Representation of Data

In contrast with higher-level programming languages, C does not provide strong abstractions for representing data. Indeed, while languages like Racket has a rich notion of data type—integers, floats, strings, tuples, lists, etc.—C has a low-level perspective on (primitive) data: a datum is either an integer or float of some finite size. While inconvenient at times, this perspective allows C programs to manipulate data in ways that are impossible in other languages. In turn, programmers are able to maximize performance, minimize space usage, perform other sorts of bit-level tricks, or otherwise have a better handle on precisely how data is represented in their programs.

In order to take advantage of the benefits that systems programming offers, the programmer must understand these various representations and able to manipulate their data at this level. Because C breaks up data in two sorts, integers and floats, we study each in turn.

Integer Data

Integer data correspond to numbers without fractional components. C represents integer data using the primitive types char, int, and their various modifications with short, long, and unsigned.

Bits and Bytes

A bit, a value that is either 0 or 1, is the fundamental unit of data in modern computing. All data in a computer is ultimately realized as sequences of 0 and 1. This concept came about with the origins of computing in the punch cards used to power Joseph Marie Jacquard’s automated looms in the early 1800s and Herman Hollerith’s counting machines in the late 1800s. Punch cards are pieces of thick paper adorned with grids of circles (or other shapes) that may be perforated (representing a 1) or not (representing a 0). Machines are fed these punch cards and through mechanical or electric means, detect the perforations and interprets them as commands or data.

In a modern-day computer, bits are realized as electrical charge within circuits. In particular, a latch or flip-flop is an elementary circuit that stores a stable electrical charge. A low charge is interpreted as one bit value (usually 0) and a high charge is represented as a high bit value (usually 1). A computer is composed of many such circuits, e.g., an 8-core Core i7 Haswell-E Intel Processor alone has 2.6 billion of them\(^1\) and the remaining parts of the computer, e.g., main

\(^1\)https://en.wikipedia.org/wiki/Transistor_count
memory, contain many more!

While programmers are sometimes concerned with manipulating individual bits, the primitives of C are composed of many bits. For example, on most platforms, a char is 8 bits, an int is 32 bits, and a double is 64 bits. It is most convenient to use a larger unit to talk about these sizes, the byte. A byte is simply 8 bits, so the size of a char is 1 byte, an int is 4 bytes, and a double is 8 bytes. The choice of 8 bits is not completely arbitrary; it is the smallest unit of addressable memory on most computer architectures because it is the size of a single character (a char) using the ANSI character standard. That is, the computer allows us to access and modify data in terms of whole bytes rather than individual bits.

Binary Representation

Now that bits and bytes are defined, how does a computer use them to represent integer data? Recall that an integer is a number without a fractional portion, either positive, negative, or zero. To answer this question, we need to revisit the number system that we use to write down numbers. A number system is a way to write down a particular quantity on paper.

For example, consider the number: 8401.

It is easy to take for granted the meaning of this string of digits as “eight thousand, four-hundred, and one”. However, let’s break down the meaning of the string precisely and then use that as motivation for how a computer represents an integer. Each digit is assigned a particular meaning based off its position in the string. Starting from right-to-left, the first position is traditionally called the “ones-place”, the second position the “tens-place”, the third position the “hundreds-place”, and finally the fourth position the “thousands-place”. Intuitively, this is “how many” of each type of value the string represents, i.e., there are 1 ones, 0 tens, 4 hundreds, and 8 thousands in this quantity.

More formally, we can write this down as a summation of powers:

\[ 8401 = 8 \cdot 10^3 + 4 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0 = 8000 + 400 + 0 + 1 = 8401. \]

This way of writing numbers is called Hindu-Arabic number system. It is a positional (positions denote powers), decimal (the base of the powers is 10, commonly written base-10) number system. This is why the set of valid digits is 0–9: each position can take on 10 values denoting how many of each power a quantity can take on before it “rolls over” into the next power.

However, the choice of base-10 seems arbitrary. Can we choose another base and represent the same range of numbers? The answer is yes and it leads us to the binary representation of integers that computers use.

A binary number is simply a number written in positional, base-2 style. In base-2, each position only requires two possible values, traditionally written “0” and “1”. For example, here is a binary...

---

2 The ANSI character standard only accounts for English characters which was sufficient for the earliest computers. But as computers have become ubiquitous in society, programs need to account for many more characters than English, e.g., Cyrillic or Kanji letters. The standard for modern computing is Unicode which is capable of representing many more characters at the cost of being two bytes rather than one byte in size.
number: 10101.

Like the traditional base-10 system, each position represents powers of different magnitude. However, in the base-2 system, the bases of the powers are two rather than ten. To determine what quantity this binary number represents, we can write down the quantity as a summation of powers:

\[ "10101" = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 16 + 0 + 4 + 0 + 1 = 21. \]

Note that because binary numbers use either 0s or 1s as digits, computing the value of a binary number amounts to adding powers of twos in various combinations. It is therefore useful to know the powers of two by memory as these numbers come up again and again when working at the level of bits and bytes:

- \( 2^0 = 1 \)
- \( 2^1 = 2 \)
- \( 2^2 = 4 \)
- \( 2^3 = 8 \)
- \( 2^4 = 16 \)
- \( 2^5 = 32 \)
- \( 2^6 = 64 \)
- \( 2^7 = 128 \)
- \( 2^8 = 256 \)
- \( 2^9 = 512 \)
- \( 2^{10} = 1024 \)

A computer represents integers as a sequence of bits of a pre-determined length. The quantity these bits represents corresponds to the value of the sequence of bits interpreted as a binary number. In C, this quantity corresponds to the number of bits dedicated to each type, e.g., a char is an integer value consisting of one byte (eight bits) and an int consists of four bytes (32 bits).

Note that other bases are possible and are sometimes more convenient to use when representing longer strings of bits. Two common examples are octal numbers, base-8, and hexadecimal numbers, base-16. With octal numbers, the range of allowable digits is 0–7. For example, the quantity 21 (written in base-10) or 10101 (written in base-2) can be represented in octal as:

\[ "25" = 2 \cdot 8^1 + 5 \cdot 8^0 = 16 + 5 = 21. \]

In contrast, base-16 requires 16 possible values in each digit. However, there only have 9 numbers, 0–9, in most written languages! To get around this limitation, hexadecimal uses the letters A–F for the values 10–15, respectively:

\[ "76D" = 7 \cdot 16^2 + 6 \cdot 16^1 + 13 \cdot 16^0 = 1792 + 96 + 14 = 1901. \]

These different notations can be confusing because, for example, a string "10" has different interpretations depending on the base:

- In base-10, "10" = 1 \cdot 10^1 + 0 \cdot 10^0 = 10 + 0 = 10.
• In base-2, “10” = 1 · 2^1 + 0 · 2^0 = 2 + 0 = 2.
• In base-8, “10” = 1 · 8^1 + 0 · 8^0 = 8 + 0 = 8.
• In base-16, “10” = 1 · 16^1 + 0 · 16^0 = 16 + 0 = 16.

To distinguish between them, we usually include a prefix before the less common notations to denote what base they are in:
- “0b10” implies binary.
- “010” implies octal.
- “0x10” implies hexadecimal.

In C, we can write octal or hexadecimal (but not binary) integer literals using these prefixes, e.g.,
```c
int x1 = 010;  // 8
int x2 = 0x10;  // 16
```
which is useful when a program requires us to specify a bit pattern, a particular sequence of bits to be interpreted as an integer.

**Converting a Quantity To Binary**

When working in another base, in particular base-2, we will frequently have a quantity in mind and need to convert that quantity to a representation in this base. To derive an algorithm to do this, we utilize the summation of powers formula we used in the previous section to calculate the value of a particular numeric string. Generalizing the formula to an arbitrary binary string in base-2, a binary string composed of digits \( \cdots n_3 n_2 n_1 n_0 \) has the value:

\[
\cdots + n_3 \cdot 2^3 + n_2 \cdot 2^2 + n_1 \cdot 2^1 + n_0 \cdot 2^0.
\]

If a number can be rewritten as a summation of powers in this form, we can simply read off the each of the digits to obtain a final string. In binary, this amounts to discovering how to decompose a quantity into a sum of powers of 2.

For example, consider the number 193. To convert this into binary, we find the largest power of 2 that fits in 193 which is \( 2^7 = 128 \). We subtract this value from the original quantity to obtain \( 193 - 128 = 65 \) and repeat the process. The largest power of 2 that fits in 65 is \( 2^6 = 64 \) leaving \( 65 - 64 = 1 \) left over. The next largest power of 2 that fits is \( 2^1 = 1 \) leaving \( 1 - 1 = 0 \). Once we have reached zero, we are done. Finally, we note all the powers of two that we subtracted from our quantity and plug them into our formula, filling in 0s whenever we did not use that power of two in the process:

\[
193 = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0
\]

Now we can read off the digits, producing the binary number “0b11000001”.

In summary, the algorithm to convert a quantity to binary proceeds as follows:

1. Repeatedly subtract the largest power of 2 that fits in the target quantity until you reach 0, noting each of the powers you use in the process.
2. In the resulting binary string, the positions corresponding to these powers become 1s; the remaining positions become 0s.

Integral Operations

Moving from a traditional number system to binary required changing the base-representation of written numbers. However, this is the only thing that fundamentally changes with respect to the number. The methods by which we carry out primitive operations over integers—addition, subtraction, multiplication, and division—proceed in the same way in binary as they did in decimal!

To illustrate this point, consider adding together five to itself. In grade school, you learned how to perform this addition by lining the numbers on top of each other and adding each column of digits from right to left (i.e., from least-significant to most-significant digit):

\[
\begin{array}{c}
5 \\
+ 5 \\
\hline \\
\end{array}
\]

Because \(5 + 5 = 10\), we put the 1s digit as the result of the column (0) and carry over the tens digit into the next column. We continue the process until we have processed all the columns.

\[
\begin{array}{c}
1 \\
+ 5 \\
\hline 10 \\
\end{array}
\]

Binary addition proceeds in the same manner except that rather than working with the digits 0–9, we are working with just 0 and 1. This greatly simplifies the math performed at an individual column at the cost of having to deal with many more columns than before.

For example, the binary expansion of 5 is “0b101”. To add “0b101” to itself, we set up the columns like before:

\[
\begin{array}{c}
101 \\
+ 101 \\
\hline ? \\
\end{array}
\]

We add up columns like before except now rather than carrying when the result is greater than ten, we carry when the result is two (“0b10”):

\[
\begin{array}{c}
1 \\
+ 101 \\
\hline 1010 \\
\end{array}
\]

The final result is therefore “0b1010” which is the binary representation of 10.

Subtraction works similarly except that we borrow from greater columns if subtraction in the current column would result in a negative result. For example, subtracting “0b01” from “0b10” yields the work:
Note that in base-2, the borrowed amount from the greater column is the quantity 2, just like how the borrowed amount in base-10 would be 10.

Finally, multiplication and division behave similarly in base-2 as well. With multiplication, we multiply columns and then add the results:

\[
\begin{array}{c}
1 \\
\times 1 \\
\hline
0 \\
\end{array}
\]

Or in other words, \(2 \times 2 = 4\). Finally, the standard long division method also works over binary numbers which operates by performing repeated subtractions of the divisor from the dividend.

Handling Negativity

So far, our integers have conveniently been positive quantities. What happens if we need to represent negative values? In written notation, this is as simple as adding a negation sign, e.g., “-0b10” is the quantity -2. However, when representing a negative value in a program, we don’t have the luxury of simply writing down an additional symbol since all we have to work with are bits (0s and 1s).

Recall that integer values in C are fixed size, that is, each number has a fixed number of bits dedicated to it. For example, each value of char type uses 8 bits (1 byte) in memory. One straightforward solution is to simply dedicate one of these bits to represent the sign of the integer, usually the left-most (most-significant) bit. Traditionally, an integer with a sign bit of 0 is positive and a sign bit of 1 means the number is negative. In this system, the value 5 would have the binary representation “0b00000101”, and the value -5 would have the binary representation “0b10000101”.

Note that by dedicating the left-most bit as the sign of the integer, the magnitude of representable numbers is smaller. The largest such number in this system occurs when all the bits are 1 except the sign-bit:

\[0b11111111 = 127\]

The smallest number occurs when all the bits are 1, including the sign-bit:

\[0b11111111 = -127\]

This is a reasonable first cut at representing negative numbers, but there are some fundamental problems with it. In particular, addition of positive and negative numbers is no longer straightforward, e.g., adding a 3-bit representation of 2 (“0b010”) and -2 (“0b110”) should be 0, however:

\[
\begin{array}{c}
0 \\
+ 1 \\
\hline
?
\end{array}
\]
It isn’t clear how to make zero pop out of this calculation. Naïve addition yields the binary number “0b1000” which is not at all related to zero. One can imagine detecting if the sign bit of the second number is one and if so, flip the sign bit and subtract instead. However, this is much more effort than necessary to make addition work. In particular, a computer performs trillions of such calculations in a single second; adding any overhead to the process will heavily affect the performance of the machine.

Another more subtle issue is that zero now has two representations: “0b10000000” and “0b00000000”. This does not seem like it is an issue. However, note that if we want to compare a value with zero, we now need to compare it against two bit patterns instead of one. And comparisons with zero happen very often in computer programs; for example, many CPUs have specific instructions for doing conditional jumps if a target value is zero.

In contrast, the most common representation today of negative integers, twos complement, avoids these issues. To obtain the twos complement of an integer of \( k \) bits, subtract that integer from \( 2^k \). Equivalently (and more efficiently), flip the bits of all the digits—0s become 1s and 1s become 0s—and then add one.\(^3\)

For example, consider the quantity 119 represented as an integer with eight bits:

\[
01110111.
\]

\(-119\) is obtained by first flipping all of the bits:

\[
10001000
\]

and then adding one:

\[
10001000 + 1 = 10001001.
\]

Thus, in twos complement, the quantity \(-119\) is represented by the binary number “0b10001001”.

Using twos complement, addition is simply adding two numbers together without consideration of whether the second is negative. For example, adding 2 (”0b010”) to \(-2\) (”0b110”) and throwing away the overflowing carry value yields zero as desired.

\[
\begin{array}{c c c}
0 & 1 & 0 \\
+ & 1 & 1 & 0 \\
\hline
0 & 0 & 0 \\
\end{array}
\]

Subtraction is also simple in the twos complement system: complement the second operand to negate it and add. Even though the system does not explicitly designate it as such, the left-most bit of the integer effectively acts like the sign-bit in twos complete: if it is 0, the integer is positive and if it is 1, the integer is negative. Finally note that there is no dual representation of zero in twos complement. Flipping the bits of zero (from “0b0000” to “0b1111”) and adding one yields zero again!

\(^3\)Without adding one, the procedure of flipping all the bits of the integer gives the ones complement of the number. Ones complement fixes some of the issues with the simple signed-bit representation, in particular, there is a sensible way to perform addition of negative numbers in ones complement. However, ones complement suffers from other issues, e.g., two representations of zero, that twos complement solves.
Sizes and Limits in C

With this description of how integers are represented we can now understand the limits of integers and their operations in C. For example, the `char` type is an integer value of size one byte or eight bits. In two's complement, the maximum value of a `char` still corresponds to the binary number that contains all 1s except in the most-significant position:

\[ 0b01111111 = 127 \]

If we overflow this number by adding 1, we obtain the bit pattern \texttt{"0b10000000"} which corresponds to the largest negative number possible using the two's complement representation: \(-128\). Note that we recovered the negated quantity by reversing the two's complement process: subtract one (\(0b10000000 - 1 = 01111111\)) and then flip the bits (\(0b10000000 = 128\)). This representation is why in C we see that we go from the largest possible number to the smallest possible number when we overflow.

The maximum and minimum value of an `int`, 2147483647 and \(-2147483647\), are obtained by a similar analysis. In general, if an integer type has \(k\) bits, then the maximum possible value is given the formula:

\[ 2^{k-1} - 1 \]

For each of the \(k - 1\) bits dedicated to the number (the last bit is the effective sign bit), there are two possible values, 0 and 1. Therefore, the total possible values that can be represented by \(k - 1\) bits is \(2 \cdot 2 \cdot \ldots \cdot 2 = 2^{k-1}\). The minus one comes from the fact that zero takes up one of these values. The minimum possible value is given by the formula:

\[ -(2^{k-1}) \]

Because there is only one representation of zero with two's complement, we can represent the entire range of negative numbers afforded by the \(k - 1\) bits.

In many cases, however, we don’t care about representing negative numbers and would like to use the final bit to increase the range of positive numbers we can represent. For example, the standard interpretation of a `char` is a character value, a mapping from a non-negative integer to a character defined by some standard such as the ANSI or Unicode standards. In this situation, we do not need negative values, so it is appropriate to mark such a type as `unsigned` to denote this fact:

```c
unsigned char ch = 'a';  // character value 97, 0b01100001
```

Note that this does not change the eight bits that constitute the `char`. It merely changes how the program interprets those eight bits—rather than treating the left-most bit as a sign-bit, it is part of the number instead.

Bitwise Operations

Normally, a programmer does not need to worry about the low-level details of how integers are represented in their programs. However, when they do, it is usually because they have identified
a need to manipulate data at the byte level. To this end, C provides a number of bitwise operators that manipulate data at the bit-level.

**Bit Shifts** One sort of operation we would like to do over a datum interpreted as a binary string is to *shift* the digits of the string left or right. The *left-shift* binary operator, written \( x \ll n \) takes a value \( x \) (interpreted as a binary string) and an integer \( n \) and shifts the bits of \( x \) \( n \) places to the left. For example, if \( x \) (of type char) contains the value 97 (in binary “0b1100001”) and \( n \) is 3, then the left shift operator proceeds as follows:

\[
\begin{array}{c}
0 & 1 & 1 & 0 & 0 & 0 & 1 \\
\ll & & & & & & \\
0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
\]

Every digit is shifted three places to the left. Note that digits that fall off the left end are lost, in particular, the two 1s in the higher positions of the string.

The *right-shift* binary operator, written \( x \gg n \) is similar but performs a right-shift. For unsigned values, the behavior is also analogous to left-shift. For example, the expression \( x \gg n \) where \( x \) contains 97 and has type unsigned char and \( n \) is 3 operates as follows:

\[
\begin{array}{c}
0 & 1 & 1 & 0 & 0 & 0 & 1 \\
\gg & & & & & & \\
0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}
\]

Note that the least-significant 1 was right-shifted out of the string.

The behavior of right-shift is markedly different if the type of \( x \) is instead int (not unsigned). There is a question of what digits we use to fill in the left-most positions of the result. Suppose we are shifting the following values:

\[
\begin{array}{c}
1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
\gg & & & & & & & \\
0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}
\]

If we perform a naïve right shift, filling the most-significant positions with zero, the integer is no longer negative:

\[
\begin{array}{c}
1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
\gg & & & & & & & \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
\]

This is called a *logical shift*. In contrast, an *arithmetic shift* uses the value of the sign bit to fill the positions:
In C, the behavior of right-shifting a signed datum is undefined, that is, the compiler is free to decide what sort of behavior it implements. Most compilers will choose to perform an arithmetic shift, preserving the sign bit. However, if a programmer writes code that uses bitwise operators, they have to take care to avoid these corner cases if they want their code to be portable (i.e., usable across many compilers and operating systems). In general, these bitwise operators only “make sense” when the underlying data are interpreted as unsigned values, so a programmer should ensure that their data is marked as such before performing bitwise operations.

**Bitwise Logical Operators**  In addition to shifts, C allows programs to perform logical operations at the bit level on data. The simplest of these operators is bitwise NOT or complement. Complement, written \( \sim x \) flips the bits of \( x \)

\[
\begin{array}{ccccccccc}
  1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
  \sim & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

The bitwise AND operator, written \( x \& y \), takes two integers and compares their corresponding bits, position-wise. A bit in the output of the bitwise AND operator is set to 1 if and only if the corresponding bits in \( x \) and \( y \) are also set to 1.

\[
\begin{array}{cccccc}
  1 & 1 & 0 & 0 & 1 & 0 & 1 \\
  \& & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
  & & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\end{array}
\]

In contrast, bitwise OR, written \( x | y \), sets a bit in the output to 1 if one or both of the corresponding bits in \( x \) and \( y \) are also set to 1.

\[
\begin{array}{cccccc}
  1 & 1 & 0 & 0 & 1 & 0 & 1 \\
  | & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
  & & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\
\end{array}
\]

Finally, the bitwise XOR or exclusive-or operator, written \( x ^ y \), sets a bit in the output to 1 if exactly one of the corresponding bits in \( x \) or \( y \) are also set to 1.

\[
\begin{array}{cccccc}
  1 & 1 & 0 & 0 & 1 & 0 & 1 \\
  ^ & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
  & & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

These bitwise operators can be combined to perform more higher-level operations on the individual bits of an integer. For example, suppose we had the following binary number, “0b10010100”, and we wished to see if the fifth bit of the binary number was 1. We can accomplish in a variety of ways. One method would be to shift the fifth bit down into the least-significant position, i.e., right-shift by four.

\[
\begin{array}{ccccccccc}
  1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
  >> & 4 \\
  & & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]
Finally, to clear the remaining bits, we can create a \textit{bit mask} that will zero out all the other digits except the least-significant. Performing a bitwise AND operation with 1 (in binary, “0b00000001”) serves this purpose:

\[
\begin{array}{c}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
\& 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\]

Now the final result is 0 (false) if the fifth bit is set to 0 and 1 (true) if the fifth bit is set to 1. We can generalize this approach into a function that checks to see if the \(k\)th bit of integer \(x\) is set to 1:

\[
bool \texttt{isSet}(unsigned \texttt{int} \ x, \ \texttt{int} \ k) \ { \\
\quad \text{return} \ (x >> k) \ & \ 1; \\
\}
\]

An alternative approach would be to create a bit mask that is all 1s but in the fifth position and then bitwise AND that with the original number:

\[
\begin{array}{c}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
\& 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]

Note that if the fifth bit is set, then the result is non-zero (true); if it is not set, then result is zero (false). To obtain this bit mask, we can simply shift 1 five places to the left:

\[
\begin{array}{c}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\ll & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]

This results in an alternative implementation of \texttt{isSet}:

\[
bool \texttt{isSet}(unsigned \texttt{int} \ x, \ \texttt{int} \ k) \ { \\
\quad \text{return} \ x \ & \ (1 \ll k); \\
\}
\]

Both functions work perfectly fine, although a slight nod goes to the second implementation because it avoids the problems associated with right-shifting and whether the input is unsigned.