-bit machines, many hidden word size dependencies have arisen as bugs in migrating these programs to new machines. For example, many programmers historically assumed that an object declared as type `int` could be used to store a pointer. This works fine for most 32-bit programs, but it leads to problems for 64-bit programs.

2.1.3 Addressing and Byte Ordering

For program objects that span multiple bytes, we must establish two conventions: what the address of the object will be, and how we will order the bytes in memory. In virtually all machines, a multi-byte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used. For example, suppose a variable `x` of type `int` has address `0x100`; that is, the value of the address expression `&x` is `0x100`. Then (assuming data type `int` has a 32-bit representation) the 4 bytes of `x` would be stored in memory locations `0x100`, `0x101`, `0x102`, and `0x103`.

For ordering the bytes representing an object, there are two common conventions. Consider a w-bit integer having a bit representation $[x_{w-1}, x_{w-2}, \ldots, x_1, x_0]$, where x_{w-1} is the most significant bit and x_0 is the least. Assuming w is a multiple of 8, these bits can be grouped as bytes, with the most significant byte having bits $[x_{w-1}, x_{w-2}, \ldots, x_{w-8}]$, the least significant byte having bits $[x_7, x_6, \ldots, x_0]$, and the other bytes having bits from the middle. Some machines choose to store the object in memory ordered from least significant byte to most, while other machines store them from most to least. The former convention—where the least significant byte comes first—is referred to as *little endian*. The latter convention—where the most significant byte comes first—is referred to as *big endian*.

Suppose the variable `x` of type `int` and at address `0x100` has a hexadecimal value of `0x01234567`. The ordering of the bytes within the address range `0x100` through `0x103` depends on the type of machine:



Note that in the word `0x01234567` the high-order byte has hexadecimal value `0x01`, while the low-order byte has value `0x67`.

Most Intel-compatible machines operate exclusively in little-endian mode. On the other hand, most machines from IBM and Oracle (arising from their acquisi-

tion of Sun Microsystems in 2010) operate in big-endian mode. Note that we said "most." The conventions do not split precisely along corporate boundaries. For example, both IBM and Oracle manufacture machines that use Intel-compatible processors and hence are little endian. Many recent microprocessor chips are *bi-endian*, meaning that they can be configured to operate as either little- or big-endian machines. In practice, however, byte ordering becomes fixed once a particular operating system is chosen. For example, ARM microprocessors, used in many cell phones, have hardware that can operate in either little- or big-endian mode, but the two most common operating systems for these chips—Android (from Google) and IOS (from Apple)—operate only in little-endian mode.

People get surprisingly emotional about which byte ordering is the proper one. In fact, the terms "little endian" and "big endian" come from the book *Gulliver's Travels* by Jonathan Swift, where two warring factions could not agree as to how a soft-boiled egg should be opened—by the little end or by the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, and hence the arguments degenerate into bickering about sociopolitical issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

For most application programmers, the byte orderings used by their machines are totally invisible; programs compiled for either class of machine give identical results. At times, however, byte ordering becomes an issue. The first is when

binary data are communicated over a network between different machines. A common problem is for data produced by a little-endian machine to be sent to a big-endian machine, or vice versa, leading to the bytes within the words being in reverse order for the receiving program. To avoid such problems, code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation. We will see examples of these conversions in Chapter 11.

A second case where byte ordering becomes important is when looking at the byte sequences representing integer data. This occurs often when inspecting machine-level programs. As an example, the following line occurs in a file that gives a text representation of the machine-level code for an Intel x86-64 processor:

```
4004d3:  01 05 43 0b 20 00        add     %eax,0x200b43(%rip)
```

This line was generated by a *disassembler*, a tool that determines the instruction sequence represented by an executable program file. We will learn more about disassemblers and how to interpret lines such as this in Chapter 3. For now, we simply note that this line states that the hexadecimal byte sequence `01 05 43 0b 20 00` is the byte-level representation of an instruction that adds a word of data to the value stored at an address computed by adding `0x200b43` to the current value of the *program counter*, the address of the next instruction to be executed. If we take the final 4 bytes of the sequence `43 0b 20 00` and write them in reverse order, we have `00 20 0b 43`. Dropping the leading 0, we have the value `0x200b43`, the numeric value written on the right. Having bytes appear in reverse order is a common occurrence when reading machine-level program representations generated for little-endian machines such as this one. The natural way to write a byte sequence is to have the lowest-numbered byte on the left and the highest on the right, but this is contrary to the normal way of writing numbers with the most significant digit on the left and the least on the right.

A third case where byte ordering becomes visible is when programs are written that circumvent the normal type system. In the C language, this can be done using a *cast* or a *union* to allow an object to be referenced according to a different data type from which it was created. Such coding tricks are strongly discouraged for most application programming, but they can be quite useful and even necessary for system-level programming.

Figure 2.4 shows C code that uses casting to access and print the byte representations of different program objects. We use `typedef` to define data type `byte_pointer` as a pointer to an object of type `unsigned char`. Such a byte pointer references a sequence of bytes where each byte is considered to be a nonnegative integer. The first routine `show_bytes` is given the address of a sequence of bytes, indicated by a byte pointer, and a byte count. The byte count is specified as having data type `size_t`, the preferred data type for expressing the sizes of data structures. It prints the individual bytes in hexadecimal. The C formatting directive `%.2x` indicates that an integer should be printed in hexadecimal with at least 2 digits.

```
1  #include <stdio.h>
2
3  typedef unsigned char *byte_pointer;
4
5  void show_bytes(byte_pointer start, size_t len) {
6      int i;
7      for (i = 0; i < len; i++)
8          printf(" %.2x", start[i]);
9      printf("\n");
10 }
11
12 void show_int(int x) {
13     show_bytes((byte_pointer) &x, sizeof(int));
14 }
15
16 void show_float(float x) {
17     show_bytes((byte_pointer) &x, sizeof(float));
18 }
19
20 void show_pointer(void *x) {
21     show_bytes((byte_pointer) &x, sizeof(void *));
22 }
```

Figure 2.4 Code to print the byte representation of program objects. This code uses casting to circumvent the type system. Similar functions are easily defined for other data types.

Procedures `show_int`, `show_float`, and `show_pointer` demonstrate how to use procedure `show_bytes` to print the byte representations of C program objects of type `int`, `float`, and `void *`, respectively. Observe that they simply pass `show_bytes` a pointer `&x` to their argument `x`, casting the pointer to be of type `unsigned char *`. This cast indicates to the compiler that the program should consider the pointer to be to a sequence of bytes rather than to an object of the original data type. This pointer will then be to the lowest byte address occupied by the object.

These procedures use the C `sizeof` operator to determine the number of bytes used by the object. In general, the expression `sizeof(`T`)` returns the number of bytes required to store an object of type T. Using `sizeof` rather than a fixed value is one step toward writing code that is portable across different machine types.

We ran the code shown in Figure 2.5 on several different machines, giving the results shown in Figure 2.6. The following machines were used:

Linux 32	Intel IA32 processor running Linux.
Windows	Intel IA32 processor running Windows.
Sun	Sun Microsystems SPARC processor running Solaris. (These machines are now produced by Oracle.)
Linux 64	Intel x86-64 processor running Linux.

code/data/show-bytes.c

```
1  void test_show_bytes(int val) {
2      int ival = val;
3      float fval = (float) ival;
4      int *pval = &ival;
5      show_int(ival);
6      show_float(fval);
7      show_pointer(pval);
8  }
```

code/data/show-bytes.c

Figure 2.5 Byte representation examples. This code prints the byte representations of sample data objects.

Machine	Value	Type	Bytes (hex)
Linux 32	12,345	`int`	39 30 00 00
Windows	12,345	`int`	39 30 00 00
Sun	12,345	`int`	00 00 30 39
Linux 64	12,345	`int`	39 30 00 00
Linux 32	12,345.0	`float`	00 e4 40 46
Windows	12,345.0	`float`	00 e4 40 46
Sun	12,345.0	`float`	46 40 e4 00
Linux 64	12,345.0	`float`	00 e4 40 46
Linux 32	`&ival`	`int *`	e4 f9 ff bf
Windows	`&ival`	`int *`	b4 cc 22 00
Sun	`&ival`	`int *`	ef ff fa 0c
Linux 64	`&ival`	`int *`	b8 11 e5 ff ff 7f 00 00

Figure 2.6 Byte representations of different data values. Results for `int` and `float` are identical, except for byte ordering. Pointer values are machine dependent.

Our argument 12,345 has hexadecimal representation 0x00003039. For the `int` data, we get identical results for all machines, except for the byte ordering. In particular, we can see that the least significant byte value of 0x39 is printed first for Linux 32, Windows, and Linux 64, indicating little-endian machines, and last for Sun, indicating a big-endian machine. Similarly, the bytes of the `float` data are identical, except for the byte ordering. On the other hand, the pointer values are completely different. The different machine/operating system configurations use different conventions for storage allocation. One feature to note is that the Linux 32, Windows, and Sun machines use 4-byte addresses, while the Linux 64 machine uses 8-byte addresses.

New to C? Naming data types with `typedef`

The `typedef` declaration in C provides a way of giving a name to a data type. This can be a great help in improving code readability, since deeply nested type declarations can be difficult to decipher.

The syntax for `typedef` is exactly like that of declaring a variable, except that it uses a type name rather than a variable name. Thus, the declaration of `byte_pointer` in Figure 2.4 has the same form as the declaration of a variable of type `unsigned char *`.

For example, the declaration

```
typedef int *int_pointer;
int_pointer ip;
```

defines type `int_pointer` to be a pointer to an `int`, and declares a variable `ip` of this type. Alternatively, we could declare this variable directly as

```
int *ip;
```

New to C? Formatted printing with `printf`

The `printf` function (along with its cousins `fprintf` and `sprintf`) provides a way to print information with considerable control over the formatting details. The first argument is a *format string*, while any remaining arguments are values to be printed. Within the format string, each character sequence starting with '%' indicates how to format the next argument. Typical examples include `%d` to print a decimal integer, `%f` to print a floating-point number, and `%c` to print a character having the character code given by the argument.

Specifying the formatting of fixed-size data types, such as `int_32t`, is a bit more involved, as is described in the aside on page 67.

Observe that although the floating-point and the integer data both encode the numeric value 12,345, they have very different byte patterns: 0x00003039 for the integer and 0x4640E400 for floating point. In general, these two formats use different encoding schemes. If we expand these hexadecimal patterns into binary form and shift them appropriately, we find a sequence of 13 matching bits, indicated by a sequence of asterisks, as follows:

```
    0   0   0   0   3   0   3   9
00000000000000000011000000111001
                ************
          4   6   4   0   E   4   0   0
01000110010000001110010000000000
```

This is not coincidental. We will return to this example when we study floating-point formats.

New to C? Pointers and arrays

In function `show_bytes` (Figure 2.4), we see the close connection between pointers and arrays, as will be discussed in detail in Section 3.8. We see that this function has an argument `start` of type `byte_pointer` (which has been defined to be a pointer to `unsigned char`), but we see the array reference `start[i]` on line 8. In C, we can dereference a pointer with array notation, and we can reference array elements with pointer notation. In this example, the reference `start[i]` indicates that we want to read the byte that is `i` positions beyond the location pointed to by `start`.

New to C? Pointer creation and dereferencing

In lines 13, 17, and 21 of Figure 2.4 we see uses of two operations that give C (and therefore C++) its distinctive character. The C "address of" operator '`&`' creates a pointer. On all three lines, the expression `&x` creates a pointer to the location holding the object indicated by variable `x`. The type of this pointer depends on the type of `x`, and hence these three pointers are of type `int *`, `float *`, and `void **`, respectively. (Data type `void *` is a special kind of pointer with no associated type information.)

The cast operator converts from one data type to another. Thus, the cast `(byte_pointer) &x` indicates that whatever type the pointer `&x` had before, the program will now reference a pointer to data of type `unsigned char`. The casts shown here do not change the actual pointer; they simply direct the compiler to refer to the data being pointed to according to the new data type.

Aside Generating an ASCII table

You can display a table showing the ASCII character code by executing the command `man ascii`.

Practice Problem 2.5 (solution page 144)

Consider the following three calls to `show_bytes`:

```
int val = 0x87654321;
byte_pointer valp = (byte_pointer) &val;
show_bytes(valp, 1); /* A. */
show_bytes(valp, 2); /* B. */
show_bytes(valp, 3); /* C. */
```

Indicate the values that will be printed by each call on a little-endian machine and on a big-endian machine:

A. Little endian: _____ Big endian: _____

B. Little endian: _____ Big endian: _____

C. Little endian: _____ Big endian: _____

Practice Problem 2.6 (solution page 145)

Using `show_int` and `show_float`, we determine that the integer 3510593 has hexadecimal representation `0x00359141`, while the floating-point number 3510593.0 has hexadecimal representation `0x4A564504`.

A. Write the binary representations of these two hexadecimal values.

B. Shift these two strings relative to one another to maximize the number of matching bits. How many bits match?

C. What parts of the strings do not match?

2.1.4 Representing Strings

A string in C is encoded by an array of characters terminated by the null (having value 0) character. Each character is represented by some standard encoding, with the most common being the ASCII character code. Thus, if we run our routine `show_bytes` with arguments `"12345"` and 6 (to include the terminating character), we get the result 31 32 33 34 35 00. Observe that the ASCII code for decimal digit x happens to be `0x3`x, and that the terminating byte has the hex representation `0x00`. This same result would be obtained on any system using ASCII as its character code, independent of the byte ordering and word size conventions. As a consequence, text data are more platform independent than binary data.

Practice Problem 2.7 (solution page 145)

What would be printed as a result of the following call to `show_bytes`?

```
const char *s = "abcdef";
show_bytes((byte_pointer) s, strlen(s));
```

Note that letters 'a' through 'z' have ASCII codes `0x61` through `0x7A`.

2.1.5 Representing Code

Consider the following C function:

```
1  int sum(int x, int y) {
2      return x + y;
3  }
```

When compiled on our sample machines, we generate machine code having the following byte representations:

Linux 32 55 89 e5 8b 45 0c 03 45 08 c9 c3

Windows 55 89 e5 8b 45 0c 03 45 08 5d c3

Sun 81 c3 e0 08 90 02 00 09

Linux 64 55 48 89 e5 89 7d fc 89 75 f8 03 45 fc c9 c3

Aside The Unicode standard for text encoding

The ASCII character set is suitable for encoding English-language documents, but it does not have much in the way of special characters, such as the French ç. It is wholly unsuited for encoding documents in languages such as Greek, Russian, and Chinese. Over the years, a variety of methods have been developed to encode text for different languages. The Unicode Consortium has devised the most comprehensive and widely accepted standard for encoding text. The current Unicode standard (version 7.0) has a repertoire of over 100,000 characters supporting a wide range of languages, including the ancient languages of Egypt and Babylon. To their credit, the Unicode Technical Committee rejected a proposal to include a standard writing for Klingon, a fictional civilization from the television series *Star Trek*.

The base encoding, known as the "Universal Character Set" of Unicode, uses a 32-bit representation of characters. This would seem to require every string of text to consist of 4 bytes per character. However, alternative codings are possible where common characters require just 1 or 2 bytes, while less common ones require more. In particular, the UTF-8 representation encodes each character as a sequence of bytes, such that the standard ASCII characters use the same single-byte encodings as they have in ASCII, implying that all ASCII byte sequences have the same meaning in UTF-8 as they do in ASCII.

The Java programming language uses Unicode in its representations of strings. Program libraries are also available for C to support Unicode.

Here we find that the instruction codings are different. Different machine types use different and incompatible instructions and encodings. Even identical processors running different operating systems have differences in their coding conventions and hence are not binary compatible. Binary code is seldom portable across different combinations of machine and operating system.

A fundamental concept of computer systems is that a program, from the perspective of the machine, is simply a sequence of bytes. The machine has no information about the original source program, except perhaps some auxiliary tables maintained to aid in debugging. We will see this more clearly when we study machine-level programming in Chapter 3.

2.1.6 Introduction to Boolean Algebra

Since binary values are at the core of how computers encode, store, and manipulate information, a rich body of mathematical knowledge has evolved around the study of the values 0 and 1. This started with the work of George Boole (1815–1864) around 1850 and thus is known as *Boolean algebra*. Boole observed that by encoding logic values TRUE and FALSE as binary values 1 and 0, he could formulate an algebra that captures the basic principles of logical reasoning.

The simplest Boolean algebra is defined over the two-element set {0, 1}. Figure 2.7 defines several operations in this algebra. Our symbols for representing these operations are chosen to match those used by the C bit-level operations,

~	
0	1
1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Figure 2.7 Operations of Boolean algebra. Binary values 1 and 0 encode logic values TRUE and FALSE, while operations ~, &, |, and ^ encode logical operations NOT, AND, OR, and EXCLUSIVE-OR, respectively.

as will be discussed later. The Boolean operation ~ corresponds to the logical operation NOT, denoted by the symbol $\neg$. That is, we say that $\neg P$ is true when P is not true, and vice versa. Correspondingly, $\sim p$ equals 1 when p equals 0, and vice versa. Boolean operation & corresponds to the logical operation AND, denoted by the symbol $\wedge$. We say that $P \wedge Q$ holds when both P is true and Q is true. Correspondingly, $p \,\&\, q$ equals 1 only when $p = 1$ and $q = 1$. Boolean operation | corresponds to the logical operation OR, denoted by the symbol $\vee$. We say that $P \vee Q$ holds when either P is true or Q is true. Correspondingly, $p \mid q$ equals 1 when either $p = 1$ or $q = 1$. Boolean operation ^ corresponds to the logical operation EXCLUSIVE-OR, denoted by the symbol $\oplus$. We say that $P \oplus Q$ holds when either P is true or Q is true, but not both. Correspondingly, $p \,\hat{}\, q$ equals 1 when either $p = 1$ and $q = 0$, or $p = 0$ and $q = 1$.

Claude Shannon (1916–2001), who later founded the field of information theory, first made the connection between Boolean algebra and digital logic. In his 1937 master's thesis, he showed that Boolean algebra could be applied to the design and analysis of networks of electromechanical relays. Although computer technology has advanced considerably since, Boolean algebra still plays a central role in the design and analysis of digital systems.

We can extend the four Boolean operations to also operate on *bit vectors*, strings of zeros and ones of some fixed length w. We define the operations over bit vectors according to their applications to the matching elements of the arguments. Let a and b denote the bit vectors $[a_{w-1}, a_{w-2}, \ldots, a_0]$ and $[b_{w-1}, b_{w-2}, \ldots, b_0]$, respectively. We define $a \,\&\, b$ to also be a bit vector of length w, where the ith element equals $a_i \,\&\, b_i$, for $0 \leq i < w$. The operations |, ^, and ~ are extended to bit vectors in a similar fashion.

As examples, consider the case where $w = 4$, and with arguments $a = [0110]$ and $b = [1100]$. Then the four operations $a \,\&\, b$, $a \mid b$, $a \,\hat{}\, b$, and $\sim b$ yield

```
  0110        0110        0110
& 1100      | 1100      ^ 1100      ~ 1100
  0100        1110        1010        0011
```

Practice Problem 2.8 (solution page 145)

Fill in the following table showing the results of evaluating Boolean operations on bit vectors.

Web Aside DATA:BOOL More on Boolean algebra and Boolean rings

The Boolean operations |, &, and ~ operating on bit vectors of length w form a *Boolean algebra*, for any integer $w > 0$. The simplest is the case where $w = 1$ and there are just two elements, but for the more general case there are 2^w bit vectors of length w. Boolean algebra has many of the same properties as arithmetic over integers. For example, just as multiplication distributes over addition, written $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$, Boolean operation & distributes over |, written $a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, c)$. In addition, however. Boolean operation | distributes over &, and so we can write $a \mid (b \,\&\, c) = (a \mid b) \,\&\, (a \mid c)$, whereas we cannot say that $a + (b \cdot c) = (a + b) \cdot (a + c)$ holds for all integers.

When we consider operations ^, &, and ~ operating on bit vectors of length w, we get a different mathematical form, known as a *Boolean ring*. Boolean rings have many properties in common with integer arithmetic. For example, one property of integer arithmetic is that every value x has an *additive inverse* $-x$, such that $x + -x = 0$. A similar property holds for Boolean rings, where ^ is the "addition" operation, but in this case each element is its own additive inverse. That is, $a \,\hat{}\, a = 0$ for any value a, where we use 0 here to represent a bit vector of all zeros. We can see this holds for single bits, since $0 \,\hat{}\, 0 = 1 \,\hat{}\, 1 = 0$, and it extends to bit vectors as well. This property holds even when we rearrange terms and combine them in a different order, and so $(a \,\hat{}\, b) \,\hat{}\, a = b$. This property leads to some interesting results and clever tricks, as we will explore in Problem 2.10.

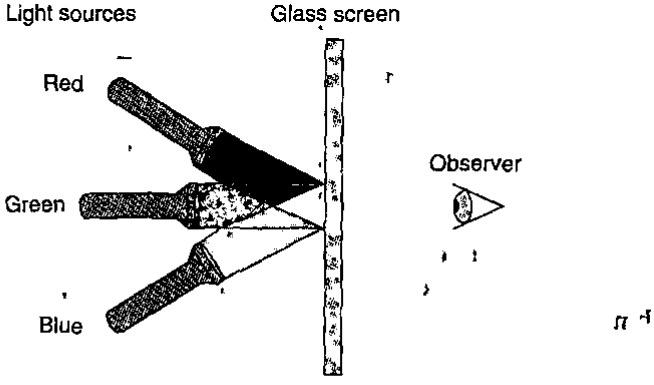
Operation	Result
a	[01101001]
b	[01010101]
$\sim a$	_____
$\sim b$	_____
$a \,\&\, b$	_____
$a \mid b$	_____
$a \,\hat{}\, b$	_____

One useful application of bit vectors is to represent finite sets. We can encode any subset $A \subseteq \{0, 1, \ldots, w-1\}$ with a bit vector $[a_{w-1}, \ldots, a_1, a_0]$, where $a_i = 1$ if and only if $i \in A$. For example, recalling that we write a_{w-1} on the left and a_0 on the right, bit vector $a = [01101001]$ encodes the set $A = \{0, 3, 5, 6\}$, while bit vector $b = [01010101]$ encodes the set $B = \{0, 2, 4, 6\}$. With this way of encoding sets, Boolean operations | and & correspond to set union and intersection, respectively, and ~ corresponds to set complement. Continuing our earlier example, the operation $a \,\&\, b$ yields bit vector [01000001], while $A \cap B = \{0, 6\}$.

We will see the encoding of sets by bit vectors in a number of practical applications. For example, in Chapter 8, we will see that there are a number of different *signals* that can interrupt the execution of a program. We can selectively enable or disable different signals by specifying a bit-vector mask, where a 1 in bit position i indicates that signal i is enabled and a 0 indicates that it is disabled. Thus, the mask represents the set of enabled signals.

Practice Problem 2.9 (solution page 146)

Computers generate color pictures on a video screen or liquid crystal display by mixing three different colors of light: red, green, and blue. Imagine a simple scheme, with three different lights, each of which can be turned on or off, projecting onto a glass screen:



We can then create eight different colors based on the absence (0) or presence (1) of light sources R, G, and B:

R	G	B	Color
0	0	0	Black
0	0	1	Blue
0	1	0	Green
0	1	1	Cyan
1	0	0	Red
1	0	1	Magenta
1	1	0	Yellow
1	1	1	White

Each of these colors can be represented as a bit vector of length 3, and we can apply Boolean operations to them.

A. The complement of a color is formed by turning off the lights that are on and turning on the lights that are off. What would be the complement of each of the eight colors listed above?

B. Describe the effect of applying Boolean operations on the following colors:

Blue | Green = _____
Yellow & Cyan = _____
Red ^ Magenta = _____