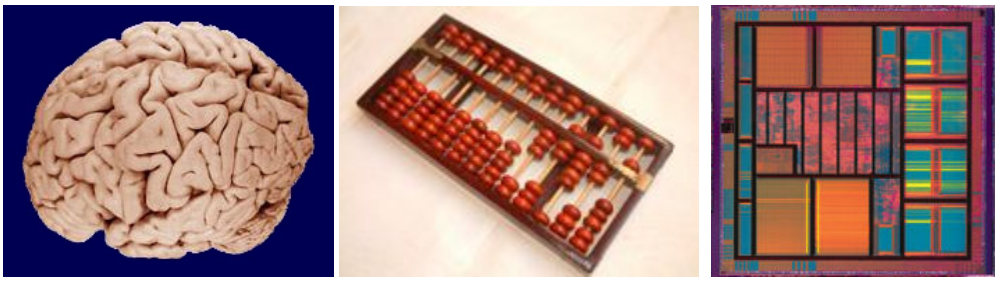
What is computation?



In the 19th century, our society fundamentally changed the way it produced material goods. Whereas production had previously depended on human and animal labor, the steam engine allowed much of that labor to be transferred to machines. This made the manufacture of many goods and services more efficient, and therefore cheaper. It made other kinds of products available that would otherwise have been impossible to manufacture. At the same time, it caused widespread economic dislocation, putting many skilled tradesmen out of work and forcing much of the population to leave their rural communities for cities, radically changing the very environment they lived in. Consumer culture, television, corporations, and the automobile, would not have been possible, or at least as influential, without this automation of production, better known as the industrial revolution.

We now hear we're in the midst of an "information revolution." Like the industrial revolution, it involves automation – the transfer of human labor to machines. However, this revolution involves not physical labor, but *intellectual* labor. While we cannot know the ultimate consequences of these changes, information technology has played an increasingly central role in our society over the last two decades. In the 1950s, if you and your friends wanted to go to a movie, you wouldn't have started by walking to the library. But today you wouldn't think twice about checking the internet for reviews and show times. Similarly, average people wouldn't have published their diaries in the 1950s, but today people blog about everything from international politics to their pets' favorite toys.

Yet for all the importance of these technologies, people still tend to think of computation as being about numbers, and computer science as being about the esoteric worship of acronyms and punctuation.

Computation isn't tied to numbers, acronyms, punctuation, or syntax. But one of the things that makes it so interesting is that, in all honesty, it's not entirely clear what computation really is. We all generally agree that when someone balances their checkbook, they're doing computation. But many people argue that the brain is fundamentally a computer. If that's true, does it mean that (all) thought is computation? Or that all computation is thought? We know that the "virtual" environments in computer games are really just computational simulations. But it's also been argued that the real universe is effectively a computer, that quantum physics

 ian@northwestern.edu

is really about information, and that matter and energy are really just abstractions built from information. But if the universe is "just" computation, then what ***isn't*** computation? And what does it even mean to say that something is or isn't computation?

Computation is an idea in flux; our culture is in the process of renegotiating what it means by the term. One hundred years ago, nearly everyone thought of computation as being a mental operation involving numbers. And being a mental operation, it could only be done by people. Today, the overwhelming majority of what we consider to be computation is done by machines. But at the same time, we still somehow associate it with thought. In Western culture, we tend to take our capacity for thought as the central distinction between ourselves and other animals. Moreover, we view our specific thoughts and feelings as being one of the major constituents of our personal identity. So thought is constitutive both of our collective humanity and of our individual identities.

Changing our ideas about computation changes our ideas about thought and so our ideas about ourselves. It reverberates through our culture, producing excitement, anxiety, and countless B-movies about virtual reality and cyber-thingies.

And yet for all our collective fascination with computation, what we mean by the term is still somewhat mysterious.

Questions and answers

I've said that computation isn't fundamentally about numbers. Nevertheless, let's start by thinking about arithmetic, since we can all agree that it's an example of computation. Suppose you ask someone:

> "What's seven plus three?"

If they reply "ten," then they just performed a computation. So we can provisionally think of computation as a kind of question-answering. In the case of addition, the question always involves a pair of numbers and the answer is always another number. For any particular addition question, there's a corresponding number, determined by the pair of numbers, which is the desired answer.

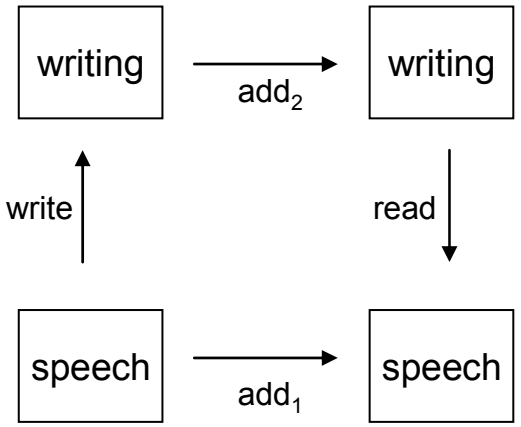
Now suppose you ask them "what's one million, seven hundred eighty-two thousand, six hundred, and seventy-eight plus three million, two hundred ninety-two thousand, seven hundred and four?" Unless they're especially good at mental arithmetic, they won't be able to keep all the numbers in their head and so they'll have to take out pencil and paper, ask you to repeat the question, write it down, perform the addition on paper, and read off the answer.

But now arithmetic is no longer just a mental operation. It's also a physical operation: it involves manipulation and change of physical objects (the pencil and paper). We're used to thinking of the arithmetic we do as being a "mental" thing that takes place "in" our heads. But

 ian@northwestern.edu

in this case, it's spread out between a person's head, hands, pencil, and paper. Another change from the seven-plus-three example is that while we *presented* the numbers to the person as a set of sounds in spoken English, they then *re-presented* them as symbols on paper, presumably Arabic numerals.[1]

What's interesting here is that none of these differences matter. So long as the person comes up with the right answer in whatever representational system or systems they happened to be using, we consider them to have successfully done the arithmetic. Put another way, it doesn't matter which path we take through the diagram below:



We'll call this the principle of **behavioral equivalence**: if a person or system reliably produces the right answer, they can be considered to have solved the problem regardless of what procedure or representation(s) they used. Behavioral equivalence is absolutely central to the modern notion of computation: if we replace one computational system with another that has the "same" behavior, the computation will still "work."

The functional model

Let's try to be more precise about what we mean by saying computation is a kind of question answering. First of all, there are different kinds of computation problems. So it's fair to ask an adding machine what 3+7 is, but not what the capital of India is. The ability to do a task like addition is roughly the ability to answer a certain prescribed set of questions with the correct answer. So we can think of a computational problem as a set of possible questions, each of which has a desired correct answer.

[1] In fact, "the numbers themselves" never even made an appearance, since they're completely intangible; we only have access to them through representations.

 ian@northwestern.edu

In the case of addition questions, the only relevant parts of the question are the numbers to be added. For an adding machine, the "what is the sum of" part of an addition question is irrelevant because that's the only kind of question it can answer. So we'll dispense with the irrelevant parts and just say the question consists of the relevant bits, in this case the pair of numbers to be added. Similarly, we'll distill the answer down to just a number.

So while we still don't know what computation is in the abstract, we can at least say something about what we mean by a computational problem. It's a group of related questions, each of which has a corresponding desired answer. We'll call the information in the specific question the **input** value(s) and the desired answer the **output** value. For any input, there is a corresponding desired output. The notions of input and output are the most basic concepts of computation: computation is – for the moment – the process of deriving the desired output from a given input(s).

If you've taken the right math courses, then you may realize that what we're calling "a computational problem" is the same as what mathematicians call a **function**. A function is just a specification of output values for any given input value(s). Sine and cosine are functions; they specify output values (numbers) for every possible input number. Addition is also a function, but of two inputs rather than one. And more esoteric operations in mathematics, like the derivative can also be thought of as functions, although they're functions whose inputs are themselves other functions. But functions don't have to be about numbers. For example, an MP3 player computes a function whose input is a compressed song and whose output is an audio waveform.

A **procedure** (aka an *algorithm*, *program*, *routine*, or *subroutine*) is a specific method for determining an output value from a set of input values. Assuming the procedure always produces the same output for a given set of inputs (e.g. it doesn't involve rolling dice or other non-deterministic operations), then it "acts like" a function in that it has a strict correspondence between inputs and outputs. The difference between procedures and functions is that functions only specify *what* their outputs are, whereas a procedure specifies *how* to compute them. There are typically many different procedures for computing a given function, but as we shall see, there are some functions we can't compute. We can provisionally think of Computer Science as the study of **procedural knowledge**: how to describe procedures, construct them, and compare competing procedures for computing the same function.

We've been assuming here that the only important part of a procedure's behavior is its output. We'll call this the **functional model** of computation: procedures are ways of computing functions and so two procedures are behaviorally equivalent if and only if they compute the same function (although they may differ in the resources they require such as length of time or amount of scratch paper). This is a limited view of computation, but it covers a surprising amount of ground. Remember that the inputs and outputs of our functions don't have to be numbers. They can perfectly well be things like MP3 files or images from a web cam.

 ian@northwestern.edu

Procedures

Let's consider one last arithmetic example. Suppose you're at a restaurant and want to leave a 20% tip. As a computational problem, this involves taking a number as input and computing 20% of it. If we call the input *c* for "check", then were computing the function:

$$\text{tip}(c) = 0.2 \times c$$

Most people aren't especially good at computing percentages in their heads because it involves multiplying multi-digit numbers; if I asked you what 73% was of 151.05, you'd probably have to take out pencil and paper or use a calculator. But 20% has the useful property that we can use the following procedure:

1. Double the number
2. Erase the last digit
   (assuming the bill and tip are whole numbers)

This is much easier because most people can double numbers of a few digits in their heads, whereas computing higher multiples is harder for them. On the other hand, it's assumes we represent the number as a string of decimal digits so that removing a digit divides the number by ten. If the wait person writes the check in Roman numerals, you're out of luck (quick: what's XX% of MMCMXLIV?).

Representation

In practice, inputs and outputs are always encoded in some specific **representation**. Often, we can ignore which representation is used and just think about the procedure as manipulating the "information" in the input rather than as manipulating the components of a specific representation. Consequently, computation is often referred to as **information processing** and the computing industry as the information technology industry. However, as this example shows, the choice of representation can affect our choice of algorithm. Consequently, Computer Science is also very concerned with studying the properties of different kinds of representations or **data structures**.

Part of the reason representation matters is that it affects what simpler procedures you can take for granted. Procedures are always written in terms of other simpler procedures. In the algorithm above, we assumed the reader already knew how to double a number and how to erase a digit, since those are relatively easy to do in decimal notation. However, if we were teaching it to a child who hadn't learned multiplication, we might instead say:

1. Add the number to itself
2. Erase the last digit

This is technically a different procedure, since it replaces multiplication with addition, but since it computes the same output, it's behaviorally equivalent to the original one.

 ian@northwestern.edu

Now suppose the child hasn't learned to do multi-digit addition, and yet we still want them to compute our tip for us, perhaps as torture of a younger sibling. Then we could say:

1. Start with the last digit
2. Add it to itself
3. If the result is more than ten, then write the 1 above the next digit
4. If there are any digits left
    a. Move to the next digit
    b. Go to step 2, remembering to add in the 1 above it, if you carried a 1 over
5. Otherwise, erase the last digit

This procedure explains computing a 20% tip on a multi-digit number in terms of single-digit addition. But suppose the poor child can't add, but can only count, yet is inexplicably eager to please you, their sadistic older sibling. Then we could tell the poor child to:

1. Start with the last digit
2. Add it to itself by doing the following
    a. Write a 0 underneath it
    b. If the number underneath is the same as the digit that we started with, then go to step e
    c. Count each number up by one
    d. Go back to step b
    e. If there was a carry written above the original digit, then count it up one more time
6. If the result is more than ten, then write the 1 above the next digit
7. If there are any digits left
    a. Move to the next digit
    b. Go to step 2, remembering to add in the 1 above it, if you carried a 1 over
8. Otherwise, erase the last digit

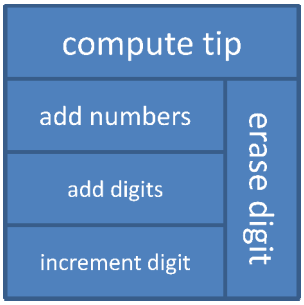
But of course, to do that, you would have to be *really* sadistic.

Levels of abstraction

We described multiplication in terms of addition, which broke down into adding digits, and then counting. In practice, we would make this explicit by describing procedures in terms of ***sub-procedures****:*

- ***To double a number****: add it to itself*
- ***To add two numbers****: add the last digits, then add the next digits, with the carry, etc.*
- ***To add two digits****: repeatedly add 1 to one while subtracting one from the other until the second digit is zero.*

If we were to diagram all the procedures involved, it might look like this:



Most programs break up into layers like this. Procedures are built "on top of" simpler procedures, which usually are built on top of still simpler procedures, until we get down to the level of the ***primitive*** *operations that the computer's hardware can execute directly. However, you can never describe a procedure "completely"; you can only describe it in terms of other procedures the reader already understands.*

 ian@northwestern.edu

The imperative model

The functional model is a pretty good model of what we mean by computation. It covers all of math and a lot of programming. But it assumes that computations always read an input, think for a while, write an output, and then stop. The only aspect of a program's behavior the functional model is concerned with is the relation between its input and its output; it considers procedures that generate the same output for the same inputs to be behaviorally equivalent.

On the other hand, the steps of our tip computing procedures involve scribbling on scratch paper. In fact, they basically work by gradually accumulating scribbles until the answer is on the page. And those individual steps of writing, erasing, and re-writing the scratch paper aren't necessarily best thought of as computations in this sense of having specific inputs and outputs. As it happens, computers work internally in much the same way; their fundamental operations involve reading, writing, and erasing locations on what amounts to electronic scratch paper.

But what about this procedure:

1. Draw an X at the top of the page
2. Draw an X under the current X
3. Erase the X above the new X
4. If the new X isn't at the bottom of the page, go to step 2
5. Draw an X on top of the current X
6. Erase the X below the new X
7. If the new X is at the top of the page, go to step 2, otherwise to step 5

This is effectively a simple animation procedure. It makes an X move up and down on the screen, or in this case, the paper. It never produces an output in the sense of the functional model of computation. In fact, it never even stops running. So we have a procedure that:

- Can be followed mechanically
- Does something useful, and
- Does it using exactly the same techniques that "normal" procedures use

Yet it isn't considered a computation under the functional model because it doesn't compute a function *per se*.

Clearly the functional model's narrow limitation of a program's behavior to its output is a significant limitation.[2] Many programs don't return results in the usual sense. The software on your cell phone never stops running, or so one hopes. Similarly, when you use a backup program to make copies of your files, the program's modification of your backup medium isn't

[2] This is somewhat unfair to the functional view. It's easy to make other functional models of computation that can embrace this kind of computation. Our purpose here is not to choose one model over another, but to draw attention to two different ways of thinking about computation.

 ian@northwestern.edu

an incidental side-effect of the program, as it was when computing a tip; it's the whole reason for the program.

Program steps that involve modifying the computer's memory or taking some action in the world are called **commands** or **imperatives**[3]. In practice, the basic instructions followed by your computer are imperatives. Programs are sequences of imperatives that the computer executes in order. As in our tip calculating example, imperatives can be strung together to compute a function. But as with the backup program example, they can also just be strung together to manipulate the computer's memory in a manner we find useful.

This gives us another model of what computation is, which we'll call the **imperative model**: procedures are sequences of commands (imperatives) that manipulate representations. The advantage of this model is that it's less restrictive than the functional model: it covers any kind of manipulation of representations, regardless of whether there are designated inputs and outputs in the functional model's sense. As we shall see later, however, it's often more difficult to think about programs at this level. One reason is it's often more difficult to specify what the program is supposed to do. In the functional model, there's a simple specification of a computational problem – the function to be computed – and a comparatively easy way to decided if the program is correct: just check whether the program computes the correct function. In the less restrictive imperative model, it can be difficult even to specify what a program is supposed to do, much less determine whether it does it.

Now that we have a couple of different ways to think about computation, let's think about what we mean by computers and what makes them unusual as a kind of mechanism.

Interpretation

What we've learned so far is that computation involves the manipulation of representations by following some specified procedure, but that the same computation can be achieved by many different procedures, representations, and mechanisms. As long as the procedures are behaviorally equivalent, they're in some sense interchangeable. The difference between what we've called the functional and imperative models above is largely a matter of what aspects of the procedure's behavior are considered relevant to the task at hand.

What may be less obvious is that what we've called procedures above are themselves representations. They're just text. And as such, they can be manipulated like any other text: copied, modified, erased, etc... What makes them different from other representations is that they can also be followed (executed). The process of reading and following a stored representation of a procedure is called **interpretation**.

---

[3] From the Latin *imperare* "to command".

 ian@northwestern.edu

Programmable devices are simply devices that can store and interpret representations of procedures. A calculator only operates on one kind of representation – numbers. But a programmable calculator provides a way of recording key presses and "playing them back" later, as if they were being pressed by the user. Those key presses form a stored representation of a procedure, and the playback process is a simple form of interpretation.

Programmability is important because it means that a single device can be made to do many different functions, not by physically reconfiguring it, but simply by changing a stored representation inside the device. Computers would be nowhere near the social or economic force they are now if you had to buy separate computers for word processing, email, and editing photos. None of those individual functions would have been worth the cost of an early personal computer. But the fact you can buy one piece of hardware and have it perform a wide range of tasks makes computers economically viable.

Everything is text

Modern digital computers are defined by two special attributes. The first is their programmability. The second is the fact that although they can manipulate many different kinds of representations – pictures, sounds, programs, text – all those different kinds of data representations are ultimately encoded within a single uniform meta-representation.[4] In point of fact, the meta-representation used by computers is perfectly familiar to you. Computers support a basic alphabet of symbols and then string those symbols together to encode information. In other words, computers use text.

The main difference between computer text and our text is that computers use a simpler alphabet with just two symbols, rather than the 26 of our alphabet. Because it has two symbols, it's known as **binary** (after the Latin *bi-* meaning "two"). Over the last 50 years, computer scientists have figured out clever ways of encoding other representations as sequences of binary symbols, including human text, programs, pictures, sound, and more abstract kinds of data such as networks of relationships.

Fortunately, you won't need to learn to think in binary for this course, although you can look at the appendix if you're curious. Most programmers rarely deal with bit-string representations directly, if ever. What matters is that all representations can be encoded in *some* uniform representational medium. The choice of binary text is ultimately a matter of engineering convenience: it's easier to make a reliable electronic memory circuit if it only has to reliably distinguish between two symbols. Early computers experimented with more complicated alphabets, but it turned out to be cheaper and easier to represent complicated alphabets using

[4] To say that something is a meta-thingy is to say that it's a kind of higher-level thingy that can somehow be used to describe, construct, or reason about all the different possible lower-level thingies. So a meta-representation is a representation that can be used to construct or represent other representations.

 ian@northwestern.edu

groups of binary symbols than to build memory circuits that could distinguish many different symbols.

Meta-programming

So modern digital computers are characterized by:

- A unified meta-representation (binary text strings) in which all other representations can be encoded
- A set of built-in commands for manipulating binary text
- The ability to store sequences of commands (i.e. procedures) in memory as binary text
- The ability to interpret (execute) previously stored procedures

The fact that procedures are stored in the same bit-string format as other data has profound consequences; it means that the same commands used to examine and manipulate data can be used to examine and manipulate the procedures themselves. And that means we can write procedures that examine, manipulate, and even create, other procedures. This practice is called meta-programming: the use of software to automate the development of other software.

Automated Testing

Many software development companies use automated systems that run at night when programmers have finished working. They may perform lengthy tasks that programmers wouldn't be willing to wait for during the day or test the program for problems. For example, some game companies check for unintended consequences to changes in their programs by running the latest version of the game at night, putting it through a set of test inputs, and capturing screen images from the play sequence. They then compare those screen captures to the screen captures from the previous day, and email any discrepancies to the programmers. Each morning, the results of a new batch of automated tests are waiting in the developers' email.

The best example of meta-programming is the use of **higher-level programming languages**. As we discussed, computers ultimately describe all data in terms of very simple operations on binary text. Shocking as it may seem, humans are not good at thinking in binary. Moreover the operations the computer understands are so highly simplified that it can easily take hundreds of thousands of machine-language commands to describe a sophisticated program. Writing a program with hundreds of thousands of commands is generally beyond the ability of a human to do reliably; it simply involves keeping track of too many details. On the other hand, keeping track of detail is one of the things computers do very well. So rather than forcing humans to program in machine language, we design programming languages that are easier for people to understand, then write programs, called compilers, to translate them into machine language for us.

There are many other kinds of program-development-programs, some of which you will use in this class. Optimizers are programs that rewrite other programs to eliminate certain simple kinds of inefficiencies, making them run faster. Debuggers are programs that monitor and control the execution of other programs, allowing a programmer to observe a program's

 ian@northwestern.edu

behavior as it runs, and to collect forensic evidence when the program fails. Static checkers scan programs before execution for certain kinds of common errors. Revision control systems keep track of different versions of a program, who made what changes, and when, so that problematic changes can be easily undone. Testing systems run programs against **test suites** – large collections of test inputs – to search for inputs that cause them to misbehave.

The information technology industry wouldn't be possible without meta-programming. Humans are by far the scarcest and most expensive resource in the computer industry. Just as the industrial revolution required the ability to use machines to manufacture other machines, large scale software development requires the ability to automate mundane development tasks, freeing programmers to focus on the tasks that require reasoning and creativity. Consequently, the overwhelming historical trend in software development has been to automate more and more of the development task, even at the expense of writing programs that require more powerful computers.

Simulation

Although all modern computers share the same broad traits, they're not identical. Simple microcontrollers, such as the ones that control small appliances, would likely have commands for adding numbers in binary, but multiplication might have to be programmed in terms of addition. The digital signal processors in cell phones and DVD players, on the other hand, have fairly elaborate command sets for arithmetic because that's what they spend most of their time doing. But even when computers have substantially similar built-in commands, they probably won't encode those commands in binary in exactly the same manner. That's the reason PCs can't run programs written for Macintoshes (at least older Macs), and vice-versa.

However, since all computers store data as bit strings, there's no reason we can't store a procedure written for one computer in the memory of another. Given that, we ought to be able to write a meta-program for one computer that reads and executes programs written for another. The meta-program would make the new computer behaviorally equivalent to the old one: it could interpret/execute the programs of the original computer perfectly, although with a significant speed penalty. Programs that read other programs and do what they say are called **interpreters**; interpreters that are designed to make one computer behave like another are called **simulators** or **emulators**. Simulators are often used to run PC programs on Macintoshes or video games written for old game consoles on newer computers.

Of course, we said that it seemed like we *ought to* be able to write a procedure for one computer to simulate another computer. But could we really do it in practice? There are a few obvious issues, such as that a computer with 1 megabyte of memory probably won't be able to simulate one with 100 gigabytes. But if we assume the simulating machine is given enough memory, can one computer always simulate another? It's always possible that one computer might have some kind of special command that simply cannot be simulated by the commands of another, and therefore that it might be able to compute functions that the other cannot. Is

 ian@northwestern.edu

there a set of commands that are sufficient for simulating any kind of computer no matter what its command set is?

We have the answer, and that answer is "probably."

Universality

The first machines that could simulate other computers were designed by British mathematician Alan Turing, and so are now referred to as Turing machines.[5] Most Turing machines weren't even programmable, but Turing showed that there was a particular kind of Turing machine, which he called the Universal Machine, which could take representations of other Turing machines and simulate them. That meant the Universal Machine was not only programmable, it could compute anything that *any* Turing machine could compute.

The details of Turing machines aren't important to this discussion. What matters is that they can simulate all the computers we're managed to think up, and consequently can compute anything those computers can compute. This means any computer that's able to simulate Turing's Universal Machine is also able to simulate any computer, and therefore able to compute anything we know how to compute. This property of being able to simulate Turing machines, and therefore being able to simulate any known computer, is called **Turing completeness**, Turing equivalence, or just **universality**.

The amazing thing is that nearly all digital computers are Turing complete. In fact, some stunningly simple mechanisms are Turing complete.[6] One of the practical consequences of this is that hardware designers can largely ignore the issue of whether their command sets are universal (they are) and focus on finding command sets that can be made to run very quickly in hardware.[7]

Of course this doesn't preclude the possibility that someone might someday come up with a radically different design for a computer that couldn't be simulated by a Turing machine. But thus far, every computational system that's been devised has either been Turing complete (can simulate and be simulated by any other Turing complete system) or is weaker than a Turing

---

[5] Alan Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, Series 2, 42 (1936), pp 230-265. Note that Turing machines were more a conceptual design than a practical engineering design. Although Turing did work on the design of actual working hardware, his work on Turing machines came before the construction of real computers was practical.

[6] Minsky and Blum proved in the 1960s that a "two-counter machine" was Turing complete. A two-counter machine is a machine whose only data representation is a pair of numbers (non-negative integers, in fact), and whose only commands are to add or subtract 1 from one of the numbers and to check whether one of them was zero.

[7] In fact, current models of the Pentium processor (both Intel and AMD) effectively don't run the Pentium command set but only simulate it. They each have their own internal command set and effectively compile Pentium programs into programs in the internal command set as they run them.

 ian@northwestern.edu

machine (i.e. can't simulate a Turing machine but can still be simulated by a Turing machine). So it is generally assumed that any function that can be computed can be computed by a Turing machine. This is known as **Church's Thesis**, or sometimes as the **Church-Turing Hypothesis**, after 20th Century meta-mathematician Alonzo Church.

The limits of computation

You may have noticed a certain carefulness of wording in our definition of universality: we didn't actually say that universal machines could compute anything, only that they could compute anything that other machines could compute. It turns out that not only are some problems provably uncomputable, but in a sense it's the very power of universal machines that makes those problems uncomputable.

Anyone who's used a computer knows there are times when a program will "hang": you tell it to do something and it just sits there, presumably doing something, but never actually getting back to you with an answer. This generally happens because of some error by the programmer. For example, if we take our tip computing procedure:

1. Start with the last digit
2. Add it to itself
3. If the result is more than ten, then write the 1 above the next digit
4. If there are any digits left
    a. Move to the next digit
    b. Go to step 2, remembering to add in the 1 above it, if you carried a 1 over
5. Otherwise, erase the last digit

And we miscopy it so that it reads (change written in **boldface**):

1. Start with the last digit
2. Add it to itself
3. If the result is more than ten, then write the 1 above the next digit
4. If there are any digits left
    c. Move to the next digit
    d. **Go to step 1**, remembering to add in the 1 above it, if you carried a 1 over
5. Otherwise, erase the last digit

Then a computer following this procedure will run forever because it will keep starting over with the first digit and never getting past step 4. This kind of bug is called an **infinite loop** – the program repeatedly runs the same piece of code without ever **halting**.

It would be extremely useful if your operating system could automatically detect whether a program is hung and inform you immediately when it happened rather than making you wait. This is called the **halting problem**: given a program and a potential input for it, determine whether the program would ever halt (finish) if it were run on that input. The problem is that

 ian@northwestern.edu

some programs just naturally take a long time, so the fact that a program has run for a long time doesn't necessarily mean that it's hung. It's always possible that it's about to finish and give you an answer.[8]

At first, it seems that we ought to be able to use a simulator to solve the halting problem. After all, a simulator tells us the exact output a program generates. That's actually more information than we need: we just need to know whether there is an output, not what its value is. Unfortunately, this doesn't work. The problem is that simulators just blindly follow the instructions of the programs they simulate; if the program it's simulating runs forever, so will the simulator.

Let's think for a moment about what it would mean for the halting problem to be computable. If the halting problem were computable, we could write a meta-procedure – call it "halts?" – that took two inputs, a procedure and an input to run it on, that would infallibly output "yes" or "no" depending on whether the specified procedure would halt if run on the specified input. That turns out to be the difficulty with the halting problem, because it leads to a paradox. For if we could solve the halting problem – if we could write the "halts?" procedure – we could write another procedure, call it "smartass," that looked something like this:

1. Follow the halts? procedure, giving it as inputs this procedure (smartass), and whatever input was given to smartass
2. If halts? says smartass would halt, then run forever
3. If it says smartass would run forever, then halt.

In other words, we could write a procedure that used halts? to ask questions about itself, and then do the opposite of what halts? said it would do. If smartass halts:

- Then halts? will output "yes"
- And so smartass would run forever
- Oops, I guess halts" should really answer "no"
- And smartass will halt after all
- And so halts? will output "yes" after all
- Etc. etc.

This means that the halting problem must be uncomputable because assuming was computable meant assuming we could write halts?, which meant we could write smartass, which lead to a contradiction. When an assumption leads to a contradiction, the assumption must be wrong.

[8] You may have had a similar experience while web surfing. You never know whether a page lookup is taking a long time because the server is down or just because it's overloaded, so you hold little debates with yourself over whether to keep waiting or to stop it and try something else.

 ian@northwestern.edu

This proof, or rather, a much longer and more rigorous version of it, was due to Turing.[9] It's important because it shows that:

- There are problems that are uncomputable by Turing machines, and so by any mechanical system we've ever been able to devise
- Worse yet, they include problems we might actually care about, like checking whether a program is broken
- Ultimately, the existence of uncomputable problems is due to the very power of general-purpose computation and, specifically, to the capacity for meta-computation.

However, the uncomputability of the halting problem is often misunderstood, so let's be very clear on what it does and doesn't entail. It doesn't mean we can never determine if a program halts; in fact, we can solve the halting problem for many of the programs that we care about in real life. What we can't do is write an infallible program for determining whether an arbitrary piece of program text will halt for an arbitrary input text; any program we write for the halting problem must necessarily get it wrong some of the time, even if it generally gets it right in the cases we really care about.

The halting problem has also been used to argue that the brain must not be a computer because people can tell whether programs will halt and computers can't. Regardless of whether or not the brain is best thought of as a computer, people are actually just like computers on the halting problem: we can solve it a lot of the time, but we also get it wrong sometimes.

Imitation, equivalence, and intelligence

There's an old party game called "the imitation game" in which two people, classically a man and a woman, hide in different rooms while the other guests try to guess which is which by submitting written questions and receiving written answers. The idea is to see whether the hiders can fool the rest of the guests. In the terminology we've used above, the people in the rooms are trying to act behaviorally equivalent to one another. Obviously, this doesn't mean that if the impersonators win they've actually changed identities or sexes.[10]

But imagine we play the game not with two people, but with one person and one computer. What if the computer could fool people into thinking it was human? In an important 1950 paper, *Computing Machinery and Intelligence*, Alan Turing (of Turing machine fame) argued that if a computer could fool humans into thinking it was human, then it would have to be considered to be intelligent even though it wasn't actually human. In other words, intelligence

[9] *Ibid*.

[10] That is, sex in the biological sense. One might argue that they've temporarily changed their (socially defined) gender identities.

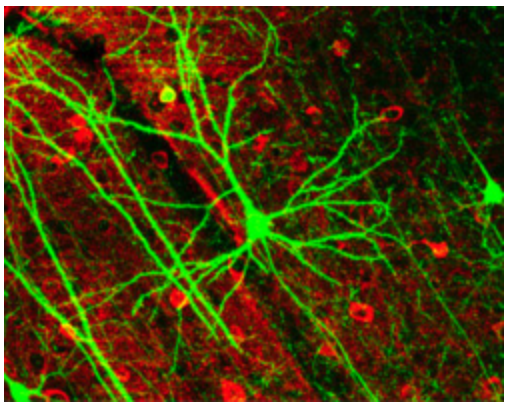
 ian@northwestern.edu

is ultimately a *behavioral* phenomenon.[11] That is, an entity is intelligent because it behaves intelligently, not because it's composed of some special substance such as living tissue or a soul.[12] Moreover, intelligence is a *computational* phenomenon, amenable to computational analysis.

Together with cybernetics, and earlier work on the theory of computation, Turing's paper set the stage for the development of artificial intelligence in the 1950s and 60s. These then provided a scientific model for the study of mental representations and the processes that operate on them, which were then imported into fields such as psychology and linguistics, leading to the development of the field of cognitive science in the 1970s.

Computational neuroscience

We started with the idea of computation being a process that humans perform in their brains, and then looked at how specific instances of that process could be modeled at a coarse level by mechanical processes. That led us eventually to both the limits of mechanical procedures, and the claim that it was nonetheless theoretically possible for a machine to be intelligent. Conversely, it's become common in our culture to talk about the brain as being a kind of computer. At this point, you may object "I don't feel like a computer." So let's talk about what it might mean to say that the brain is a kind of computer. Certainly, the brain can do computation. But that doesn't mean that if you cut open someone's skull you're going to see something that's recognizably like a Mac or a PC. But as we've seen, computation is all about behavioral equivalence. So the real question is could a brain and a PC, in principle, simulate one another? One side is obvious. A person can simulate a PC, by writing down the contents of the computer's memory on (a whole lot of) scratch paper and then manually executing the instructions in its program, one at a time. It would be incredibly boring, but it would be conceptually possible. So the real question is whether a sufficiently big PC could simulate the human brain.



Pyramidal neuron from mouse cerebral cortex (from Wikipedia)

The brain is a network of roughly 100 billion cells called neurons, together with other tissues that help the neurons function. To a first approximation, each neuron has a set of inputs, called dendrites, that receive signals from neighboring neurons and an output, the axon, that

[11] Although widely accepted, the computational view is not universally accepted. See for example, Searle, John. R. (1980) Minds, brains, and programs. *Behavioral and Brain Sciences* 3 (3): 417-457.

[12] Note that this isn't an argument as to whether people have souls or not, simply that it's possible to be intelligent without one. It's not a theological claim.

 ian@northwestern.edu

produces a signal based on the inputs the cell is receiving and the cell's own internal state. Each neuron's axon is connected to dendrites of many other neurons. When one neuron stimulates another neuron, it predictably increases or decreases the rate or likelihood of the second neuron stimulating other neurons. If it's predictable, then it should be possible to write a computer program that predicts and simulates it, and indeed there are many **computational models** of different kinds of neurons. But if each individual neuron can be simulated computationally, then it should be possible in principle to simulate the whole brain by simulating the individual neurons and connecting the simulations together.

While there's a difference between "should be possible" and "is actually possible", this is the general idea behind the claim that we can understand the brain computationally. Assuming it's right, it has some important consequences. For one thing, if brains can be simulated by computers, then computers could be programmed to solve any problem brains can solve, simply by simulating the brain. But since we know that computers *can't* solve certain problems (the uncomputable problems), that must mean that brains can't solve the uncomputable problems either. In other words, there are fundamental limits to the possibility of human knowledge.

Although we're a long way from being able to fully simulate a brain, computational neuroscience, in which scientists try to understand neural systems as computational processes, is an important and growing area of biological research. By modeling neural systems as computational systems, we can better understand their function. And in some experimental treatments, such as cochlear implants, we can actually replace damaged components with computing systems that are, so much as possible, behaviorally equivalent.

Who am I?

Simulation also raises questions about personal identity. There's a long tradition in Western culture of identifying the self with one's thoughts. But if our brains, and thus our thoughts, can be simulated, to what extent does that mean we ourselves can be simulated? Is the particular substance of our construction – biological tissue – somehow crucial to our identity, or could we, in principle, map out all the neurons, synapses, and connection strengths in our brains, and make simulations of ourselves? There are people who argue we could live forever by "downloading" ourselves into silicon.

We're a long way from being able to simulate a full brain, much less map a living one. For the foreseeable future these issues will be confined to science fiction and philosophy papers.

But in many ways the question of whether we will ever actually be able to download ourselves into silicon is less important than how ideas of computing and simulation are already changing society's attitudes about self and identity. As artificial agents become more life-like, will we start to view them as real people? Will we someday have a Mario Liberation Front arguing for legal rights for video game characters?

 ian@northwestern.edu

Computational-X (for all X)

The general approach of understanding the behavior of systems by constructing computer simulations of them isn't limited to biology. Any system whose behavior is predictable can be simulated, or at least approximated, in software. In computational science, theories about the natural world can be tested more precisely by embodying them in computational simulations and comparing their behavior to the observed behavior in the physical world. For example, in **computational chemistry**, the behavior of atoms can be simulated under conditions that could not be adequately controlled in the laboratory. In **computational biology**, simulation is used to understand the folding of proteins inside of cells to better understand the relationship between their amino acid sequences and their functional structure. These can then be used in applications such as **computational drug design** to try to custom design molecules with desired pharmacological behavior. In engineering, computational simulation allows prototype designs to be tested in cases where the construction of physical prototypes would be prohibitively expensive or dangerous. In **financial engineering**, computer models of the behavior of markets are used to design financial instruments that give investors better control over their risk levels.

Computational everything

*Quantum physics has two unusual features: first, physical properties are **quantized**, that is, limited to a set of discrete values as they are in computers, rather than a true continuum, and, second, that physical systems behave differently depending on where and how they're **observed**, therefore understanding the behavior of the universe requires explicitly reasoning about what information is available about it at what times.*

*The interpretation of quantum physics has a long and controversial history. However, one minority interpretation, **pancomputationalism**, argues the universe is best thought of as a computer itself. But then what would this even mean? Notice that the definition we've been using of computation - that it involves the manipulation of representations - doesn't work anymore. If the universe is a computer, what is it representing? Itself? Your consciousness? If computation without representation is possible, then we have to ask all over again the question of what computation is.*

Okay, so what *is* computation?

The point of the foregoing was not to argue that the universe is a computer or that we should all download ourselves into silicon, but rather that computation is an idea in flux. Our culture is in the process of renegotiating what it thinks the concepts computation and computer really mean. Computation lies on a conceptual fault line where small changes can have major consequences to how we view the world. That makes it a very interesting field to follow right now.

It's an interesting exercise to type "define computation" into Google and look at the definitions you get. They include definitions like "determining something by mathematical or logical methods", which doesn't apply well to the idea of DNA being a computer program, or "finding a solution to a problem from given inputs by means of an algorithm", which is essentially the

 ian@northwestern.edu

functional model that, as we saw, doesn't apply well to things like computer games, or for that matter brains (although it may apply well to individual parts of brains). Another possible answer is that computation is "information processing", although that then begs the question of what we mean by information.

A pragmatic definition would be that computation is what (modern, digital) computers do, so a system is "computational" if ideas from computers are useful for understanding it. This is probably the closest definition to how real people use the term in practice. As we've seen, there are a lot of different kinds of systems that are computational under this view.

But that isn't a very satisfying definition. Another view would be that computation is wrapped up with the study of behavioral equivalence. Under this view, computation is the process of producing some desired behavior without prejudice as to whether it is implemented through silicon, neurons, or clockwork. The key theoretical issues in computation are then:

- How do we specify the desired behavior?
  The functional model gives us a conceptually simple approach, but it isn't easily applicable to all problems
- How do we analyze natural systems – weather systems, stock markets, brains – using the conceptual tools of computation?
- What kinds of behavior are possible to produce mechanically?
- What is the cost of producing a given kind of behavior? Are there some kinds that are prohibitively expensive? (Answer: you bet!)
- How do we piece together existing capabilities (behaviors) to produce new kinds of behavior?

In practice, most computational systems that we engineer will be built using general-purpose, programmable computers. For these systems, we also have an additional set of concerns:

- How can we automate as much of the programming task as possible?
- What are common building-blocks of programs that we can reuse without having to rebuild them each time?
- How do we design computer hardware to run as fast as possible?
- How do we design computer languages to be flexible and easy to understand?

Finally, there are a number of psychological and social issues we need to think about:

- What kinds of computational systems are easy for people to use?
- How can we use human psychology to design systems that people find rewarding? For example, what kinds of games or other interactive entertainment systems are naturally engaging?
- How can we design large systems so that they're easier for humans to understand?
- How can a group of people divide up a programming task in such a way that they can work together effectively?

 ian@northwestern.edu

- What are the risks of information technology? Do we really want to make airplanes that are controlled by computers, so that if program crashes, the plane does too? Do databases and surveillance technology give governments and corporations too much power?
- How can we use information technology to improve people's lives?
    - Can we use automation to help the elderly to live independently, or disabled people to move about their environments better?
    - How would inner city schools be different if we had one laptop per child?
    - Can we build automated tutoring systems to allow everyone to have one-on-one help when they need it?
    - What sorts of automation would be useful and sustainable for people in developing countries?

We won't have time to go into all of these questions in class. But these are active research topics in computer science.

Computation is a broad, rich field. It has had deep influences on our lives and culture. If your PC, cell phone, and web access suddenly disappeared, you would probably have to radically reconfigure your life, even though these have only been widely available for the last 15 years. It's unlikely that the changes to society are going to stop any time soon. By learning about computation, you can be in a position to help make sure those changes are for the better.

Appendix: binary representation of data

Although computers essentially store their information as text, they use a much simpler alphabet than human languages. Fortunately, the choice of alphabet really doesn't matter. The Japanese hiragana character set, for example, has a separate character for each possible syllable in the language. The Roman alphabet, however, uses letters to represent individual sounds within a syllable. Fortunately, we can perfectly well write the Japanese text:

"すき です よ" (English: "I like it")

in the Roman alphabet by representing each hiragana character with a set of Roman characters, in this case, the Roman characters that would be pronounced the same as the hiragana character:

す	き	で	す	よ
su	ki	de	su	yo

To form the Romanization:

suki desu yo

In this case, we happened to encode the hiragana character set in the Roman character set in such a way that the English pronunciation of the Romanization would be a good approximation

 ian@northwestern.edu

of the correct Japanese pronunciation. But we don't strictly speaking have to do that. We could just call the first 26 hiragana characters "aa", "ab", "ac", etc., and then call the next 26 "ba", "bb", "bc", etc. Or we could number then:

す	き	で	す	よ
01	02	03	04	05

To form the encoding:

0102030405

Since the alphabet doesn't really matter, electrical engineers are free to design computer hardware using whatever alphabet is convenient for electronic circuits, then encode whatever alphabet we really want to use as short strings in the computer's native alphabet. That alphabet turns out to have only two symbols.[13] This two-letter alphabet is called **binary** (from Latin *bi-*, "two"). When we say that information is encoded in binary (or "digitally"), we just mean that it's represented as a series of symbols in the binary alphabet.

You may well have been told that computers store everything as numbers, but that isn't strictly true; binary is ultimately just text with a two letter alphabet. However, if we choose to interpret those letters as digits of a number, then we can represent numbers in base two. Base two is like base ten, except we've only got two digits – 0 and 1 – and so we have to carry when we go past 1 rather than when we go past 9. Using base 2, we can write any side number as a series of binary symbols or **bits** (short for "**bi**nary digi**t**"):

- 0 is "0000"
- 1 is "0001"
- 2 is "0010"
- 3 is "0011"
- 4 is "0100"
- 5 is "0101"
- 6 is "0110"
- 7 is "0111"
- 8 is "1000"

[13] For those who are electrically inclined, here's the general intuition behind this. Ultimately, information in a computer is represented as voltages or currents on a wire. Many computers, for example, use voltages between 0 and 5 volts. However, it's easy for external influences (electrical interference, power supply variation) to make small changes to the voltage on a wire. If we try to represent 26 different symbols using different voltages between 0 and 5V, then the difference between the voltages for different symbols is around 0.2V, which makes it easy for external influences to change the wire from one symbol to another. If we only have two valid voltages – 0V and 5V – then even large amounts of interference won't change the voltage enough to confuse the circuitry. In biology, the brain uses this same tactic when transmitting information over large distances. Rather than using a continuously varying voltage, neurons use only two voltages and change how often the voltage is "high".

 ian@northwestern.edu

- etc.

Of course, we can also represent English text by using groups of binary symbols to refer to letters. The most common representation for text at the moment[14] uses groups of 8 bits, playfully called **bytes**, for each letter:

- A is “01000001”
- B is “01000010”
- C is “01000011”
- D is “01000100”
- etc.

The word “BAD” would then be represented by grouping the bit-string representations of the individual letters into one large bit-string:

B	A	D
01000010	01000001	01000100

However, the old 8-bit code can only represent the Roman alphabet, the Arabic numerals, and a few extra symbols. Recently, the computing industry has been moving to a 16-bit code, called Unicode, which can represent not only the Roman alphabet, but also the character sets of all major languages of the world.[15] For example, the Unicode encoding for “すき です よ” would be:

す	き	で	す	よ
0011000001011001	0011000001001101	0011000001100111	0011000001011001	0011000010001000

Digital media

Hopefully by this point we’ve convinced you that you can represent any text in binary, no matter what the actual alphabet of the text is, and that we can represent numbers as text, and therefore in binary. What about images? Images can be represented as series of pixels, each of which has a specific brightness, and recording the brightnesses of each pixel as a number.[16] Since each number can be represented in binary text, the whole thing can be represented in binary text. And we’ve already seen that procedures can be represented as text, since all the

[14] This is the American Standard Code for Information Interchange or “ASCII” code, the more modern version of which is the International Standards Organization’s standard number 646 or “ISO” code.

[15] A complete listing of all the scripts represented in Unicode is available on the web. Check out the pages on Cuneiform and Runic.

[16] This is technically only a way of approximating an image, since we think of images as being continuous and as potentially having finer details than can be captured by the pixels. However, we can approximate the image to any desired level of accuracy by using more pixels. Similarly, other continuous quantities, such as the brightness of an individual pixel, can only be approximated, but can be approximated to an arbitrary accuracy by using more bits just as numbers can be written more or less exactly in decimal notation by using more digits after the decimal point (more “places of accuracy”).

 ian@northwestern.edu

procedures we've discussed so far have been presented to you as English text. Although computers use a more precise and efficient encoding for their procedures than English, the fundamental idea is the same. Other kinds of representations can be encoded in bit-strings using various clever techniques that computer scientists have developed over the last 50+ years.

 ian@northwestern.edu