Category Overview

11/21/07

Computation

A new way of science

Less than a generation ago, mathematicians and technologists were the only serious users of the word algorithm. Today, algorithms are part of everyday conversation in many fields including engineering, science, commerce, literature, art, and music. They are even the subject of a political jibe: "What did Bill Clinton play on his sax? Al Gore rhythms."

Algorithms not only describe and control what computers do, they have become a way of thinking. The algorithmic interpretation sees worldly phenomena as manifestations of information and seeks algorithmic solutions for problems. But there are limitations. Despite their awesome power, computing machines cannot do many things that humans can do such as recognize faces, label images, or create new interpretations. Those who have devoted many years trying to make computers behave like brains have succeeded mainly in cultivating brains that think like computers!

What's behind the transformation of computation from the work of esoteric electronics to the thought of everyday persons?

From Artifact to Natural Process

This transformation has actually been taking place incrementally for seventy years.

Computer science was born in 1936, at the dawn of the electronic digital computer, when Alan Turing wrote about the capabilities of computing machines. The first digital computers came to life in laboratories in the early 1940s and became commercial realities in 1950. In those days, computation was touted as a tool for science, engineering, and business. The principal early applications of computing were as equation solvers, code breakers, data analyzers, and database managers. By the 1970s computing also encompassed data communications and coordination.

By the middle 1980s the view of computation had moved beyond tools. Nobel Laureate Physicist Ken Wilson, honored for his

computational discoveries of magnetic properties of materials, was one of the first to claim computational science as a new field. In this field, computation is a new method of science, taking its place alongside the traditions of theory and experiment. There were numerous supporters in other fields besides Physics. Biologists used computational methods to piece together the genome database and to understand the transcription of DNA. NASA scientists used computational methods to design the heat shield materials for the Jupiter probe. Pixar created all-new computational animation methods for motion pictures. Economists developed simulations of national and world economies that became the bases for setting economic policies.

Within fifteen years after Wilson advocated computational science, other scientists were making bolder statements. Nobel Laureate David Baltimore claimed that DNA transcription is inherently computational: nature placed an information process at the foundation of life. Physicists now see all particles and their interactions as manifestations of information-carrying quantum wave functions. They have contributed new notions of quantum computing, quantum cryptography, and quantum algorithms. Economists see economic systems as large complex information systems; computational models are information replicas of the real thing. To these people, computation is not simply a tool, it is a principle at the core of their fields.

This is a remarkable shift. In the 1940s, computation was what computers did. In the 2000s, computation is something nature does; computers are tools to study computations.

Computer science has oft been faulted for the word "science" in its title. Although computer science met most criteria for being science, the critics said it failed because it did not study natural objects. With the discovery of natural information processes, all that has changed. Computer science studies natural as well as artificial information processes.

The lesson for us is that computation is the principle and the computer is the tool. In the Great Principles project we have had to rewrite many of our basic definitions to reflect this reality.

Representations, The Heart of Computing

The historical record clearly shows computer science maturing from a tools field to a widely-used natural science. What makes computing so broadly attractive?

Perhaps it's because the stuff that computers act on is broadly useful. What do computers act on?

The simplest computers contain a program and a memory to hold data. The program controls the actions of the computer on the data. So the first-cut answer is: computers act on data.

What are data? In computers data are arrangements of bits (0 or 1). The bit is actually a representation of a state distinguishing two alternatives. The same bit is represented in different ways in different media -- for example, magnetic patterns on hard disks, pits and peaks on CDs, circulating currents in RAMs. The computer has mechanisms that can write bits into the various media and read them back later. The bit became popular in the 1940s because engineers found they could make logic circuits very reliable if they only had to distinguish two states rather than three or more. They could represent larger states with bit strings.

This is not exactly how a user thinks of data. To a user the data stand for (hold information about) objects or effects in the world. For example, the samples of a sound wave stand for the actual sound; the numbers in bank records stand for account balances; the numbers output by an autopilot stand for signals that drive servo-motors.

You may now be thinking, "Wait a minute. Data are an arrangement of bits. My brain supplies the meaning when perceiving the arrangement. The bits themselves have no meaning. How then can bits hold information?" This is exactly right. Information is actually a brain response to the stimulation of a pattern of bits. Holding information is a metaphor. Fortunately, this metaphor does not render meanings of representations purely subjective: we live in communities wherein we all assign the same meanings to the same patterns.

Therefore we can refine our answer: computers act on representations. The computer's output representation can be converted to action by a device that translates the representation into a physical effect, such as a servo-motor in the airplane.

The data are not the only representations in a computer. The program that controls the computer is also a representation. It represents an algorithm, a set of mechanical steps that transform input data to

output data. Therefore, we further refine our answer: a computer is a device that transforms data representations under the control of a procedural representation.

We conclude that there are only two essentials of computation:

- A series of representations; and
- A set of rules for transforming each representation to the next in the series.

The computer is not essential! The computer is one of many possible media in which computations can happen. Computing is pervasive because representations are pervasive.

Sometimes Analog

Not all computers work by following the steps of a discrete algorithm. Analog computers demonstrate. In the 1920s and 1930s Vannevar Bush at MIT developed the differential analyzer, a machine that represented differential equations as operators connected by gears, shafts, and integrating wheels. It could solve a large number of physical problems such as ballistic trajectories of artillery shells for the military.

In the 1940s and 1950s, electrical engineers developed electrical simulations for physical systems described by differential equations. The components of the circuit represented operators and coefficients in the differential equation; the currents flowing in the circuit represented the solution of the differential equation. Programming such a machine consisted of writing down the differential equation and wiring up a circuit to simulate it. The output was the recorded waveform of the current flowing in the circuit.

Some analog computers work by setting up electric fields in a conducting medium with boundary conditions simulating conditions in the world. The electric field represents the solution of a differential equation with the same boundary conditions. The output is a recording of the electric field in the medium.

Some neural network computers work by passing currents through a feedback network of operators that combine internal currents. These networks fall into one or more stable states, which represent the learned responses of the network.

The common feature of these analog computations is that they represent a physical process as a set of equations and then use an electrical or mechanical medium to find a current or a motion that solves the equations. In this case, the computation state is a

continuous function. The rules for transforming state at one point of time and space to state an infinitesimal time or space later are given by the differential equation. When this is taken into account, the two essentials of computation are:

- A series of discrete representations or a continuous-time function representation.
- A set of rules for transforming a representation into the next one in sequence or at the next infinitesimal increment of time or space.

Sometimes Nonlinear

What should we make of DNA as a natural representation of the basic information to generate the cells of an organism?

Our common lore about DNA is that it is a double-helix strand of proteins, each component of which is one of two possible amino-acid base-pairs. DNA sequencing reveals the series of base-pairs strung together in DNA. All 3.2 billion base-pairs of the human genome were finally sequenced in 2005.

Our lore also says that the RNA of a cell reads the DNA by finding and following subsequences, producing new amino acids as instructed by the DNA. Many popular assumptions are linked to this, for example, the idea that particular subsequences indicate a disposition to a disease.

There is a problem with this lore. The real DNA is folded up into a tightly wound cluster. The folded shape is unique and is a minimum energy configuration of the DNA molecule. An unfolded DNA strand will re-fold into this unique configuration when the forces that keep it unfolded are released. The actual function of DNA is intimately linked with its folded form. The RNA that reads the DNA contains receptors that match certain proteins. An RNA molecule can link to a pair of proteins that are physically close on the folded DNA, but are separated by a large distance on the unfolded DNA strand. That RNA cannot function on an unfolded DNA. It works only when the DNA is folded.

DNA-RNA illustrate two basic principles of representation: (1) information is embodied in the physical configuration of a carrier, and (2) the interpretation of information is assigned by the observer. Because the RNA cannot "see" certain important protein-pairs in unfolded DNA, some information present in the folded form is absent in the unfolded form. The 3D shape of the carrier is important to the information that can be perceived. This consideration is not present when we think of purely linear representations such as bit sequences.

These aspects of representations highlight the problem in the lore about DNA: the entire system depends not only on the 3D shape of the DNA but on the RNA receptors. Hardly anything is known about either. We have a long way to go before we gain an understanding of DNA transcription and how we might manipulate or control it.

Representations of the Infinite

The simplest representations stand for a single entity or a finite set of entities. For example, a single number can be represented as a finite binary string, and a set of experimental data by a series of binary strings separated by markers. An algorithm can also be represented by a finite binary string.

Things get really interesting when we want to find finite representations for infinite sets.[1] An example is the grammar of a language. A grammar is a finite description of a potentially infinite set of sentences (strings) in a language. A grammar can help us analyze a given string to see if it is part of the language. A grammar can also be used to generate the strings of the language in some systematic order (such as all strings of length 1, then of length 2, and so on). The important point is that the grammar is finite, the language infinite. A syntax description of the Java language represents the set of all allowable Java programs.

The algorithm is another example of a finite description of the infinite.[2] In this case, the entities represented are the computations that the algorithm can generate. This is what makes programming so difficult. The program's designer needs to be able to show that every computation in the infinite set meets the input-output specifications of the algorithm.

Compression

Compression means to find a shorter representation that holds the same information as the original.

There are many representations for any given entity. For example, by adding leading zeros, we could represent a number in an infinite number of ways: 11, 011, 0011, 00011, 000011, etc., all represent

---

[1] An amusing paradox illustrates the dangers of assuming that infinite sets can be consistently represented finitely. Let S denote the infinite sum 1+2+4+8+16+... . Then S = 1 + 2*(1+2+4+8+16+...) = 1 + 2*S. This implies S= -1, a contradiction.

[2] Finite descriptions of algorithms generate their own paradoxes; for example, if we assume we can enumerate all algorithms, we can show there is always an algorithm that is not enumerated.

the number "3". Given a representation, can we find a shorter one that means the same? Is there a compression threshold below which we lose information?

These questions are of immense practical value because long files require more storage and take longer to process or transmit than short ones.

Around 1840, Samuel Morse created the Morse code for the telegraph. He assigned short codes to the most frequent letters and longer codes to the least frequent (thus "E" got the code "dot" and "J" the code "dot-dash-dash-dash". Morse found that this code would allow for faster transmissions than if all letters had the same number of dots and dashes (5-symbol code words would be needed to capture all 26 letters). Thus the Morse code compressed standard English. It was a lossless compression because telegraph operators could reproduce the English letters by listening to the dots and dashes.

A century later, Claude Shannon asked what would be the shortest possible code for lossless transmission in a radio channel. He concluded that it would be a binary code whose average number of bits equals a number called entropy. Entropy measures the average number of bits of uncertainty resolved when the receiver decodes a code word. In 1952, David Huffman showed how to construct a code that would be at most one bit away from Shannon's optimum. Shannon showed that if the code were any shorter than this, some information would be lost and the receiver would not be able to accurate reconstruct the original message.

In the computer era, the Huffman code was adapted into the popular "zip" compression algorithm, which cuts file sizes by as much as a half.

Can greater compression be achieved by tolerating some loss? For example, in a digital image that was sampled with 16 bits per pixel, we might save only 2 bits per pixel and compress the file to 1/8 its original size. This would be worthwhile if the eye could not detect the loss of gray scale in the image. A large variety of image and video compression schemes grew up from this observation.

Sound researchers discovered that the ear cannot hear notes that are close in frequency to louder notes. The implication is profound: there are notes in Mozart symphonies that no human, not even Mozart, has ever heard. This fact was exploited in the MP3 sound compression technology, that achieves a reduction to 1/10 the original sound file at no perceivable loss of fidelity. MP3 and its variants gave birth to a whole new music distribution industry.

These video and audio compression schemes are lossy but value preserving. They discard bits that are of no value to the receiver.

The compression schemes discussed above all work on representations of finite things. What can we say if we try to compress a representation of an infinite set? For example, can we find a shorter algorithm with exactly the same input-output behavior as a given algorithm? Now it gets weird. There is no effective way to tell if it any algorithm -- shorter or longer -- implements exactly the same input-output transformations as the original. Not only that, we can't even tell if a shorter algorithm exists at all! The formal name for this is Kolmogorov complexity, defined as the length of the shortest representation equivalent to a given one. Any attempt to compute this number runs headlong into a paradox like this 14-word beauty: "Define n as the smallest integer that needs at least 15 words to define."

Turing's Computability

In 1936, Alan Turing published a famous paper, "On the computable numbers, with an application to the Entscheidungsproblem." Turing defined computation, computing machines, and universal machines, and he nonchalantly showed that the halting problem for machines was not computable. From that conclusion he demonstrated that the century-old "Entscheidungsproblem" (German for decision problem) had no solution. That problem posited a complete and consistent universal logic system that would be able to tell if a proposition from any other logic system is true. It was on the mathematician David Hilbert's 1928 list of challenge problems in mathematics. It dreamt of a method to tell "by inspection" whether a computation halted. Turing showed that if such a logic system existed, it would be able to answer the halting question. He concluded that the dream of a "by inspection" method to answer halting questions was impossible. He showed that the very steps a mathematician might use to apply a "by inspection" method were fundamentally computational. Therefore the only general method of approaching the halting question is to run the computations and see what happens. Turing thus showed that computation is unavoidable. This truly was the birth of computer science.

Turing defined a simple machine model consisting of a finite control unit, an infinite tape, and a read-write head. The control unit reads a current symbol from the tape and enters a next state while writing a new symbol and moving one symbol left or right. This model threw out all considerations of processor speed or memory size. The only

thing it kept is the logical steps needed to transform an input representation into an output representation.

Turing thought it especially important that there is a "universal machine" that can simulate any other machine. This is achieved by encoding the program and data of the simulated machine in a standard notation. The universal machine repeatedly looks up the logic rule for the next state transition and applies it to the encoded data. In spite of its utter simplicity, Turing's machine model can simulate any other computer. Aside from differences of speed and memory usage, all computers are equivalent in that any input-output function computable on one can be computed on any other.

Eventually people working with various models of computation all discovered they could simulate each other's model. These included Church's Lambda Calculus, Post's Production Systems, and Kleene's Recursive Functions. This led to the Church-Turing thesis, which says that anything that can be effectively computed can be represented as a Turing machine. The CT thesis can never be proved, but the evidence is so strong that no one doubts it.

Resolution Times

Turing's machine model gave a clean way to approach one of computing's most difficult questions: How long does it take for a machine to compute an output for a given input? Turing machines have no complications like instruction pipelining, multiple CPU cores, clock speed, or memory hierarchy to complicate the answer. All that matters in is how many steps the machine takes.

Turing answered one of these resolution questions in his 1936 paper. He examined whether it is possible to design a machine that could tell whether any other machine would halt for a given input. Halting means that a machine stops in a state signifying "all done, output ready." He showed that if such a machine existed, we could ask it whether it stops when its input is a copy of its own program. This creates a monstrous paradox. Therefore, no such machine can exist. In effect Turing said that there is no hope to determine "by inspection" if a machine halts. We can only simulate it and see what happens. Direct simulation is inclusive: if the simulation is still running by our deadline, we can conclude nothing about whether it will ultimately halt.

Very quickly others identified a whole slew of interesting questions that cannot be resolved by any algorithm -- for example

- Are two programs equivalent? (Same output for every input.)

- Does a program meet its input-output specification?
- What is the maximum time a program can possibly take and still halt?
- Do two different coding schemes give the same representation of the same message?
- What is the length of the shortest representation equivalent to a given representation?

Even more important than saying what cannot be computed, Turing's model offered algorithms analysts a powerful, orderly way to approach the resolution times of real computations. Can we classify algorithms by their resolution times? Can we extend the classification to problems, where resolution time refers the time of the fastest algorithm solving the problem? Can this help us decide how much processor and memory resource we need for practical problems of science, engineering, and commerce?

As they searched for answers, algorithms analysts discovered a very weird answer. They found a large group of over 3000+ common science, engineering and commerce problems, which we call the "3000 Club", that were all related in a strange and beautiful way. For every one of them, the best known algorithm is excruciatingly slow, taking years or centuries to solve problems of moderate size. And, in the unlikely event someone in any field finds a fast way to solve any of them, that solution procedure could quickly be translated into a fast solution procedure for all the others. In other words, they are all hard or they are all easy. No one knows. This question has been listed as one of the most baffling problems of mathematics, just as the Eintsheidungsproblem baffled mathematicians in the decades before Turing.

The "3000 Club" of problems includes

- Finding the capacities of Internet links to avoid congestion
- Finding shortest shipping routes in a transportation system
- Packing transistors on to a silicon wafer to minimize waste
- Scheduling assembly line jobs for minimum completion time
- Configuring a molecule from Schroedinger's Equation, and
- Rendering a scene in a movie.

Finding good algorithms for these problems is worth billions of dollars to the economy in resources not wasted by suboptimal solutions.

The first step in the search that led to the 3000 Club problems was to characterize the resolution times of common, well-known algorithms. Following the Turing machine's bare-bones simplicity, analysts created the O-notation ("order of") to bound the worst case time of an algorithm as a function of input size. The only thing that matters in a Turing machine is the trend line of how the resolution time grows with the size of the input. For example:

(1) A linear search algorithm takes time O(n) to find a particular item in a list of n items.

(2) A bubble sort algorithm takes time $O(n^2)$ to rearrange n items into ascending order.

(3) A matrix multiply algorithm takes time $O(n^3)$ to compute the product of two nxn matrices.

(4) An enumeration algorithm takes time $O(2^n)$ to list all the possible orderings of n items.

Exponentials grow so large so fast that even for small inputs the algorithm can take many centuries or may exceed the lifetime of the universe. Any algorithm exponential or worse is for all practical purposes beyond our reach even if it is technically computable. For this reason all algorithms of exponential time ($O(2^n)$) or worse came to be called "intractable". In contrast algorithms whose computation time was polynomial ($O(n^k)$) or better were called "tractable". The first three entries in the list above are tractable, the last intractable.

This idea was extended to problems by defining the inherent complexity of a problem as the O-function of its best algorithm. For example, the "sorting problem" is to arrange a list of n items in ascending order. There are numerous different sorting algorithms. Some of them take $O(n^2)$ and the best take O(n log n). We also have a proof that it is impossible to sort in time less than O(n log n). Therefore the difficulty of sorting is O(n log n). Knowing inherent problem difficulty is of enormous help in designing algorithms -- we would not, for example, settle for an exponential algorithm if we knew the problem was polynomial.

By 1970 algorithms researchers had compiled some of the catalog of the 3000 Club. These algorithms share the common feature that they rely on enumerations of combinations of n variables. The enumeration is the source of the $O(2^n)$ behavior. Was there any way to overcome all this work?

One attractive simplification is to convert optimization problems to decision problems. An optimization problem seeks to find a combination of variables that maximizes a value function. A decision

problem seeks to find any combination of variables whose value exceeds a given threshold. Decision problems model a less greedy approach: instead of going for the global optimum, they accept anything that attains a good-enough level.

Unfortunately, the decision-problem versions of the 3000 Club problems still had exponential time. Is there a way to completely sidestep the enumeration behavior?

In 1971, Steve Cook proposed such a way. He proposed to consider nondeterministic algorithms for solving decision problems. Nondeterministic means that the algorithm makes a random guess among alternatives when it encounters a choice point at which it cannot know which alternative leads to the desired outcome. With the right set of guesses, the algorithm would avoid enumerating anything and would find the desired outcome rapidly. The resolution time of the nondeterministic decision problem is the time taken when all the guesses are right. Another way to put this is: a proposed solution can be verified in polynomial time. The verification applies the value function without enumerating anything.

The knapsack problem is a nice example of Cook's insight. We are given a set of n items of various sizes and values. The objective is to find the highest-value subset to pack in the knapsack. The decision problem form says that we want any subset that makes the value of packed items at least V. The only known algorithms for doing this enumerate subsets, checking each one for its value, and stopping when a subset has value at least V. This still takes exponential time because in the worst case the algorithm would have to enumerate all the subsets before finding one with value at least V. On the other hand, if someone proposes a subset, it is really easy -- time O(n) in fact -- to compute its value and tell if that is at least V.

Now it seems like we are getting somewhere. We have two classes of problems:

- P: those solvable in polynomial time by a deterministic algorithm; and
- NP: those solvable in polynomial time by a nondeterministic algorithm.

The 3000 Club is part of the class NP. Nondeterminism definitely gets around the exponential problem. Unfortunately, it's not practical because we can't build a nondeterministic machine. Therefore, knowing a problem is NP doesn't tell us whether or not we can find a deterministic polynomial algorithm for the problem.

Cook also noticed another thing: the hardest problems in NP can be grouped into the subset NPC, meaning NP-complete:

- Every problem in NP can be reduced to one in NPC.
- Every problem in NPC can be reduced to every other problem in NPC.

Reduction means that one problem can be encoded as an instance of the other and solved by proxy. In 1983, Mike Garey and David Johnson published a book inventorying the 3000 Club NPC problems. If there is a fast (deterministic polynomial) algorithm for any member of the Club, there is a fast (deterministic polynomial) algorithm for every member of the Club. In fact, a fast algorithm for any NPC problem can be transformed to a fast algorithm for any NP problem.

The question whether there is a fast algorithm for any NPC problem has been abbreviated: P = NP ? If there is a fast deterministic algorithm for any member of NPC, it becomes a fast deterministic algorithm for all of NP, thus converting all of NP to P. The class NP stands or falls together. Either they are all hard or they are all easy.

The question of whether P = NP become the biggest open question in computing and mathematics. No one knows. That no one in any field of science, engineering, or commerce has ever found a fast algorithm for any of these 3000 problems suggests empirically that P ≠ NP. In our current state of knowledge all those hard problems are hard and there is little hope anyone will find a way to solve them quickly.

The Birth of Computer Science

In the introduction, we outlined a consensus in the science, engineering, and business arenas that information processes and computation are fundamental and unavoidable in their fields.

This is an amazing consensus. Barely a generation earlier, there was no such consensus. Computations were seen as manmade products of manmade computers. They thus did not earn the same stature as the natural. Many skeptics questioned whether there could even be a legitimate science of the artificial. All that has changed. The new consensus gives computation a place among natural processes.

It is even more amazing that Alan Turing told us this much back in 1936 when he discussed what is meant by computation in his famous paper on the computable numbers. As a consequence of his definitions of computability, there is no algorithm to decide if a computer halts, and there is no answer to the famous decision problem in logic. Any "by inspection" method to tell if a theorem is

provable in a logic system could be used to answer the halting question for a computer. The mathematician David Hilbert, who was inclined to think that such a system of logic existed, must have been baffled by Turing's answer.

Turing showed that the steps of a "by inspection" method are fundamentally computational. When the steps of the method are provided as an input to the method, a monstrous contradiction results.

Turing's discovery implies that there can be no "inspectional" method to tell if a computation terminates. The only choice is to plunge in, run the computation, and see what happens. Computation, thus said Turing, is unavoidable.

This truly was the birth of computer science.