Chapter 0, Computers, Bits, and Bytes

John M. Morrison

September 20, 2017

Contents

0 Introduction

We will embark on the study of *computer science*, a mathematical discipline in which we study problems solvable in a finite amount of time with a finite amount of resources. What is interesting to note is the following remark by the computer scientist E. Dijkstra, " Computer Science is about computers like Astronomy is about telescopes." The actual machine is a tool we use to carry out computations; the craft of getting the machine to do this is Computer Science. In the words of Aho and Ullman, Computer Science is the "mechanization of abstraction."

In this chapter we will introduce you to the basic anatomy of a modern computer and then we will introduce you to some fundamental properties of computer science problems.

We shall begin with the machine, since it should be very familiar to you.

Exercises Which of these problems do you think are computer science problems?

1. Take a King James Bible, extract all of the different words from it, alphabetize these, and tell how many times each word is used.
2. Tell if π has the sequence 123456 in the first million digits of its decimal expansion.
3. Tell if π has the sequence 123456 in the digits of its decimal expansion.
4. Compute the entire decimal expansion of $\sqrt{2}$.

1 A Simplified Anatomy Lesson

You have, in all likelihood, worked with some kind of computer. Let us look at the computer from the vantage point of your experience using it.

When you turn the computer on you say it *boots*; this is short for saying that it is going through the "bootstrap process." This process gets its fanciful name because the operating system, the master program that controls the computer's activity, is loaded into memory. So the operating system must, in effect, lift itself up by its bootstraps.

Now we list some things that are clearly present in or around your computer, and classify them as input devices, which get information into a computer, and output devices, which allow the computer to report to you, the user.

- **Input Devices** Familiar examples include the mouse, keyboard, or touchpad. For example, when you strike a key while in an editor window, the character corresponding to that key may appear in a window. Depending on the position of the cursor, your computer will react to mouse clicks, movement or dragging.
- **Output Devices** The most basic output device on your computer is the screen. Speakers and printers are also output devices. Since they are connected to your computer by cables, they are often referred to as *peripherals*; they are, indeed, things around your computer.
- **A Pizza Box** This is the box holding the "guts" of your computer. The name comes from older UNIX workstations that were about the size, shape, and color of a pizza box. Cables would connect the pizza box to the monitor, keyboard, and sometimes the mouse. If you use a laptop, the contents of the pizza box are shoehorned into the small space inside of the computer's case under the keyboard. Our next order of business will be to take a simplified look at what lurks inside of it.

Computers receive, store, process, retrieve, and report information for their users. For this purpose, they need memory in which to store that information.

Your computer has several major types of memory, but let us discuss the two most familiar ones first. Random access memory (RAM), or main memory, gives rapid retrieval for performing immediate tasks. It is the memory in which all programs on your computer are run. RAM is fast but it is dependent on having a continuous power supply.

The *hard drive* is for archival storage. The hard drive or disk drive is where files and programs are archived for permanent storage. The hard drive is a magnetic medium in which data is stored long-term on your computer. These data remains stored, even when no power is being supplied to the system.

All programs on your computer run on RAM. When you start a program for the first time in a computing session, it is copied from the hard drive into RAM. This process is called *loading* the program. If you open a file it is copied, or loaded, into RAM as well.

When your computer is turned off or *booted down*, the various programs running on your computer end their execution. The operating system then shuts down, and the contents of the RAM are obliterated. RAM maintains itself electronically; when the power is gone, the things it stores are lost.

The hard drive is a magnetic medium, on which data can recorded and subsequently read, much as music is recorded on a CD or a cassette tape. Data remain on your hard drive even when your computer is booted down.

To fully understand how RAM is organized, we must first learn a little about the representation of numbers. This is the key to understanding how numbers and other data are stored in a computer's memory. We shall explore number representations in some detail, and then then return to how RAM works toward the end of the chapter.

Data recorded in a computer are of little use, unless a computer manipulates them. This job is done on the *motherboard*, which contains some critical components we shall list and describe here.

- **Arithmetic-Logic Unit (ALU)** This unit does arithmetic operations and handles logical operations. It is the computer's integer (whole number) calculator. It is an integrated part of the CPU on all modern computers.
- **Central Processing Unit (CPU)** This is the control center of the computer; all processing on the computer happens through the CPU. A computer may have several CPUs, all of which may handle computational tasks concurrently.

All of the major components of your computer communicate via a system of cables called the *bus*. Data are sent along the bus as they move between the CPU, the hard drives, and the RAM. Physically, the bus looks like a thin piece of grey lasagna noodle.

1.1 The Memory Pyramid

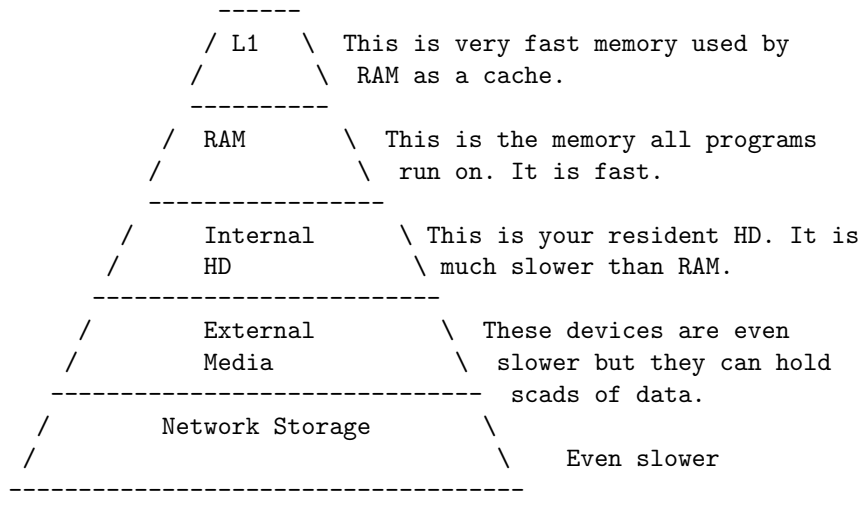
Let us take a moment to discuss the memory of a computer in a little more detail. There are actually several layers of memory; we shall visualize this with the "memory pyramid;" memory at the top is faster, scarcer and more expensive. Memory at the bottom is slower, cheaper and more abundant. Roughly speaking, a layer of difference in the memory pyramid represents an order of magnitude of difference in speed or or cost.

Computers are engineered use a technique called *caching* to temporarily store frequently needed data in a place where the data can be rapidly retrieved. This speeds performance. Each layer of memory uses the one above it as a cache.

At the top of the pyramid is the `L0` cache; this consists of the registers for the chip, which are high-speed memory used directly by the CPU.

All programs run on RAM, but some of their work is farmed out to cache memory, which consists of faster layers of memory than RAM. We show the pyramid here and the various layers of memory.

```
 /\ This is super-fast cache memory
/L0\ used by the L1 Cache. (aka: registers)
```



Notes

1. You can learn more about the inner workings of computers from [10].
2. A fairly sophisticated treatment can be found in the early pages of [1]. This book discusses the memory pyramid in greater detail.

Some Questions to Research

1. Consult the article [6]. Read the section on Mac ROM (read-only memory). Why was Mac ROM important to its graphical user interface?
2. What is BIOS on a PC? What is its importance to to the bootstrapping process?
3. Learn more about caches. How do caches interact with the hard drive, RAM and the CPU?

1.2 Our Tool Set

We shall carry out our investigations on various types of computing technology. The technology we use is a mere detail; it is effective insofar as it does not obstruct our labors nor obscure our understanding of the problem we are tackling. In the beginning, we will construct a very crude model of computing, and we shall refine it as we as we learn more. We need to ask some fundamental questions: What is a computer? How does it "remember" stuff? How are things like characters and numbers represented in the machine's memory?

You will need tools for your explorations. In the next chapter, you will learn how to use Linux, an *operating system* for a computer; an operating system is the master program that controls the activities of your computer. Windoze, MacOSX and Linux are commonly-used operating systems. You may have seen older machines with DOS (disk operating system), Mac Classic operating system, or an Apple 2E.

Today there are many ways to gain access to LINUX that are free and which have considerable support and documentation. The reader is encouraged to check these out and to put one on his machine. This book is primarily geared toward programming in a LINUX environment. You can use it on Windoze, but you will miss some good stuff. Some of this good stuff can be found if you install PowerShell on your Windoze box. Starting with Version 10, Microsoft will have an experimental BASH shell, which accepts UNIX commands. Ubuntu, Fedora and LINUX Mint are all excellent, freely available LINUX distributions. You should Google these and learn about them. It is possible to place one of these on a USB memory key and boot off of it.

In Ubuntu you can do an install called a Windoze install that allows LINUX to run happily alongside Windoze with a minimum of complexity. You can also run a virtual machine using VMWare or VirtualBox. These things can easily be sought out by searching the Web.

Your school, university or company may have a Linux computer; you can inquire about getting account so you can use it.

2 Computer Science

Prepare for a radical shift in direction. What we have discussed so far is very nuts-and-bolts and very concrete. It was about things. Now let us look at some problems of computer science. This is about ideas.

One central activity in computing is searching. Let us take a very prosaic example: looking up an entry in a phone book. Think big phone book: f'rinstance, the Manhattan phone directory. Here is a sure-fire strategy. Suppose you are given a name such as Ada Lovelace. We know she would be listed as `Lovelace, Ada`. We also know the phone book places its names in alphabetical order, primarily by last name and then by first name or initial. Therefore we can create this set of instructions for doing the job.

```
Open the phone book to the beginning of the listings.
While a name is not far enough into the alphabet:
    Look at the next name.
    If that name is the one you seek:
        report the phone number attached to it.
        Stop.
```

```
    If that name is beyond the one you seek alphabetically:
        report that this person is not listed:
        Stop.
    Restart this subprocedure.
```

This procedure will indeed work. What is its shortcoming? If you are looking up Antoni Zygmund, you will have to scan practically the entire book! On average, we'd expect to look through about half of the phone book. The cost of this search is proportional to the size of the phone book. You might ask yourself: is this reasonable? It might be OK for a list with ten names in it, but for a million it becomes cumbersome. If the phone book has n entries, and the cost of searching using this algorithm it is at worst proportional to the book's size, we say that this is an $O(n)$ procedure, or a *linear-time* procedure. We do so because it resolves in a time that is, at worst, proportional to n, the number of objects we are searching through.

A procedure such as this one is called an *algorithm*. Algorithms have these characteristics.

1. Algorithms convey unambiguous, complete and precise instructions on what to do at every step.
2. Algorithms halt with a result in a finite amount of time.
3. Algorithms require a finite amount of memory (stuff on paper if you will) to do their jobs.

This makes them (theoretically) feasible on a sufficiently powerful computer.

This algorithm we just described takes poor advantage of the fact that the directory is sorted. It only does this by throwing in the towel after the name you seek is passed in the alphabet. You spend a lot of time scanning irrelevant stuff. Now let's try this idea on for size, a *divide-and-conquer* strategy.

```
Go halfway into the phone book.
Look at the name there.
if the name is beyond the one you seek:
    Disregard the second half of the phone book.
if the name is before the one you seek:
    Disregard the first half of the phone book.
if the name is the one you seek:
    Report the phone number
Repeat this until a single name remains.
if the name is not the one you seek:
   Report the person unlisted
```

Here, we chop out half of the search field at each step. Suppose you start with 1,000,000 names. Then with each chopping, the size of the search field goes as follows.

```
1000000
500000
250000
125000
62500
31250
15625
7813
3907
1954
972
486
243
122
61
31
16
8
4
2
1
```

So this will, in the worst case, resolve in 20 operations. That is a lot better than half-a-million. If you look at the column of numbers, you can see that it seems to decrease in number of digits at a nearly linear rate. The search field's size goes down by a factor of two in each iteration of the process, so the cost is at most proportional to the logarithm (specifically $\log_2$) of the number of entries in the phone book. We say that this algorithm is $O(\log(n))$. Were our phone book to have a billion entries, we would use about 30 checks to locate an individual. To convince yourself, keep dividing 1,000,000,000 by 2 (round up if the division is not even to be conservative) and you will see this.

In another example, let us count the number of people in a room. You could walk through, counting one at a time. You can see that this is a $O(n)$ procedure; the time to count the bodies is proportional to the number present.

Instead, you could do this.

```
Everyone in the room stands up.
Each person remembers the number 1 at the beginning.
Choose any standing neighbor, if one is not available
    wait for the next round
Ask your neighbor his number and compute the sum
    of your number and his.
One of the two of you should sit down
    if you stay standing your new number is that sum.
The last person standing will know how many people are
```

`in the room.`

The number of people standing at the end of each round will be roughly half that of the number of people standing at the beginning. This will require $O(\log(n))$ rounds.

You can see that some smart thinking can produce a faster procedure. The creation of algorithms is a big area of study in Computer Science.

Notes

1. The book [3] is an important work on the area of algorithms. It's big, white (Hello, Mr. Melville!) and well worth spending some time reading.

Exercises

1. Suppose you are given two numbers n and t. Then execute these instructions. The left arrow shown below means, "store the stuff on the right under the name `x`.

   ```
   x <-  n/2
   while | x*x - n | > t:
         x <- (x*x + n)/(2*x)
   ```

   What is the result? What can you say about n and t. Experiment with several values (make t a small number and use familiar small integers for n).

2. Here you will create a what-if experiment by using a spreadsheet. This exercise assumes very little knowledge of spreadsheets. You can use Libre Offices's or OpenOffice's Calc program, or Microsoft Excel. Open a spreadsheet program. Type these things into the following cells.

`A1`	2	Put the number you are experimenting on in this cell
`A2`	= `A1/2`	Divide your first number by 2, as directed by `x <- n/2`.
`A3`	= `(A2*A2+ \$A\$1)/(2*A2)`	The$s in `$A$1` indicate an absolute position in the spreadsheet.
`B2`	= `abs(A2*A2 - $A1)`	This is the quantity after `while`. We watch for it to drop below `t`.
`C1`	.01	This represents the number `t`.
`C2`	= `B2 < $C$1`	This is the test after `while`.

   Now go to cell `A3` and copy. Paste the contents into the next 15 or 20 cells below. Go to cell `B1`, copy it and paste it into the cells below so you have two lists of numbers in the first two columns of the spreadsheet. The number in `A1` is `n`. Now copy the stuff in cell `C2` into the cells below it. Change `n` and watch the spreadsheet change in response to your work!

3 Numbers and their Representation

Suppose we have two finite collections of objects A and B. What does it mean for them to have "the same number of elements?" Can we determine this without counting them? The answer to this question is, "yes." The test works as follows. Pair one element from each set until you have no leftovers in one of the sets. If the other has no elements too, the sets have the same size. This is called creating a *one-to-one correspondence* or a *bijection* between the two sets. We say that such sets have the same number of elements.

We see that we can do this with any two finite sets of items. This is the basis for the idea of number. We declare two sets to be "of the same size" if they pass this test of having a one-to-one correspondence between them.

Separate from this is the process of *representing* numbers with symbols. Let us discuss several ways of representing numbers. The simplest is just with tally marks. You make one tally mark for each item present. To count the number of eggs in a standard carton, you would make the symbol

```
IIIIIIIIIII
```

These symbols have some advantages. Addition is easy. Just glue together (concatenate) the globs of `I`s. It is pretty easy to multiply too. If you think for a minute or two, you can devise a very simple scheme for doing this. However, the tally system rapidly becomes cumbersome if you want to keep track of hundreds of items. It is simple but bulky, error-prone and cumbersome. It would work for a pre-agricultural society that only keeps track of a few things at a time.

The Romans improved on this with a denominational number system. They had the alphabet {`I, V, X, L, C, D, M`} of symbols. These represent, respectively, the values 1, 5, 10, 50, 100, 500, and 1000 in our number system. They also used a borrowing convention. For example, to represent the number 9, they used IX (borrow 1 from 10). Arithmetic in this system is pretty clunky. With a little imagination, you can figure out how to add and multiply.

The Mayans created a denominational number system based on the numbers 1, 5, and 20. The numbers 1–4 are represented by a row with that number of dots. A 5 is represented by a horizontal bar. The numbers 6–9 are represented by a bar (5) underneath an apprpriate number of dots. Then 10 is two horizontal bars. Bars are stacked and dots are put on top to represent numbers less than 20. The Mayans had a zero too; it looked a little like a football. You can see more detail here [4].

Notice that modern currencies use a denominational number system, so you do not have to carry a Kansas City roll of bills to pay for things.

Exercises

1. Convert the Roman Numeral `MCMLVII` to a decimal number.
2. Convert the number 4096 to a Roman Numeral.
3. Convert your birth year and this year into Roman Numerals. See if you can subtract them directly using only Roman numerals and get your correct age.
4. Write a procedure to decide if a string of characters is a valid Roman numeral.
5. In how many ways can you change a dollar using just dimes, nickels, and pennies?

Notes

1. The article [5] discusses the history of our modern numbering system.

2. The article [8] discusses the history of numeration systems and gives a nice bibliography of further references.

4 Number Bases Computers, and Algorithms

A computer's memory is essentially a gigantic collection of switches called *transistors*; these switches have two possible states: "on" and "off." We can label these states with numbers: 0 for "off" and 1 for "on."

Here is the shocking central fact of life: all information on a modern digital computer is represented by a collection of 0s and 1s!

The informational value of a single switch is called a *bit*; this name is short for "binary digit." A bit can store the value 0 or 1. There is one other unit that is commonly used, a `byte`; one byte is eight bits. It is the information that can be stored in a sequence of eight bits. Bytes look like this: 00110001. The next question is: How do we represent numbers this way?

First, it's time for a trip back to grade school to remember how we *parse*, or impose meaning upon, numbers. When you see the number 256, you do not read it as a sequence of characters 2, 5, 6; it is something more. Our numbering system is based on *place value*; a digit's place in a number determines its contribution to the whole number. The number 256 is a *token* because it is an atom of meaning: If we break it into smaller pieces, such as 25 and 6, it loses its original meaning.

When we see 256, we learned in grade school that the 6 is in the "ones place", the 5 is in the "tens place" and the 2 is in the "hundreds place." To wit,

$$256 = 6 + 5 * 10 + 2 * 100.$$

When reading a number, the last digit has place value one. As you move to the left, the place value of a given digit's value goes up by a factor of 10. This is the familiar *decimal* system of numbers

Why 10? The answer is simple: humans have 10 fingers so the number 10 is a basic unit of counting. Computers do not possess fingers: their bits understand 0,1; "off," "on." We have an *alphabet* of digits 0-9 for our *base 10* or *decimal* arithmetic. You can learn about the history and origins of this alphabet of symbols in this article [5].

Computers represent numbers in a similar manner using the alphabet $\{0, 1\}$ of symbols.

We will use a subscript to indicate a base; writing 256_{10} indicates that we are parsing 256 in the manner we just described for our familiar number system. Computers conduct arithmetic in base 2; the base is the size of the alphabet of

symbols used to represent numbers. For example the base–2 number 11101 is parsed as follows. The last digit is the ones place; as you move to the left the value of a digit doubles. Here is a sample calculation.

$$11101_2 = 1 + 0 * 2 + 1 * 4 + 1 * 8 + 1 * 16 = 29_{10}.$$

We will henceforth refer to base 10 numbers as *decimal* numbers and base 2 numbers as *binary* numbers.

4.1 Computer Science Again!

Now we are confronted with a new computer science problem: how do you translate between base 10 and base 2 representations of numbers?

We need an algorithm to achieve this end. This seems pretty easy, since we can crib from the example we just did showing

$$11101_2 = 29_{10}.$$

Before tackling the main chore, we make a brief but important digression that will give us the tools we need to do the job. We will allow ourselves to create names called *symbols* under which we can store data. Think of the symbols as little sticky–note labels. We are allowed to create a new datum and to redirect an existing symbol to point at it or create a new symbol to point at it. The process of assigning a new value to a symbol, is called, simply enough, *assignment*. When you do an assignment, you are no longer pointing at any old value. It is lost unless you first attach it to another symbol.

To perform an assignment we will use the $\leftarrow$ notation. For instance `x` $\leftarrow$ `5` means assign 5 to the symbol `x`. Observe that this is an inherently asymmetric operation; on the left side goes the symbol which will point at the information. On the right side, goes an expression which is evaluated and given to `x`. The left-pointing arrow $\leftarrow$ should remind you of that asymmetry.

Let us begin by creating a tabular method for this conversion. You will see that this procedure is simple and visual.

The allowable input to our algorithm is a base 2 number. This consists of a When writing down an algorithm, it is important to specify the allowable inputs to the algorithm. The sequence 2100011 is not an allowable input, since it contains the character '2', which is not a part of the alphabet. Now let's abstract what we did in the sample calculation. contiguous sequence of characters (a "string") from the alphabet $\{0, 1\}$.

Suppose we have the binary number 1000111100 that we wish to convert to decimal. Make a table as follows. Put your number in the top row like so.

1	0	0	0	1	1	1	1	0	0

Next, put a 1 at the end of the second row.

1	0	0	0	1	1	1	1	0	0
									1

Populate the second row by doubling the entry you see in the cell to the left until you run out of cells.

1	0	0	0	1	1	1	1	0	0
512	256	128	64	32	16	8	4	2	1

In the last column, multiply the two numbers above that number and add it to the running total, which starts a 0.

1	0	0	0	1	1	1	1	0	0
512	256	128	64	32	16	8	4	2	1
									0

The running total is just the last number you wrote into the table. Now let us take care of the next column to the left.

1	0	0	0	1	1	1	1	0	0
512	256	128	64	32	16	8	4	2	1
								0	0

The next column gets a 4, since the entries above it have product 4 and the running total is 0. Once this step completes, the running total is 4.

1	0	0	0	1	1	1	1	0	0
512	256	128	64	32	16	8	4	2	1
							4	0	0

Now we keep going

1	0	0	0	1	1	1	1	0	0
512	256	128	64	32	16	8	4	2	1
572	60	60	60	60	28	12	4	0	0

The desired number is in the lower left cell of the table. One thing we did without thinking of it is that we need to keep track of where we are in the process. Let us add a row to the top of the table for that purpose.

9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	1	1	0	0
512	256	128	64	32	16	8	4	2	1
992	480	240	120	60	28	12	4	0	0

Now give each row a title.

9	8	7	6	5	4	3	2	1	0	`loc`
1	0	0	0	1	1	1	1	0	0	`digit`
512	256	128	64	32	16	8	4	2	1	`place`
992	480	240	120	60	28	12	4	0	0	`sum`

Programming Exercises

1. Convert the decimal number 121 to a binary number.
2. Convert the binary number `0b111011101` to a decimal number.
3. Convert the hex number 0x169 to binary. Is the result a perfect square? Do you have a reason why or why not?
4. Convert the octal number 0o221 to hex.
5. Suppose the base b of a number is unknown but that we have

$$164_b = 59.$$

   What is the base?
6. What is an easy test to see if a binary number is divisible by 8?
7. Is there a simple test to see if an octal number is divisible by 7? How about to see if a decimal number is divisible by 9?
8. Can you fill in this table?

1011110			
	0o213		
		121	
			0xA5d

9. Open your favorite spreadsheet program. Put the rows into columns 1-4. Can you create a spreadsheet that converts the binary number whose digits appear in column 1?

Now we will use a generic language called *pseudocode* to create a compact set of precise, human-readable instructions that is fairly easy to read and carry out. Imagine we have a a binary number. For any binary number, let us assume we have a procedure, `len` which counts the number of digits the binary number has; i.e., if `x` is a binary number, `len(x)` is its number of digits.

So for the purposes of this pseudocode, we denote our binary number by x.

following procedure. You should break out some paper and a pencil and see if you can play the role of the computer and follow along. Just take a piece of paper and make one column for each symbol. As it gets overwritten, put a new number at the bottom of that column. All figures above it are "dead history."

1. Create a symbol `n`
2. Store the length of `x` under the name `n`, `n` $\leftarrow$ `len(x)`
3. Create a symbol we will call `sum`; This keeps track of the running total.
4. Create a symbol we will call `place`, which knows the power of 2 we are working with.
5. Create a symbol we will call `loc`, which keeps track of where we are.
6. Store the value 0 in `sum` and `loc`; our notation for this is `sum` $\leftarrow$ `0` and `loc` $\leftarrow$ `0`.
7. Since $2^0 = 1$, `place` $\leftarrow$ `1`.
8. If the last digit in the binary number is a 1, add 1 to sum. We can also say this with `sum` $\leftarrow$ `sum + 1`.
9. `place` $\leftarrow$ `2*place`; this doubles the place value since we are moving to the left
10. Now move to the next digit to the left.
11. While there are more digits to the left:
    (a) if the digit is a 1, `sum` $\leftarrow$ `sum + place`
    (b) `place` $\leftarrow$ `2*place`
    (c) Move to the next digit to the left
12. The value `sum` contains our translated value. To report it we will say `return sum`.

As we shall see later, computers are very good at carrying out such procedures such as this one, provided we teach them how to do so correctly. This teaching is done using a *programming language*, and the resulting set of instructions is called a *program*.

You should try translating a few values using this algorithm. As you experiment, see if you can answer these questions.

Exercises Here is a little project you can do to try out basic concepts with binary numbers.

1. What can you say about a binary number whose last digit is a 1? a 0?
2. What can you say about a binary number whose last two digits are 0? What if the last 3 digits are zero?
3. Can you think of a general rule for the number of zeroes at the end of a binary number?
4. What are the analogues of these rules for decimal numbers?
5. What do all binary numbers consisting of all ones have in common? Compute the first few of these and see if you can discern a pattern. Can you write down a formula for 1111..... 1 (n 1s)?
6. Can you think of a way to add two binary numbers without converting to decimal representation? Think back to elementary school and Mrs. Wormwood teaching you how to add decimal numbers! The procedure of "carrying" still works.
7. Create a spreadsheet like the one shown that converts the binary number whose digits are listed in the left column to decimal. Note: you never have to directly compute 2 to any power.

	A	B	C	D
1	1	16384	16384	29017
2	1	8192	8192	
3	1	4096	4096	
4	0	2048	0	
5	0	1024	0	
6	0	512	0	
7	1	256	256	
8	0	128	0	
9	1	64	64	
10	0	32	0	
11	1	16	16	
12	1	8	8	
13	0	4	0	
14	0	2	0	
15	1	1	1	

4.2 No Royal Roads!

Do you think that our algorithm is the *only* way to convert a binary number to decimal number? In fact it is not! We will now confect a second procedure to do this job. To make things simple we will allow symbols to store numbers, single characters, or strings (contiguous sequences) of characters. We will use the operator `+` for the concatenation (gluing together) of strings. For example,

```
"some" + "thing" = "something".
```

- Create a symbol called `sum` that stores a number.
- `sum` ← `0`
- while there are digits remaining:
- `sum` ← `2*sum + presentDigit`
- move to the next digit on the right.
- return sum

Let's apply this to the example 11101_2. We will indicate the present digit by bold-facing it.

sum	presentDigit	What is happening?
0	**1**1101	begin here
1	**1**1101	`sum ← 2*sum + presentDigit`
1	1 **1**101	move to next digit
3	1 **1**101	`sum ← 2*sum + presentDigit`
3	11 **1**01	move to next digit
7	11 **1**01	`sum ← 2*sum + presentDigit`
7	111 **0**1	move to next digit
14	111 **0**1	`sum ← 2*sum + presentDigit`
14	1110 **1**	move to next digit.
29	1110 **1**	`sum ← 2*sum + presentDigit`

The procedure ends and returns 29 since we are out of digits! We know our stuff backwards and forwards since we can convert the number using *either* direction.

	1	1	1	0	1
2					

Begin by dropping the lead `1` in the second column of the table.

	1	1	1	0	1
2					
	1				

Now multiply the entry at the bottom of the second column by the entry in the first column and place it in the middle of the next column like so.

	1	1	1	0	1
2		2			
	1				

Now add the contents of the next column and place the total in the bottom.

	1	1	1	0	1
2		2			
	1	3			

Repeat the foregoing procedure.

	1	1	1	0	1
2		2	6		
	1	3	7		

Keep going for the rest of the columns

	1	1	1	0	1
2		2	6	14	28
	1	3	7	14	29

Et voila! The converted value lives in the lower-right hand cell of the table.

Try using both algorithms. Which one do you think is nimbler and easier to execute?

Now that you have seen some algorithms at work, you can see that any algorithm worth its salt must exhibit these properties.

We will also show a simple way to do this using a table. We will convert `11101` using this method. Make three rows and populate the table like so.

- **Repeatability** The instructions you furnish must be precise and the user must be able to carry them out without your intervention. Note that his user might be an unthinking machine! You must also tell what the allowable inputs are and expected outputs are.
- **Finite Resource Restriction** You may only use a finite number of symbols store data. Each datum only has a finite size.
- **Termination Condition** The algorithm must halt after a finite number of steps.

How do you get from decimal to binary? To do this we will learn about two new arithmetic operations for integers; these will be quite helpful in our development. For a non-negative integer `b` and a positive integer `a`, we will define

```
b % a
```

to be the remainder when `a` is divided into `b`. This operation is not defined if `a = 0`, and we will restrict our attention to it for non-negative numbers. Here is an example

```
100 % 3 = 1
```

because we can use elementary school arithmetic as follows.

$$\begin{array}{r} 33 \\ 3\overline{)\,100} \\ \underline{90} \\ 10 \\ \underline{9} \\ 1 \end{array}$$

This long division shows us that $100 = 33 * 3 + 1$.

There is an even easier way to see this. Imagine you have a one-handed "clock" with 3 numbers labeled 0, 1 and 2 and that it goes forward one number each second. Now set the hand at 0 and let the clock run for 100 seconds. The hand will make 33 complete revolutions, than move forward one more number, ending up at 1. You can think of `%` as doing clock arithmetic on this kind of clock.

For a non-negative integer `b` and a positive integer `a`, we define new operation *integer division* which does division of integers discarding the remainder. We denote this by `b//a`. In our example, we will define `100//3 = 33`.

Here is the algorithm. Try converting a number to binary and back to decimal to verify the algorithm is indeed correct. Suppose we have a decimal number named `d`.

```
stub ← d store d in a symbol called stub
buf ← "" make buf be an empty string of characters
while stub > 0:
    digit ← stub % 2
    stub = stub // 2
    buf = digit + buf prepend the digit to the string buf
return buf
```

This algorithm converts decimal to binary. Experiment with converting both ways and see that it works. Here is a sample calculation for 29_{10}. The table given here shows, step by step, what is happening in each symbol as the calculation progresses. Try following it yourself on a piece a paper.

stub	buf	digit	What is happening?
29	""	–	We begin.
29	""	1	digit ← stub % 2
14	""	1	stub = stub // 2
14	"1"	1	buf = digit + buf (append digit to buf)
14	"1"	0	digit ← stub % 2
7	"1"	0	stub = stub // 2
7	"01"	0	buf = digit + buf
7	"01"	1	digit ← stub % 2
3	"1"	1	stub = stub // 2
3	"101"	1	buf = digit + buf
3	"101"	1	digit ← stub % 2
1	"1101"	1	stub = stub // 2
1	"1101"	1	buf = digit + buf
1	"1101"	1	digit ← stub % 2
0	"1101"	1	stub = stub // 2
0	"11101"	1	buf = digit + buf
return "11101"			stub is now zero

Exercises

1. Convert the decimal number 1001 into base 5 adapting the procedures of this section. This adaptation is surprisingly straightforward.
2. Write a spreadsheet that carries out a conversion from base 1 to base 10. Place your binary digits in the first column. Shown below are the values from a conversion program. You must add the right formulae to make it work.

	A	B	C	d
1	1	16384	16384	29017
2	1	8192	8192	
3	1	4096	4096	
4	0	2048	0	
5	0	1024	0	
6	0	512	0	
7	1	256	256	
8	0	128	0	
9	1	64	64	
10	0	32	0	
11	1	16	16	
12	1	8	8	
13	0	4	0	
14	0	2	0	
15	1	1	1	

3. Does the system of long division you learned in grade school work for binary numbers? If so, what is better about it and what is worse? Experiment.

4.3 Hexadecimal and Octal Numbers

These are base-16 numbers; the alphabet of digits is

`{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}.`

The hex digit A represents the decimal number 10; likewise, B represents 11, C represents 12, D represents 13, E represents 14 and F represents 15. It is a nearly universal international convention to prepend hexadecimal numbers with the prefix `"0x"`. For example the hexadecimal number `FF` is written `0xFF`. Place value in hex numbers is 16. In the exposition below, decimal numbers have no prefix. You will often see hex digits represented with lowercase letters. Hex numbers are case-insensitive; that is, the lower case and upper case version of the letters A-F have exactly the same meaning. This usage is a universally accepted convention.

Octal numbers have the alphabet `{0, 1, 2, 3, 4, 5, 6, 7}`. These digits have their usual interpretation. Place value in octal number is 8. There are two notational conventions for octal numbers. One is to prepend a 0, the other is to use the prefix `"0o"` (zero then little o). Using these prefixes instance the octal number 12_8 is written 012 or 0o12.

Let us convert a decimal number to a hex number. We shall start with the decimal number 1000, and use an algorithm similar to the ones shown here.

stub	buf	digit	
1000	“”	–	
1000	“”	8	1000%16 = 8 = 0x8
62	“”	8	1000//16 = 62
62	“8”	8	buf = digit + buf
62	“8”	E	62%16 = 14 = 0xD
3	“8”	E	62//16 = 3
3	“E8”	E	buf = digit + buf
3	“E8”	3	3%16 = 3 = 0x3
0	“E8”	3	3/16 = 0
0	“3E8”	3	buf = digit + buf
return ”3E8”			

We see that 1000 = 0x3E8. Let's check this explicitly.

```
0x3E8 = 3*256 + 14*16 + 8 = 768 + 224 + 8 = 1000.
```

4.4 Woe is me! Why hex numbers?

There is a very cool relationship between hex and binary numbers; the same sort of thing applies to octal numbers as well.

Binary numbers offer a very simple system of arithmetic. Learning the times tables and the basic addition fact is ... not very taxing at all. However, unfortunately, binary numbers get very long very fast. For instance,

$$1000_{10} = 1111101000_2$$

A small decimal number makes an ugly mess in binary. Here is how to convert that nasty binary to a hex number in short order. Break it into chunks of 4 starting with the least significant (last) digit like so.

```
11 1110 1000
```

Each block of 4 represents a hex digit like so

```
11 = 0x3
1110 = 14 = 0xE
1000 = 8 = 0x8
```

Now concatenate the hex digits to see 1000 = 0x3E8.

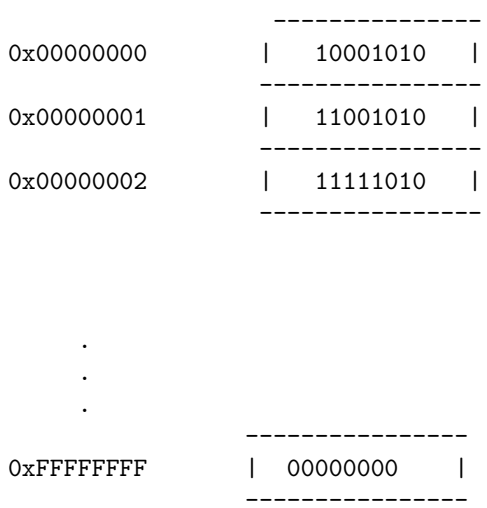
A hex number is easily convertible into binaries and vice versa, and its is one-forth as long. Computers often show hex numbers because they are easier for *humans* to read. To convert back, just expand each hex digit into its binary equivalent and concatenate. Make sure you pad with 0s where necessary!

Exercises

1. Confect and test a rule to convert binary numbers to octal numbers. Test it on the base 10 number 1000.
2. Solve this equation for b.
$$125_b = 68.$$
3. Describe a simple technique for converting hex numbers to octal.

5 Hex Numbers and RAM

Your RAM (random-access memory) is basically like a long paper tape divided into little cells. Each cell holds a byte; notice that a byte is just two hex digits. Every cell in memory has a hexadecimal number which is its address. So, your memory looks like this



This computer would have 4294967296 bytes of memory, or in common parlance, 4 gigs. The hexadecimal number that indexes each byte of memory is called that bytes *memory address*; you will often hear memory addresses called "pointers." A pointer in memory works much like the address of your house. Both describe the location of objects in a space; your address describes your house's location on a map and a memory address describes a byte's place in RAM.

A fundamental operation of programs is to create symbols as we did in the algorithms above and store data in them. These symbols are stored somewhere in RAM. A symbols datum will occupy one or more bytes of memory.

5.1 Deconstruction Time

What is the "random" in random access? This means we can travel anywhere in memory at a fixed rate of computational cost. Whether we move over one byte or a million, the computational cost is the same. We have equal access to all places in RAM from any place in RAM. Access of the contents of a memory address is often called a *constant-time* or $O(1)$ operation.

6 The Representation of Text in Memory

All of the familiar characters on your keyboard have a numerical code called an *ASCII* code. Each character occupies one byte of memory. Here are some ASCII values for familiar characters.

```
A           65          01000001
B           66          01000010
C           67          01000011
.
.


Z           90          01011010
[           91          01011011
\           92          01011100
]           93          01011101
^           94          01011110
-           95          01011111
‘           96          01100000
a           97          01100001
b           98          01100010
.
.
z           122         01111010
```

The characters you type into a text file are actually stored in memory by their ASCII codes. You can observe some features of ASCII codes right away. All lower cases letters are organized consecutively starting at `0x61` and all upper chase characters are stored consecutively starting at `0x41`. All digits are stored consecutively starting at `0x30`.

Notice that every lower-case letter has an ASCII code exactly 32 above its upper-case mate. To change case of a character a"low level" operation can change the 32s bit of the ASCII code. Look at (and). Notice their ASCII codes differ by one bit. Can you find other examples like this?

7 Programs and Files

A file is a repository of data. A file is just a collection of bytes. When inactive, a file is stored in the hard drive. When it is to be used, a file is copied (loaded) into RAM. Files can be data, or they can be programs. Modern computers use the *stored program concept*: the instructions in a program are stored in a file. When any program is run, it is copied into RAM.

The *operating system* is a very complex program that controls the activities of the computer and its hardware. It maintains your file system and schedules tasks. It also manages memory for all of the programs running on your computer. Operating systems may be graphical (MacOS, Windoze, Ubuntu, Fedora) or they may have a command-line interface. (DOS, PowerShell, UNIX in a terminal window) You interact with a graphical interface using the mouse and keyboard. In a command-line interface, you type commands and the computer executes them. We will study Linux as a command line based operating system in Chapter 1.

Now lets put it all together. When you boot your computer, your operating system is loaded into RAM. When this process, called booting completes, your machine is ready to use. Your computer will have programs and data stored into files, which are just collections of bytes. When you run a program, it is copied into RAM into a chunk of memory meted out by the operating system. Files opened by the program are also copied into RAM, and new files are created directly into RAM. When a program needs to make a computation or a logical comparison, the ALU springs into action. When you save information in a file, the file is copied onto the hard disk from RAM. When you quit an application, it closes down and the memory it is occupying is marked for reuse. The operating system controls RAM and it keeps track of the memory addresses of all the files and processes running on your computer, manages the apportionment of memory, and mediates all communication between processes.

8 Integers and Computers

Every operating system has a word size; machines today are 32 or 64 bit machines. What does this mean? The most basic unit of memory in a computer is an integer, or whole number. There are two types of integer, signed integers and unsigned integers. Unsigned integers are always non-negative. Unsigned integers are used to supply memory addresses for your computer's RAM. Signed integers (having a + or − sign) are often used in arithmetic. When the computer puts them to the screen or a file, you see a sign for the integer. When the computer manipulates them behind the scenes, it actually stores them as unsigned integers using a very clever scheme. We will discuss this scheme shortly, but we will begin by looking at unsigned integers.

If you have a 32 bit machine, integers have 32 bits; likewise, if you have a 64 bit machine, integers have 64 bits. A 32 bit unsigned integer can store numbers from 0 to $2^{32} - 1$. This number represents the 4 gig limit for usable RAM on 32 bit machines. A 64 bit unsigned integer can store numbers from 0 to $2^{64} - 1$; this can support memory up to 18 exabytes. Since today's machines are coming with 4 or more gigs of RAM, going to the 64 bit system is necessary. A 32 bit unsigned integer can be represented with 8 hex digits; a 64 bit integer requires 16 hex digits. Unsigned integers are just plain-vanilla binary numbers.

Exercise 1. UNIX operating systems keep track of time using the epoch of the *modern era*, which, it was decided, began on 1 January 1970. Time is stored as an integer in milliseconds from that time. When will this run into trouble by having this integer exceed $2^{32} - 1$ for a 32 bit operating system? Do you think this presents a real and practical problem? How about a 64 bit operating system. A 128 bit operating system?

8.1 Signed Integers on the Cyberdent

Being a computer means never having to subtract. Let's see why. Imagine we are working with the Cyberdent, an 8-bit computer. Its unsigned integers are bytes, so its integer can hold a number in the range from 0 to 255. This will keep our examples small and tidy and avoid us having to write out strings of 32 bits.

It is not a very practical computer, because it can only maintain memory addresses for 256 bytes of memory, and its system clock is in big trouble after about 1/4 of a second.

Let us begin by looking at the addition of unsigned integers on the Cyberdent. We will use *ripple carry addition* for adding two integers; this is just a simple variation on the scheme you learned in grade school. In fact, it is simpler, since we carry if we have two 1 bits in a column and we do not carry otherwise. We will do a simple example here, 7 + 5. Our computer does not see 7 or 5, it sees 00000111 and 00000101. Now let us demonstrate the ripple carry technique.

```
    111        carry space
 00000111
+00000101
---------
 00001100
```

Notice that the binary number 0001100 converts to 12. All works as it should. Now let us add 255 + 1.

```
 11111111       carry space
```

```
11111111
01111111
--------
00000000
```

Uh oh. The lead bit got "carried away;" it has been dropped into the proverbial bit bucket, never to be seen again. The ruthless logic of the adder circuit in the Cyberdent causes us to obtain the implausible result 255 + 1 = 0. This feature is called *type overflow*; when you try to add integers larger than the word size allows, you can get mysterious errors. Computers will happily and mindlessly allow this to happen. It is up to the designers and programmers of computers to be vigilant here.

1. Suppose you have an integer n that satisfies $0 \leq n < 256$. Such an integer can be stored as a Cyberdent integer. Find an integer m so that in Cyberdent addition, m + n = 0, and express it in terms of n.
2. Let m and n be as in the previous exercise. How are the binary representations of m and n related? Describe this in terms a grade school kid might understand.
3. Think about the mod operator % you learned about before. What does Cyberdent addition really do? You can describe it simply using % and +.
4. What is the range of unsigned integers you can store in the Cyberdent? How many memory addresses are possible?

8.2 Signed Integers on the Cyberdent

One obvious scheme for representing signed integers is to reserve the first bit as the sign bit, then let the magnitude of the number be represented by the rest. Let's try this "obvious scheme" on representing an integer; here we will use the example -5. The binary representation of 5 is 101. Since -5 is negative, we color its lead bit 1 and we have

```
10000101.
```

Here is an awkward problem. The integer 0 is represented on the Cyberdent with the byte `00000000`. The integer -0 is represented by `10000000`. Ugh; we have two representations for the same integer! This cannot end well.

Modern computers use a system called `two's complement notation` to represent signed numbers. Because of this extremely clever system, modern computers never subtract; they only add. What is interesting is that they take the type overflow lemon and make some glorious lemonade from it.

As in the obvious scheme shown above, the lead digit is the sign bit; if it is a 1, the number is negative and if it is a 0, the number is non-negative. Instead of

computing 7 - 5 using subtraction, we will first negate the 5 and add the result to the 7. With a little luck, the result is the expected 2.

So we need to figure out how to change the sign of a number in such a manner that we can use ripple-carry addition for subtraction once this job is accomplished. If we accomplish this mission correctly, we will not have to have a separate subtraction circuit for the Cyberdent, and as a result, we will make megabucks.

Let us see if we can find the right way to represent -1. We know that if we are on solid ground, we should get $-1+1=0$. So we produce this little picture.

```
 abcdefgh
+00000001
---------
 00000000
```

We need to pick binary digits for the slots `a-h` in such a manner that the result is zero. We have one ally in the cause: the bit bucket. It will play a key role in the process. What we will do here is to pass the buck to the left until it falls off.

Consider `h`. It cannot be 0 or our last digit is a 1 and we are out of luck. We must have `h = 1`. Let us fill in the clue and look at the carry implications.

```
       1      carry space
 abcdefg1
+00000001
---------
 00000000
```

Now the `g` and the 1 must yield a zero, so we must have `g = 1`.

```
      11      carry space
 abcdefg1
+00000011
---------
 00000000
```

By now you should get the idea that all of `a-h should be 1`. The final carried digit drops into the eminently chivalrous bit bucket. We know `-1` must be represented by `11111111`; we know 1 is represented by `00000001`.

We can use this technique to obtain the results in this table. You should use the technique we just developed to try this for yourself.

-n	-n in two's complement	n in two's complement
-1	11111111	00000001
-2	11111110	00000010
-3	11111101	00000011
-4	11111100	00000100

One thing is clear; 0s are getting flipped to 1s and vice versa. For any integer, we define the `bitwise not operator` by the following procedure. If you have a binary number n, then `~n` is found by flipping all of the 1s in n to zero and all of the 0s to one. For example,

$$\sim 12_{10} = \sim 00001100 = 11110011.$$

If we interpret 12 as an eight bit unsigned integer, $\sim 12 = 243$; as a signed integer it is a negative integer, since the lead bit is a 1.

If we flip the bits in -1, we get 0000000, or 0. If we flip the bits in -1, we get 0000001, or 1. If we flip the bits in -3, we get 00000010, or 2. We can see here an "off by 1 phenomenon." It would appear that, to negate a number, we flip the bits and add 1! Voila. Let us try this on 0.

The binary representation of 0 is 00000000. Flip the bits and get 11111111. Now add 1: the last carry digit goes into the bit bucket and we get 0. Hey, $-0 = 0$, a desired outcome!

Notes

1. The article [9] has an excellent and complete treatment of Two's complement notation.
2. The article [7] discusses the fascinating way in which floating point (numbers with decimal point) are stored and manipulated in modern computers.
3. The book [2] discusses floating point numbers and covers the art of computing in the floating-point domain. If you have an interest in doing mathematical computing, this is a great place to start. As you learn Python, you will be able to write code for many of the algorithms shown in this book.

Exercises In these exercises, you will learn how to subtract without borrowing. Take that, Mrs. Wormwood! Fill in the outline below and the tyranny of elementary school will be broken!

1. If you have a decimal integer, its ten's complement is calculated by taking each digit and subtracting it from 9. We will use the notation for this. For example $\sim 125 = 874$. Take the numbers 32 and 27 and subtract them as follows. Take the ten's complement of 27 and add it to 72. What

should go in the bit bucket? How do you adjust what's left to get the right answer.

2. Tell what to do if the number are of unequal length, such as would be the case for $89 - 452$.
3. See if you can write a little lesson to teach an elementary school kid how to subtract this way.

References

[1] R. Bryant and D. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 2010.

[2] W. Cheney. *Numerical Mathematics and Computing*. Cengage Learning, 2012.

[3] T. Cormen. *Algorithms*. McGraw-Hill, 1999.

[4] Luke Mastin. The story of mathamatics, 2010. `http://www.storyofmathematics.com/mayan.html`.

[5] `http://en.wikipedia.org/wiki/Arabic_Numerals`. Arabic numerals. *Wikipedia*, 2013.

[6] `http://en.wikipedia.org/wiki/History_of_Mac_OS`. History of mac osx. *Wikipedia*, 2013.

[7] `http://en.wikipedia.org/wiki/IEEE_754`. Two's complement. *Wikipedia*, 2013.

[8] `http://en.wikipedia.org/wiki/Numeral_system`. Numeral system. *Wikipedia*, 2013.

[9] `http://en.wikipedia.org/wiki/Twos_Complement`. Two's complement. *Wikipedia*, 2013.

[10] Ron White. *How Computers Work, 9th Edition*. Que, 2007.