Chapter 69.2 Computer Science
In Vol IX, La Grande Scienza, Storia della scienza

Computer science is unusual among the exact sciences and engineering disciplines. It is a young field, academically. The first academic departments of computer science were formed in the 1960's. Most major universities had established programs by the early 1970's. The founding faculty came from math, physics and electrical engineering. Although called a science rather than engineering because of its core of applied mathematics, it has been strongly influenced by computing practice and the rapid growth of jobs in computing. The practice of computing has changed dramatically over the roughly 60 years that have elapsed from the first computing machines of World War II to the present. In this article, I will summarize the evolution and some of the major achievements of computer science during its brief but eventful history. The nature of the computing machinery has played an important role in the selection and definition problems solved by computing, especially in its early years, before 1960. Computer science today stretches in many directions, from improving the methodologies within which commercial software is developed to making contributions to mathematics by extending our understanding of the nature of knowledge and proof. Space does not permit treating all of this, so I will focus on the things that one can compute and the contexts of use that have stimulated advances in computer science. Our understanding of the nature of computing has evolved in these 60 years. At the end of the article I include a brief section speculating on directions that might influence how computing will be done at the end of the 21st century.

Computation before computer science (pre 1960): visions and visionaries

Ancient history contains examples of fixed-function analog computing devices, usually for navigation. One example is the Greek orrery discovered in a shipwreck near Antikthera, from about 80 BCE. The device, when reconstructed, proved to predict motions of the stars and planets. Devices more directly connected to computation, such as the abacus (from about 3000 BCE) and the slide rule (the first, Napier's "bones" ca. 1610) were different in that they could be used to solve multiple problems, given different inputs. Herman Hollerith's tabulation machines, first created in 1890, used punched cards to capture the US census data of that year. Such cards had been used since earlier in that century to "program" weaving looms to produce different complex patterns. Hollerith extended this mechanical technology, developing an electromechanical version in which electrical contacts through holes in the card selected combinations of conditions and totaled the number found. This permitted implementing a wide range of general purpose accounting practices, so the machines were widely used in business.

Notice that for Hollerith, and for the abacus, numbers are represented precisely using the traditional positional notation, in which a number, X, is represented by a series of integers,

$$a_k, a_{k-1}, \ldots a_1, a_0$$

and is evaluated in its base, b (which might be 10 or 2 or …), by adding up the series

$$a_k * b^k + a_{k-1} * b^{k-1} + \ldots$$
$$a_1 * b + a_0 .$$

For the slide rule, the precision of a number depends upon the size of the scale on which each number is represented, and the care with which the result of a

multiplication is read out. This is the characteristic difference between the digital world of mathematics and computation, in which facts expressed as numbers are preserved under copying and under most operations of arithmetic, and the analog world of physical phenomena, in which all operations upon measured quantities inevitably degrade the accuracy with which they are known.

But the idea of computation, performing an arbitrary sequence of computations, came into focus with Alan Turing's definition of an idealized computing machine. The purely conceptual "Turing machine" consists of an infinite memory tape, marked off into squares. The squares can hold symbols from a finite alphabet. The possible operations of the machine at each step consist of reading a square, writing a new symbol into the square, moving to the next square in either direction, and changing the internal state of the finite control network that is making these decisions. While this is a clumsy way to achieve practical computations, it achieved Turing's objectives. In 1936, he showed that while his machine could express any mathematical computation, not all computations that could be expressed could be solved by it. In particular, it could not solve the "halting problem" – given a program, does the program halt on all of its inputs? This answered the last of a famous set of questions posed by David Hilbert in 1928 as to the completeness, consistency, and decidability of formal mathematics.

Turing went on to become the leading figure in the successful British code-breaking effort conducted during World War II at Bletchley Park, near London. This effort used new computing machines to mechanically explore the logical implications of guessing or knowing parts of a longer encoded message. After the war, Turing remained involved in the development of general-purpose electronic computing machines. In 1950, he introduced the "Turing test," one answer to the question, "Can Machines Think?" His answer takes the form of a game. If one cannot distinguish whether one is holding a written conversation with a person or with a computer at a distance, then it is fair to say that the computer (if that is what it was) was "thinking."

The first electronic calculator, built as a prototype for a larger machine which was never finished, was constructed by John V. Atanasoff at Iowa State University in 1939. It was limited to solving systems of linear equations, but employed several technologies which were ahead of their time – vacuum tube switches, and condensers used as memory "bits" which had to be refreshed to keep the data stored, just as the bits in dynamic memory chips are refreshed in computers today. Konrad Zuse, in Berlin completed the first operational general-purpose programmable computer, the Z3, based upon electromechanical relays, in 1941. Towards the end of World War II, the Americans began to design general purpose electronic computers, stimulated by their extensive use of labor-intensive mechanical computing devices to build tables of artillery parameters and answer questions arising in the development of atomic weapons. The sources of the ideas involved and the parallel inventions resulting from the secret nature of these programs and even the secrets kept between the programs of allied countries, are an interesting subject for historians. But most of the structures which form the foundations of computing were created, step by step, during this period. At first, these machines were programmed by plugging wires into a detachable panel or by reading in the program from codes punched into an external paper tape, but the need to have the program stored in the computer soon became evident as calculating speed soon exceeded the speed with which paper tapes hcould

be read. The program loop, in which a series of instructions are repeated, with a conditional branch instruction to provide an exit upon completion of the task, was invented in several places. At first, subroutines consisted of processes separately encoded on different paper tapes but read in by multiple tape readers. Eventually the concept evolved of a "jump" from one program to another, with an automatic return to the next instruction in the calling program.

Most of the earliest machines operated upon integers, expressed either in decimal or in binary notation, depending on the machine. Leibniz is credited with the first discussion of binary representations of numbers (in 1679, but not published until 1701). Charles XII of Sweden favored using base 8, but died in battle (in 1718) before he could carry out his plan to require his subjects to use octal arithmetic. Numbers with a fractional part, such as 3.1416, had to be created by carrying along a scale factor for normalization after the calculation had finished. Floating point binary representation of numbers in computing hardware appeared as early as 1936, in Zuse's incomplete Z1 machine. Numbers were represented in two parts, a "mantissa" or number smaller than unity, expressed in binary positional notation, plus an "exponent" which in Zuse's case was the power of 2 by which to scale up the number. This separates the resolution with which a number is specified (the number of bits in the mantissa) from the dynamic range of the number. Readers may be already familiar with decimal "scientific notation" of numbers, such as Avogadro's number, 6.022×10^{23}, which are very large or small. Floating point hardware units in modern computers operate directly with such numbers.

One level above instructions lie applications, programs that solve problems people care about. At least this was true in the early era; now many layers of "system software," other programs that manage the machines for the benefit of the users and their applications, lie in between. Computer science was born in the analysis of the algorithms, or repeated calculations, used in the execution of the most frequently encountered applications and in the supporting processes of system software. During the 1950s the applications of computing in business expanded enormously. These applications all had a simple "unit record" flavor. The high cost of memory meant that most data was kept on low cost media, such as reels of tape, in which only sequential access is possible. Thus all the things one wanted to do with a piece of data had to be done when the "unit" of data was first encountered. Sorting received intensive study, since only when information is in a known order can we easily search for things, or retrieve them for use.

Once computers became available in modest quantities, it was natural to estimate the complexity of computing the quantities sought in the most popular applications. The fundamental measure of complexity would be the time required to obtain a result, but often some other operational quantity provides a more relevant or a more easily generalized measure. For example, consider sorting large amounts of data into ascending order so that they can later be easily searched. In the 1950s, this data might be stored on a single long tape. With a read/write head, one could read two successive numbers on the tape, writing the smaller of the two back to a second tape, keeping the larger in a register, a storage location inside the computer that permits arithmetic operations to be performed easily. Then one reads the next number on the tape. The smaller of the two is written back on the second tape; the larger is kept for the next comparison. When we reach the end of the tape, the largest number in the

data set is stored. Subsequent passes along the two tapes, always reading from the tape just written, will move the second largest number to the position next to the end, and so forth, until all the numbers are sorted. This process requires comparing all possible pairs of numbers, so the number of arithmetic operations (here, comparisons of magnitude) is proportional to n^2 if there are n pieces of data to start with. Another measure is the number of times that a tape must be read, in this case n passes. Still another measure might be the length of the program required and the size of the internal memory used. This process, called a "bubble sort" for the tiny "bubble" of only two values that needs to be held in the computer, is the simplest known sorting algorithm. It might have been preferred when programming a computer on a plug-board was difficult and error-prone, but for any significant amount of data, more efficient sorts, that put more data into their final order per tape pass, were quickly discovered and employed.

Consider data, each of which can be labeled by a "key" that is at most a two decimal digit number, and which is punched on cards. Using punched cards, and Hollerith-style equipment, one pass through all the cards separates them into ten bins, using the equipment of that era. Stack the cards in order, sorted by the second digit, and another pass sorts them by their first digit. Now the complexity is 2 passes to sort any number of cards into order, from 1 to 100. Other, more complicated, strategies can deal with more general cases, but require log(n) passes where n is the number of data elements. An illustrative example of an algorithm with log(n) behavior, simpler to explain than a sort, is taking the transpose of a large matrix, stored on tape. A matrix, or two dimensional array, of data, can be indexed by its row and column number in the array. Thus we designate an element of a matrix of data, A, as $a_{i,j}$. The array might be written on a tape in row order,

$$a_{1,1},\ a_{1,2},\ \ldots a_{1,n},\ a_{2,1},\ldots.$$

The transpose of A is simply the same matrix written by columns, with the first index varying more rapidly. How to do this with tapes, and minimal internal storage? One possible approach takes 4 drives and log(n) + 1 passes. On the first pass, we read the even rows onto one tape, and the odd rows onto another. On the second pass, we can read

$$a_{1,1},\ a_{2,1},\ a_{1,2},\ a_{2,2},\ \ldots\ a_{1,n},\ a_{2,n},\ \ldots$$

onto the third tape, and

$$a_{3,1},\ a_{4,1},\ a_{3,2},\ _{4,2},\ \ldots\ a_{3,n},\ a_{4,n},\ \ldots$$

onto a fourth tape, so two elements at a time on each tape are in the transposed order. On the third pass, we can get four elements at a time on each tape into transposed order. Finally, after log(n) + 1 passes, all elements are reordered, with essentially no internal storage required.

By the 1950's more complex structures than linear lists and more complicated optimizations than reducing the number of times a tape needed to be read had emerged. For just one example, Edsger Dykstra in the mid 1950s developed an efficient algorithm for finding the shortest path between any two nodes in a graph. This solved problems which arose in the automation of the design of computers. A graph is a mathematical structure consisting of points ("nodes") and links between the points ("edges"), which can be described by adjacency tables, lists of which nodes are connected by the edges. Such lists represented the first flowering of data structures, efficient ways of representing complex mathematical objects in computer memory, of traversing them in a search, and of changing them. The algorithm mentioned

introduced a technique now known as "breadth-first search." For example, if all links in the graph are of equal length, we need to find the path with the fewest steps to any point from some original point. One first finds the trivial path to each neighbor of the origin. Then one finds the shortest path to each neighbor of the shell of neighbors reached in the previous step. This process repeats until the desired destination is reached. Each point needs to be analyzed only once, and the computing cost of the analysis at each node is proportional to the number of links out of that node. If there are N nodes in the graph, and an average of k links out of each node, the total number of computer instructions needed to perform this algorithm for a large graph will be proportional to the product of N and k.

To create such complex structures and algorithms, tools such as assemblers, interpreters, compilers, and linkers were required, and evolved in small steps. The first was what is now called a linker, a tool which decides where in the computer's memory to place the programs which will run, computes the locations of other programs to which the first program may wish to transfer control, and stores this linkage information in memory among the program's instructions. Assemblers are translators which map each statement into one or a few machine instructions, and use statements which closely match the capabilities of the underlying machine.
For an example of assembly language, consider adding two numbers which are stored in the machine and saving the result immediately following them. In the simplest computers (like a modern-day pocket calculator), addition might be performed in an "accumulator" register, and the assembly language would read something like:

```
LDA  2000
ADD  2001
STA  2002.
```

A translation of this could be "load into the accumulator, the contents of memory word 2000. Then add to whatever is in the accumulator, the contents of memory word 2001, leaving the result in the accumulator. Then store whatever is in the accumulator into memory word 2002. Assembly language, and the machine instructions that it maps into, can get much more complicated than this example. But the advantage of having a compiler, which generates a correct sequence of machine instructions for an algorithm which is expressed in the form of a formula or equation is obvious. It is much easier to write C = A + B, and let the computer itself, through its compiler program, handle the details.

Interpreters fill an intermediate role. Assemblers and compilers create a correct sequence of machine instructions that are linked with other routines as needed, loaded into the computer and then run. With an interpreter, the original "high-level" or human readable description of the algorithm is executed line by line. The interpreter program generates the machine instructions on the fly. The FORTRAN language and its compiler were developed by John Backus and others, and introduced by IBM in 1957. LISP, an interpreted language for manipulation of logical expressions was invented by John McCarthy about 1958 (with a compiler as well). Many other languages have been invented since, but few survive. These original languages survive to the present day in highly refined forms.

It is hard today to appreciate how slowly the world began to recognize the truly revolutionary impact of digital computing during this period. Vannevar Bush, who had directed all scientific efforts in support of the war effort from Washington, DC

during WW II, in 1945 published a vision of the impact of future technology which was awesome in its breadth and prescience, yet intriguing in its blind spots and omissions. He recognized that many technologies could continue their steady acceleration of capabilities that the wartime effort had demonstrated, but did not see that digital computing was unique in its applicability to essentially all the problems that he considered. Before the war, Bush, a professor of electrical engineering, had built some of the first analog electronic computers. He took, as the main challenge to technology in the remainder of the 20th century, managing all of mankind's scientific knowledge as well as keeping abreast of its accelerating rate of accumulation. He recognized that the storage of this information could be miniaturized dramatically. Computers could help in searching through all of it, especially by memorizing one's favorite paths and half-finished connections between the works of one's colleagues. These ideas have only recently been given concrete form in the hyperlinks and bookmarks used in browsing the World Wide Web, as described below. He also predicted that speech could be recognized and turned into written text by computers, and that the content of the stored information in computers could be "mined" for its meanings. But Bush thought that data miniaturization might be achieved by pursuing the analog technology of microfilming. The device he visualized as aiding him in the perusal of all this knowledge looked rather like a microfilm reader and was to be found in the library. He never envisioned data networks, which now make it possible to access information in one place from almost everywhere else, or long term digital storage of information. And he might be surprised today to see that computing is being used by all the population, not merely scientists and scholars.

The age of algorithms (1960 – 1980): still a time for specialists

Users of the first computers soon separated the tasks of composing a program for their computation from those of managing the output and input devices required, reporting error conditions, and all the rest of the housekeeping chores involved. These were soon absorbed into "monitor" software, the progenitor of the modern operating system. The first monitors supported loading one task at a time, and unloading its results at completion. Utility routines, to support tape drives, typewriters or printers quickly grew in complexity. In 1963, IBM replaced its separate business and scientific computers with a single family of computers, spanning about a 20:1 range of performance, the System/360. Its operating system, OS/360, was the largest software project ever tackled by that time. Achieving all of its initial objectives required several years beyond the appearance of the first IBM/360 computers. Still another layer of program became evident with the System/360. Under a common set of computing instructions, all but the highest performance models used microcode, instructions resident in the computer, to actually implement the instructions executed on the different hardware of each machine. In this way the same program could run on all of the family's computers, but at different speeds, over a range that approached 200:1 by the end of the 1960s.

The first multi-user system, which could share the computer's facilities among multiple continuing tasks, was MIT's CTSS, developed around 1963. This could support 32 users, gave rise to the phrase "time sharing," and introduced for the first time the possibility of collaboration between users facilitated by the computer. UNIX and its programming language, C, developed together at Bell Labs starting around 1969 took a different approach to achieving the ability to support many different

computer types. C, which is still widely used, offers both elegant logical structures and the ability to control features of the computer such as individual bits in memory and registers. UNIX is written almost entirely in C, with assembly code appearing only in parts of the "kernel," those small portions of the code which are specific to a particular machine. UNIX was distributed to its users in its C source code form, and those users would compile it for their computers. While the original UNIX software was distributed under license, the availability of source code as the ultimate documentation stimulated others to create freely available "open source" versions of UNIX and its most popular utilities.

By the beginning of the 1970s, centralized business databases had become the major computer application. In these, data about an enterprise's operations is collected into stored digital records. Updating of this information is centrally controlled for accuracy and timeliness. The relationships between the records and their contents that constitute the processes of the business are made into programs which can produce reports for management or even initiate desired actions, such as when the inventory of some item is exhausted and should be replenished, or when bills are to be sent out. In the oldest systems, the relationships between data records were implicit, often conveyed by the inherent hierarchy of the file storage system in which the data were held. This reflected the performance tradeoffs of earlier storage technologies, and was highly inflexible. Subsequent efforts at architecting made the relationships external to the data, as a network of pointers. The modern view of database architecture, the "relational database," was first articulated in papers by E. F. Codd, starting in 1970. Codd treats the objects in the data base and the relations between them on an even footing. The objective is to capture the data model of the enterprise and its operations which the ultimate users understand, and to hide the implementation details. Products which followed this approach have been widely accepted, even though it took most of the decade of the 1970s for this to happen. The exposition of relational database by C. J. Date, in his authoritative textbook, "An Introduction to Database Systems," played a key role in the acceptance of these ideas. This appeared in 1974, at a time when there were no relational database products in the market, evolved through seven editions, and remains in print. Database design and a later development, object-oriented programming, an advance in software development methodology, occupy the interests of a large fraction of the computer science community.

A second radical change began to take place around 1960. The first proposals were made for shipping digital data over shared networks as packets, blocks of bits combining destination address information with the data to be transmitted. J.C.R. Licklider in 1962 proposed a "Galactic Network" of interconnected computers, and started a small organization at the Advanced Research Projects Agency (ARPA) of the US Department of Defense to realize this vision. In 1965 the first coast to coast computer linkages were tested. By 1970, computers at a dozen universities and research centers across the United States participated in the original ARPAnet, with remote log-in and program execution (telnet) the only service offered. In 1972, electronic mail was added to the offering, and the @-sign was assigned to signify an electronic address by Ray Tomlinson of Bolt Beranek and Newman. What was radical about these steps was that a global communications network, the analog telephone network, already existed. The telephone network worked with extremely high reliability and good voice quality, but was not especially suited for transmission

of digital data. Nor, for good economic reasons, were the owners and operators of the telephone voice network particularly interested in data transmission. But that would change.

The next field to be transformed from an analog form to digital was graphics, the creation, storage, and transmission of both static and moving images. Analog technologies for handling images were widespread, for example, photography, movies, and Vannevar Bush's favorite, microfilm. Since the 1930s, television made electronic handling of images possible. The NTSC standard for video transmission in color was established in 1950. The computer screen could be treated in either an analog or digital manner. The first method used was writing on the screen with a moving spot of light, just as one draws on paper with a moving pen. This sort of analog line drawing, called "vector graphics," was employed by Boeing around 1960 for computer generated drawings to see how average-sized pilots would fit into the cockpits of future aircraft, giving rise to the first use of the term "computer graphics." Alternatively, the computer screen could be treated as a matrix of possible spots at a fixed resolution. These spots, called pixels, could be displayed from separately computed values stored in a computer memory. While this was costly and clumsy at first, hardware to do it soon became available, fast, and inexpensive, and techniques to create highly realistic images by simple calculations were invented throughout the 1960s and 1970s, so that vector graphics had disappeared from use by the end of this period.

Artistic competition and performance have played an important role in the subsequent evolution of computer graphics. The first computer art competition in 1963 brought out SKETCHPAD, written by Ivan Sutherland at MIT. And the annual SIGGRAPH competition for computer animated short videos for many years was the first appearance of techniques that soon would show up in movies and commercial art of all types. Sutherland and Bob Sproull in 1966 combined a head-mounted display with real time tracking of the user's position and orientation to create a virtual reality environment. The early graphic systems used vector displays, but pixel oriented techniques dominated from the mid-1970s onwards. The first advances were very fast methods of interpolation and shading, to obtain realistic-looking curved surfaces. Next ray-tracing methods permitted realistic illumination of an object in the presence of multiple light sources. "Z-buffers," which computed images in depth, then allowed the objects closest to the viewer to be visible, blocking things behind them, made possible real-time rendering of complex scenes. A final technique is texture mapping, in which a simulated surface texture is wrapped around the surface of a modeled object.

During the 1960s the first serious attention was given to the human interface to computers. In its most effective period, 1963 to 1968, Doug Engelbart's Augmentation Research Center, a large research group then located at the Stanford Research Institute, invented the mouse and explored early forms of most of the elements of today's graphical user interfaces. Engelbart shared a vision with Licklider and Robert Taylor, the two leaders of the ARPAnet effort, of computing as a means to enhance man's ability to do scientific work through collaboration with possibly distant colleagues. As befits innovations in collaboration, an area that was not widely appreciated this early, the impact of Engelbart and his group was felt not through their papers, patents, or products, but by public demonstrations and through

the migration of the people of his group into the laboratories at Xerox Parc, Apple and other companies. His most famous public demo, given at the Fall Joint Computer Conference in 1968, in San Francisco, can still be seen in streaming video at http://sloan.stanford.edu/mousesite/1968Demo.html . In 90 minutes, Engelbart demonstrated, among other things, videoconferencing, the mouse, collaboration on a shared display screen, hyperlinks, text editing using direct manipulation, and online keyword search of a database.

During this period the theory of algorithms advanced greatly, reducing many problems to their most efficient forms. Texts written then are still authoritative. An outstanding example is D. E. Knuth's exhaustive exploration of the "Art of Computer Programming," a three-volume series which first appeared in the mid-1960s and continues to be updated to the present. It is the most exhaustive of the basic texts (for others, see References) in providing historical and mathematical background. An extensive literature of problem complexity developed. Some of the first questions addressed were the inherent complexity of arithmetic itself. If we consider the multiplication of two numbers, each with 2n bits of information, we might think that inherently $4n^2$ bitwise multiplication operations are required. But by decomposing each number into its high order and low order bits, e.g.

$$u = u_1 * 2^n + u_0 ,$$
$$v = v_1 * 2^n + v_0 ,$$

we find that the product u*v can be written as

$$uv =$$
$$= (2^{2n} + 2^n) * u_1 * v_1 - 2^n * (u_1 - u_0) * (v_1 - v_0) + (2^n + 1) * u_0 * v_0 \quad .$$

This reduces the problem of multiplying two 2n-bit numbers to three multiplications, each of two n-bit numbers, followed by some shifts and then addition. Not only can this provide a number with more precision than the number of bits which can be multiplied in the available hardware, it is the basis for a recursive procedure. To multiply two numbers, first split each in half. To multiply the halves, again split each part in half, continuing until the result can be obtained by looking it up in a small table or some other very fast method. If we can ignore the cost of shifting and addition, as was appropriate with early computers for which multiplication was very slow, the time required to multiply the 2n-bit numbers, T(2n) is now given approximately by 3*T(n). Solving this as an equation determining T(n) we find that T(n) is proportional to $n^{\log 3 = 1.585}$. This is better than the n^2 time cost that we began with, at least for sufficiently large numbers, when the overhead of shifting and adding that we have neglected can in fact be overcome.

This sort of recursive approach, both as algorithm and as analysis, pervades the computer science results of the 1950's and 1960's. An even more surprising result of this type is that matrices can be multiplied with a savings in the numbers of multiplications by a recursive decomposition into the product of smaller matrices. V. Strassen first showed that when the product of a pair of 2n x 2n matrices is factored into the product of n x n matrices, a clever method of combining the smaller matrices before performing the multiplications gave the final result with only seven matrix multiplications, instead of the eight that would be expected. Thus the time to multiply n x n matrices can be asymptotically proportional to $n^{\log 7 = 2.81}$, rather than n^3. Much more complicated rearrangements were subsequently discovered that reduce the asymptotic limit to $n^{2.376}$. The true limiting cost may be even lower, if additional

schemes can be found. The overhead in the rearrangements used in either method, however, is large enough that these recursive methods of matrix multiplication are seldom used in practice, and remain curiosities.

The most widely used results of this phase of algorithm exploration are data structures and efficient methods based on them for searching, sorting, and performing many sorts of optimization of systems described as graphs or networks. (refs: Aho, Hopcroft, Ullman; Tarjan) These became fundamental techniques as soon as computers had enough memory to permit performing these calculations entirely internally. The analysis, however, now takes on a probabilistic character, with several new aspects. Consider sorting a large array of numbers into ascending order, all within memory. One way in which this can be done quite effectively is a recursive algorithm known as Quicksort. The critical piece of Quicksort is a routine to partition an array of numbers. We first select one member of the array as a "pivot" element. Next move the elements in the array into two contiguous groups in memory. The lower group consists of those elements smaller than or equal to the pivot, the upper is those larger than the pivot. Finally, place the pivot between the two groups. Then apply the partitioning routine first to the lower and then to the upper group, recursively, continuing until partitioning results in either just one element, or a group of equal elements. With good programming, this can sort large arrays extremely fast, but it is sensitive to the original ordering of the numbers and the choice of the pivot elements.

Suppose that we are consistently unlucky in the choice of pivot elements, and each partitioning of n elements leaves us with a lower array of n-1 elements, the pivot, and an empty upper array. Then the process is no more efficient than bubble sort, costing n^2 comparisons. In fact, this is exactly what might happen if the array is already nearly sorted, with only a few elements out of place, a common occurrence.
On the other hand, if each partitioning divides the n elements considered into two equal groups of roughly n/2 elements, the process will terminate in log n steps, each involving n comparisons. So we estimate the best case cost to be n*log(n) and the worst case to be n^2 comparisons. It is natural to ask what is the average cost of a particular sorting algorithm on many different arrays of data. To answer that question, we need a probability measure over initial orderings of the arrays of data. If all orderings are equally likely, the answer is that almost all orderings will sort with roughly n*log(n) comparisons using Quicksort, so the average cost is indeed proportional to n*log(n). The algorithm is typically improved in two ways. One can devote extra time to choosing good pivot elements, or one can employ randomness to ensure that any initial ordering can be sorted in n*log(n) comparisons. The second strategy simply consists of choosing the pivot element at random before starting each partition step. This will produce nearly balanced partitions even with initially ordered data. This characteristic, that the worst-case and average computational cost of an algorithm differ widely when the problem gets large, is quite common. The use of a randomly chosen element to guarantee good performance with high probability in such an algorithm is also widely applicable, even though it may in special cases sacrifice opportunities for still further reduction of the running time.

After algorithms were discovered for most of the problems which can always be solved in a time proportional to a small power of the size of the problem description, attention shifted to "intractable" problems. Classifying intractable problems is a

subtle question, in which the mathematically rigorous distinctions may not always be relevant in practice. Obviously the potentially insoluble problems, such as the halting problem, are "intractable." There are also many problems for which it is believed that for the worst case inputs, computing time for a problem of size N will always increase as $\exp(N/n_0)$ for some constant, n_0. That sounds pretty intractable, since even if small enough problems can be solved on a sufficiently powerful computer, adding a few times n_0 elements to the problem might increase the cost of a solution by ten or more times, making it infeasible to solve in practice.

However, in the best known examples, the so-called NP-Complete problems, proof of exponential worst-case complexity still eludes us, and good approximations are often able to reach "good enough" or even provably optimal solutions in all but a very small fraction of the problem instances. NP-Complete problems, strictly speaking, are defined as decision problems, with the possible answers "yes" or "no". If one is presented with the problem definition and a proposed proof of a "yes" answer, this can be checked in time which is only polynomial in N. Thus a hypothetical highly parallel computer, in which each parallel unit checks just one possible proof, can solve the problem in polynomial time. If any unit says "yes," the answer is "yes," and if none do, the answer is "no." Unfortunately, the number of proof-checking units required for solution by this "nondeterministic polynomial-time" (NP) type of computer is exponential or greater in N. The NP-Complete class of problems are problems solvable in NP that can be converted into one another by a transformation whose cost is polynomial in N, and thus does not increase or decrease the asymptotic cost of their solution.

A typical example is the "traveling salesman problem," in which the salesman must visit each of N cities to sell his wares, and then return to his starting point. To optimize this, one seeks the shortest "tour", or path through N points whose pairwise distances are known, returning to the first point at the end. This optimization problem leads to a decision problem ("Is there a tour which is shorter than L in length?"), which is NP-Complete. To solve the decision problem, we might list all possible sequences of cities, add the lengths of the steps from city to city, and stop whenever we discover a tour shorter than L. But there are $(N-1)*(N-2)*(N-3)\ldots = (N-1)!$ possible tours, a number growing slightly faster than $\exp(N)$. Although there are many ways to eliminate candidate tours that are obviously longer than L, as far as anyone knows there is no procedure guaranteed to find a tour shorter than L or to prove that none can be constructed without considering of order $\exp(N)$ tours. S. A. Cook first expressed this notion of a class of comparably difficult problems; R. M. Karp then showed that a standard reduction method could reduce a great many widely studied problems into the class NP-Complete; and the list has now grown to thousands. See the book by Mike Garey and David Johnson for an early survey of this work. Johnson's columns in the Journal of Algorithms between 1981 and 1992 have greatly extended the survey.

At the same time that the list of "intractable" NP-Complete problems was growing rapidly, engineers and applied mathematicians were routinely solving some of these problems up to quite large sizes in pursuit of airline scheduling, inventory logistics and computer aided design problems. "Solving" in these communities meant obtaining a feasible solution that, while perhaps not the absolute optimum, was sufficiently good to meet product quality and deadline requirements. Heuristic

programs were developed which were demonstrably better than manual methods, with running times which increased as some power of N, the problem size, but offered no guarantee of solution optimality. Also, it was found that exact algorithms, which only halt with a provably optimal solution, may often give good intermediate results before halting, or halt in acceptable amounts of time. Identifying the problem characteristics which make heuristic solution possible was, and to some extent remains, an unsolved problem. However, it became clear that the spread between worst case limiting solution cost and typical cost is wide. When there is a natural ensemble of instances of an NP-Complete problem with similar overall characteristics, one often finds that almost all observed solution costs are polynomial in N;.

In the late 1970s, several groups recognized a connection between a commonly used class of heuristics for optimization and the simulation methods used in studying the properties of physical systems with disorder, or randomness, such as alloys or liquids. Iterative improvement heuristics are used to find reasonable configurations of systems with many degrees of freedom, or parameters. One takes a feasible but non-optimal configuration, picks one of the parameters which can be varied, and then seeks a new setting of that parameter which improves the overall cost or quality of the system, as measured by some cost function. When no further improvements can be found, the search halts, in what may be the optimum but most of the time is some sort of local minimum of the cost function. The connection is simple: The degrees of freedom are the "atoms" of the physical system. The constraints and the cost function correspond to the energy of the system. The search carried out in iterative improvement is very similar to the modeling done of the evolution of such physical systems with one important difference. Physical systems have a temperature, which sets the scale of energy changes which an "atom" may experience as the system evolves. At high temperatures, random motions are permitted. At lower temperatures, the atomic rearrangements naturally lead to system organization, and if the limit of zero temperature is approached slowly enough, the system will only be in its lowest energy state or states. In practice, the introduction of temperature or "simulated annealing" in which the temperature is slowly lowered to zero during an iterative improvement search leads to dramatic improvements in the quality of heuristic solutions to large problems, or to reduced running times. While there are theorems that guarantee convergence of this search process to a ground state, apparently they continue to require exponential cost. This connection to statistical mechanics has also permitted, in some cases, accurate predictions of expected results for an ensemble, or class of similarly prepared problems, such as the expected length of a traveling salesman's tour. .

1980s to the present: the customers take over

The end of the age of computing as a specialist field and its present position in the center of the world's economy as essentially a consumer product is a result of Moore's Law, the empirical observation that successful mass-produced electronic components increase in capability by a constant multiple every year. This is a social or business "law," based on self-fulfilling prophecy, not a law of physics. Applying a constant multiple year after year yields an exponential rate of growth of the power of a single chip. Thus the first memory chip with 1024 bits was introduced in 1971, the first chips with 1 Mb about 10 years later, and chips with 1 Gb are now in common use. Similar exponential growth, albeit with differing rates, is seen in computer logic,

storage technologies such as magnetic disks, and even batteries. Batteries are critical for portable computers and cellular phones. They have increased in capacity by about a factor of 2x every ten years for the past fifty years. Their "Moore's Law" is so glacial because batteries do not benefit directly from the technologies that make circuits smaller every year. Instead small improvements in chemistry must be turned into safe, reliable manufacturing processes for the industry to move ahead. Still, the economic feedback from success in selling longer-lived batteries has driven a long term exponential rate of improvement.

The key development in computing as a commodity was the appearance of single-chip microcomputers, beginning with such as the Intel 4004 and 8008 in 1971 and 19782. These were limited in function, performing arithmetic on only 4 or 8 bits at a time, and intended as building blocks for calculators and terminals. With the Intel 8080, a microprocessor with capabilities and performance approaching the minicomputers of the same era was available, and quickly incorporated into simple "hobbyist" computers by companies such as MITS (the Altair), Cromemco, and North Star, none of which survived very long after 1980. The earliest machines offered only versions of BASIC and assembly languages for programming, with typically an analog tape cassette as the permanent storage medium.

The Apple II, which appeared in 1977 based on the Mostek 6502 8-bit microprocessor, offered a floppy disk as a permanent storage medium with more nearly random access capability. But only in 1979, with the introduction of Visicalc, was it evident that such simple computers could be useful to customers with no interest in the technology. In Visicalc, developed by Dan Bricklin and Bob Frankston, the user could enter numbers or expressions into a two-dimensional array form that was already familiar to accountants and business students, and easily learned by anyone needing to manage accounts or inventories. Changing any quantity in the array triggered updates of the outcomes, so hypotheses could be explored directly. Programming by embedding simple formulas directly into the table of data proved unexpectedly easy for non-programmers to understand and exploit. Note that Visicalc was preceded by work on "direct input languages," such as IBM's Query by Example (QBE), which for the preceding five years or more had allowed a user to complete a familiar form and submit it to a time-shared computer to automatically simulate a hypothetical business process. But QBE and its extensions operated in a mainframe environment, did not facilitate user programming, and never caught on. The immediacy of the personal computer environment made a huge difference.

Apple II plus Visicalc established the viability of this class of equipment and the one-user focus. The microprocessor-based computers had come nearly as far up the evolutionary scale in 6 years as mainframes and minis had moved in the 35 years since 1945. The IBM Personal Computer (PC) was not a breakthrough in hardware when it appeared in 1981, but the time was right and it was widely marketed and well-supported. It added a simple monitor-type operating system (Microsoft's DOS), soon offered hard disks, and was sufficiently open that other manufacturers could easily clone it and produce compatible copies. The PC added one contribution to the computing vernacular which is almost as well-known as the "@" sign of email on the Internet. This is the combination "ctl-alt-delete", which causes a PC to restart when all three keys are pressed simultaneously. It was introduced by David Bradley as a combination which would be impossible to produce by accident,

During the 1980's rich graphical user interfaces or GUI's evolved on single-user systems, with the mouse as an intuitive means of selecting actions or items, and various extensions of selection, such as dragging objects to new places and dropping them suggestively on top of applications to be performed on them, came into use. These interfaces, first made into products at XEROX PARC, were most widely available on Apple products or UNIX systems until the early 1990s. In time they became generic to all systems with a single primary user, of which the PC's by the mid-90's had become the dominant presence because of their low cost and wider set of applications.

The final software layer, the "browser" through which one gains access to content of all types in the world of the Internet, has largely eliminated the need for many computer users to care much about the computer or even the operating system beneath. The first popular browser, NCSA's Mosaic, and its associated network protocols appeared during the early 1990s, a time in which multiple approaches to locating Internet content coexisted. Commercial versions, Netscape and Internet Explorer, supported by indexing and retrieval systems such as Yahoo, AltaVista, and subsequently, Google, based on extensive automated collections of Internet content, appeared after 1995. They have improved continuously and have made this approach a defacto standard.

An important development of the 1990s that made the widespread storage and communication of audio and video data possible is the creation of families of standard compression codes for low to high resolution still and moving images and their accompanying sound tracks or narration. This subject also has its roots in the 1940s, in Shannon's 1948 demonstration that the variability of a collection of data placed inherent limits on the rate at which it could be communicated across a channel, and on the extent to which it could be compressed by coding. A natural strategy for coding a message to keep it as short as possible is to use the shortest code symbols for the most frequently encountered symbols. This was intuitively obvious before Shannon's information theory provided a rigorous basis. Thus the Morse code for telegraphy (invented in the 1830s) uses "dot" for "E", "dash" for "T", "dot dot" for "I", "dot dash" for "A" and "dash dot" for "N," since these were felt to be the most common letters, and encodes a rare letter like "Q" with the longer sequence "dash dash dot dash". Huffman code, defined in 1952, orders all possible symbols by their frequency of occurrence in a body of possible messages and builds a binary tree in which the leaves are the symbol alphabet to be encoded. (An example is worked out in more detail as Appendix A.) Then the sequence of left and right hand turns, starting at the root of the tree, which bring you to each symbol in the message are the code for that symbol. This comes close to the Shannon lower bound when encoding sufficiently long messages that obey the presumed statistics. Adaptive codes were subsequently developed which build encoding rules as the data is first read, then transmit or store the "code book" along with the data being encoded. The most powerful of these approaches was introduced by Ziv and Lempel in 1977, and is practiced in the most popular data compression utilities.

The most obvious sort of data that will benefit from encoding to reduce its volume is text. Each ASCII character occupies one byte (8 bits). Simply recoding single characters, without attempting to find codes for commonly occurring groups of

characters will shrink a text document by roughly a factor of two without any loss of information, and a Lempel-Ziv encoder reduces a document such as the file containing this article by almost a factor of 4. Other codes, called "error-correcting codes," may make data files longer, if their purpose is to add redundancy in order to guarantee lossless transmission across a noisy channel, such as that used inside a storage device controller, which takes bits stored on a hard disk, detects them in a sensitive magnetic read head, decodes them into their ASCII or binary form , and transmits them at high speed to a computer.

Finally, codes need not preserve all the information in the data, if the objective is to save only the information which is to readily heard or seen by humans. Such perceptually-oriented lossy codes are widely used in telephony, and in digital still and video imaging. Because these codes are used in transmitting information, standards are critical to their success. It is not enough to have an efficient means of encoding a conversation to permit transmission at a very low bit rate. If no one but the sender is using the code, the conversation will not be heard. As a result, much effort is spent in groups representing the affected industries and academia assessing the merits of different encodings of voice, music and images, and a considerable publication record has been accumulated regarding the different approaches. The International Telecommunications Union, or ITU-T (formerly known as the CCITT, its French acronym) is the body which has set standards for data modems (devices that transmit digital data over analog telephone lines), for facsimile or halftone image transmission (G3 and G4 in the 1980s, then JBIG in the 1990s), for still images (JPEG) and for video (the various MPEG levels). JPEG stands for Joint Photographic Experts Group, and MPEG for Motion Picture Experts Group. The JPEG and MPEG groups have operated continuously since the late 1980s (JPEG) and early 1990s (MPEG), releasing standards in stages with the sponsorship of the International Organization for Standardization (ISO) and ITU-T.

The JPEG and MPEG codes are based upon a common set of building blocks, first for encoding a single picture, and then for taking advantage of temporal correlations within a series of pictures as occurs in video. A pixel, captured in color on the highest quality digital cameras, might contain 8 bits of information in each of three channels: red, green and blue. Our eyes are more sensitive to variations in the overall intensity of light than to changes in its color, so the first step in a perceptually-aware code is to transform the pixels from R, G, and B into one variable representing brightness and two representing color. Then the least significant bits of the color coordinates can discarded. The second step is to group the pixel data into small squares (generally 8x8 pixels) and Fourier transform them, using the efficient discrete cosine transform. Bits representing the higher spatial frequencies can again be discarded since they are less apparent to the eye. In this way the data in a single image can be reduced by a factor of 10 or 20 before any loss in picture quality is perceived. The MPEG standards have each targeted a specific application. Thus MPEG-1 is used for putting video into the compact disk format, which offers 650MB of storage. MPEG-2 was first used in TV broadcasting over digital satellites, so its principal objective was to maximize quality within a transmission bandwidth of 1.5 MB per second. MPEG-4, the simplest elements of which are just coming into use in 2002-3, aims at supporting much lower data rates, so that video can be downloaded and played in one step over the Internet. In MPEG-1, some frames are predicted by saving the difference between that frame and the preceding picture. In MPEG-2,

bilinear interpolation, in which the difference between a frame and the average of the frame before and the frame after it is encoded and saved, is also used. The three types of frame encodings, I for an image frame, P for predicted and B for bilinear are used in a repeating pattern, called a GOP (for group of pictures). For example, the popular DVD encoding, which is displacing the videotape cassette for home viewing of movies, employs MPEG-2 with a GOP of 12 frames, in the sequence IBBPBBPBBPBBI…. For efficient decoding, the frames are transmitted out of order, and are buffered in the decoder before reassembly. Additional tricks employed in MPEG-2 include identifying blocks in one frame which have moved without change from their position in a previous frame. This aids greatly in encoding objects which move in front of a static background. Each generation of video encoding, MPEG-1, -2 and -4, offers an inherent improvement of at least 2x in the density of the encoding at a computational cost of 4x or more per pixel.

A second theme in complexity studies emerged in the 1990s, after the flowering of algorithmic complexity, but its roots lie in the earliest uses of computers to crack enemy codes under the pressure of wartime. The secret work at Bletchley Park lead in 1943 to the Colossus, the first large, working, programmable computer, used to defeat the special codes used for transmissions between Hitler and his generals. The British government continued work on cryptography, the mathematics and algorithms of code-breaking and encrypted communication, long after the end of the war, but kept all of its efforts secret. As a result, several advances which have shaped today's computing environment were rediscovered in the middle to late 1970s by others, were patented, and were commercialized successfully in the 1980s. The British contributions finally received a small share of the credit when declassified in the 1990s.

The first widely used electronic encryption tool, DES, was based on Horst Feistel's 1970 Lucifer encoder. Originally it used a 64 bit key, but with key length reduced to 56 bits, it was accepted by the US government as the standard code in 1974 and became the world's most widely used encryption scheme for commercial use, such as electronic transfer of funds. The same key is used for encoding and decoding, so it must remain a secret. But how do you distribute a DES key to someone to begin communication? The problem is particularly acute in diplomatic and military security, where different keys are needed for each link and must be changed as often as daily. The need for secure electronic key exchange was what prompted invention of public key encryption, first understood in principle in 1969 by James Ellis, whose work remained secret until 1997, and independently discovered and published by Diffie and Hellmann in 1975.

The trick is that to transfer sensitive information from a sender to a receiver without ever exposing unencrypted material, the receiver must participate in the encipherment. We will share information, a "public key," that is of no value to a possible eavesdropper, but the receiver will keep his "private key" a secret from everyone, including the sender. Suppose I wish to receive a key for a high rate encoder, such as DES, from you, the sender. I have two encryption methods, M1 and M3, and you have another method, M2. These methods are known to me, you, and all who might wish to intercept our conversation. I take a "private" key, k, a large number that only I know, and encrypt it using M1 into x, then send it to you as my "public key". You then use x as the key for M2 to encrypt your message, m (in this case a DES key), and

send the resulting coded message, c, back to me. I then use my private key, k, with method M3 to decrypt the coded message, c, and reveal m. M1, M2 and M3 need to have a special "one-way function" property. If M1 is such that running it backwards to find k given x, and the M2 is such that without M3 and k to find m given x and c is not merely intractable but unthinkable, requiring thousands of years of computing time, then we have a sound encryption system. The only step left open in Ellis' original proposal was to construct efficient methods M1, M2 and M3. His colleague, Clifford Cocks found a viable group of methods in 1973, anticipating the algorithm published by Ron Rivest, Adi Shamir and Len Adelman in 1977.

Functions M1, M2, and M3 must be easy to compute, but hard to invert. Most traditional codes are based on functions like shifting or substituting within an alphabet. These are easily inverted, and thus messages and keys can be guessed with sufficient effort. The mathematical techniques leading to sufficiently strong "one-way" functions as are needed for M1, M2 and M3 come from the theory of prime numbers and modular arithmetic. An integer is prime if no other integer divides it exactly, leaving no residue. An integer is composite if it can be obtained by multiplying together some number of primes. Thus 2, 3, 5, 7, 11, and 17 are the smallest primes. 1 and 0 are omitted from the list because they are of no use in building composite numbers as products of primes. Modular arithmetic is arithmetic restricted to the range of values from 0 to some number, n, called the modulus. Thus if n is 5, 3+3 mod(n) is 1, the remainder when we divide 6 by 5.

Cocks' method M1 consists simply of multiplying together two very large primes, P and Q and sending their product, N = PQ, an even larger composite number, out as the "public" key, to be used in encrypting. The technology for encryption of the message will involve arithmetic modulo N. M1 is a good "one-way" function because there is presently no known way that an adversary, given N, could find the factors P and Q easily if P and Q are both large numbers, for example with a thousand bits each. In Cocks' proposal, M2 enciphers the message m into the code c by

$$\text{M2:} \quad c = m^N \pmod N \ .$$

To decipher c, I must find two more numbers (which need not be prime, and are not hard to construct), P' and Q', such that

$$P * P' = 1 \pmod N \ ,$$

and

$$Q * Q' = 1 \pmod N \ .$$

Then either of

$$\text{M3:} \quad m = c^{P'} \pmod Q$$

Or

$$\text{M3:} \quad m = c^{Q'} \pmod P$$

will decipher the original message, m. Raising m to the N-th power is done by repeatedly squaring it (mod N), so its computational cost is proportional to log(N). The RSA algorithm in use today differs from this in some details, but relies upon the same relationships. In RSA, and the "public key" framework, I send out a second

number, E, along with N (or put it into a public directory). E is found as one of a pair of numbers, D and E, such that

$$DE = 1 \pmod{(P-1)(Q-1)} \quad .$$

Now the methods M2 and M3 become

$$M2': \ c = m^E \pmod{N} \quad ,$$

And

$$M3': \ m = c^D \pmod{N} \quad .$$

Cocks' first method, M2, is a special case of RSA, in which E = N.

Although the original motivation for public key encryption was key distribution, it meets many other needs. For example, this allows me to create a digital signature, that could only have been created with my private key, yet can be checked by anyone holding my public key. With increasingly fast computers, RSA encryption is now possible in software. This comes just in time. Special-purpose computers have been able since 1998 to extract DES keys by exhaustive search in a few days. DES has been replaced by AES, a new, more complicated standard with much larger keys, for high bandwidth commercial use. The most important benefit of RSA and public key encryption methods in the future may be privacy in personal communications, not just security for financial transactions.

The number theory that RSA depends upon is subtle and different from the computational mathematics and the network algorithms that computer science first focused on. The discovery of these codes and the construction of secure systems to use them raised some new concepts in computational complexity, leading to a greater emphasis on the nature of proof and verification, and finally to the demonstration that verification could be accomplished without communicating information by any direct path. First, how does one check whether a given large number is a prime? One powerful approach dates back to Fermat, who stated in 1640 (without proof, of course, but both Leibniz and Euler later offered proofs) that for any prime, P, and a number A which is relatively prime to P (for example, $1<A<P$),

$$A^P \pmod{P} = A \pmod{P} \quad ,$$

or equivalently,

$$A^{P-1} \pmod{P} = 1 \quad .$$

To see why this is the case, consider the product of the first P-1 multiples of A:

$$A * 2A * 3A * \ldots * (P-1)A \pmod{P} = (P-1)! * A^{P-1} \pmod{P} \quad .$$

But the numbers produced by this sequence after the operation of taking (mod P) are all distinct, so the product is also equal to (P-1)! (mod P), and $A^{P-1} \pmod{P} = 1$ follows.

Note that although P a prime implies Fermat's result, the converse is not always true. There are composite numbers, P, which for some A give A^{P-1} (mod P) = 1. This can be checked by trying again with a different base A, but unfortunately, there are some very rare composite numbers Q, called Carmichael numbers, which give A^{Q-1} (mod Q) = 1 for all A which are not factors of Q. (The smallest of these is 561 = 3*11*17.) Primality tests which closed this loophole were developed by Solovay and Strassen in 1977 and by Rabin in 1976 and 1980. These used random selection of the base A plus methods which efficiently exposed composite factors to guarantee that the chance of p not being prime if the test succeeds is no more than 1 in 4. This leads to a new sort of proof – an "industrial strength" prime number which has a probability no more than 4^{-1000} of being composite can be verified with 1000 steps of such an algorithm, and could be used with high confidence as one half of a private key for RSA encryption. Examples such as primality testing have introduced a new type of proof, one in which certainty is approached at a definite rate, but, as in Xeno's paradox, never quite achieved. This gave rise within the discussion of complexity theory, to a class of "randomized polynomial" algorithms, or proof systems in which the chance of disproof could be reduced to an arbitrarily small value by a sequence of polynomial cost checking steps. Even though the problem of testing for primality has since been shown to be truly polynomial, the explicitly polynomial algorithm is slower than the best randomized search.

The next step in this evolution, was the discovery that verification of a claimed proof can be conducted without communicating any of the evidence, again by successively ruling out counter-evidence. This has similarities to the accomplishment of public key cryptography in achieving extra concealment through the cooperation of the receiver with the sender, but it goes a step beyond. Consider the following extreme example. Someone who has found a rapid way to test whether two large graphs are isomorphic or not wishes to convince me of the validity of his method but he is unwilling to share the details. Since proving or disproving graph isomorphism is in NP, this is an unlikely claim, but it is useful in modern theoretical computer science to imagine persons with such powers. They are termed "oracles." Suppose the oracle and I construct two very large random graphs with the same number of nodes and edges, which we call A and B. The oracle studies them and assures me that they are not isomorphic to each other. To convince me that this is true, he asks me to pick either A or B. Without letting him know which one I have chosen, I permute the labels and give it to him to inspect. If he is truly an oracle, he can quickly tell me whether I gave him a variant of A or of B. If he is only a mortal, the chance that he can do this correctly n times in a row is 2^{-n}. Almost certainly, I will quickly lose faith in him.

This sort of interactive proof was first discussed by Goldwasser, Micali and Rackoff in 1985, and has since been greatly generalized and extended. It can be used by mortals as well as oracles, and has important applications in cryptography as it provides distributed approaches to establishing trust and security, through a specialized variant called "zero knowledge" proof. Here is an example of a "zero knowledge" certification process. A one-way function just as difficult as finding the prime factors p and q of a large composite number n = p*q is finding the square root of a number mod (n). Suppose I know a large number u whose square mod(n) is v, and I have previously disclosed v. To prove my identity, I wish to convince you that I

know the value of u. Just sending u to you would reveal the secret and invalidate this for future use. Instead I pick a second secret random number, x, and send you

$$y = x^2 \bmod(n) \quad .$$

Now you pick either i = 0 or 1 and ask me to send $z = u^i * x \bmod(n)$, which you can check by calculating

$$z^2 \bmod(n) = v^i * y \bmod(n) \quad .$$

If I did not know u, but guessed that you would specify i = 0, I could just send you x. If I had guessed that you would specify i = 1, I would have sent you a false value, $y' = x^2/v \bmod(n)$ instead of $x^2 \bmod(n)$. Then when you ask me for z I send x, and your test shows that $x^2 = v*y'$. My ruse fails any time you choose the value of i opposite to what I was expecting. The chance that I could successfully anticipate which value of i you are going to specify and respond successfully for n trials using a new value of x each time, without knowing the value of u, is 2^{-n}.

In contrast to NP problems, which are often easy in practice, problems from number theory, such as factoring large numbers, are almost always hard. It was discovered recently that some of the best known problems in NP have thresholds, at which they also become almost always hard. Consider a classic NP-Complete problem, 3-SAT. The problem is described by N binary variables and a formula that consists of M clauses, each constraining three of the variables, selected at random. The decision problem is to decide if there is some configuration of the N variables which will permit every clause in the formula to be satisfied, or prove the contrary. The ratio of M to N is a natural parameter with which to characterize the problem as both N and M grow large, keeping the ratio M/N constant. Computer experiments using heuristic methods have shown that there is a threshold ratio, below which the formulae specified in this way are almost always satisfiable, and above which they are almost always unsatisfiable. At this threshold, the best heuristic search methods appear to take a long time on all instances of this problem. Away from the threshold most instances can be solved with relative ease. Since the sharp threshold between two well-defined "states of matter," (satisfiable and unsatisfiable), is identical to a phase transition in physics it is natural to ask if the methods of statistical mechanics can provide some new insights. Recent results are promising. Parisi, Mezard, and Zecchina find that the position of the threshold can be predicted very accurately and their techniques appear to lead to methods of solving even the almost always hard instances. It is still too early to determine if these methods will generalize to many other NP-Complete problems. A further challenge is to find ways of applying these methods to the problems of number theory.

As a new century begins, several trends are shaping the problems that will define computer science in the coming decades. Many years of Moore's Law have greatly reduced the initial cost of acquiring ever more powerful computers. As a result, large companies have noticed that the major components of the cost of computing lie elsewhere, in the numbers of people required to keep computers and their networks working effectively. Efforts to automate the management of computing as a sort of social "operating environment" are now underway, under slogans such as "autonomic computing." The network as a computing resource gives a new focus for parallelism

in distributed computing which takes advantage of the enormous number of processors connected in computers mostly used only for personal tasks such as word processing, and the enormous amount of memory and permanent storage that such distributed systems provide.

Will networked "appliances" be the next big thing to follow the personal computer? PC sales have clearly reached a plateau. At most, 50-100 million new PCs are sold each year in the world, while the number of cellular phones sold is greater than that. Some conclude, as a result, that further increases in the fraction of the world's population that finds a computer useful will come only when computers become "appliances." In this view, an appliance is something that accomplishes a single task extremely well, and offers controls appropriate only to that task, thus not confusing the customer in the many ways that a computer can be confusing. We can imagine single-purpose computing equipment becoming inexpensive enough to be considered "appliances." However, Moore's Law tells us that even better examples of such appliances will appear every year. In contrast to such disposable appliances, we expect traditional appliances such as a refrigerator or a washing machine to last us for at least 20 years. While many of us are now used to rapid obsolescence of our computers, telephones and cars, it is not clear that the world's entire population will agree to acquiring appliances with this sort of built-in obsolescence in order to make their lives more connected, or to manage their access to all the world's knowledge.

The physics of computation – what forms will it take in 2100?

Despite the enormous increases in density and speed achieved by Moore's Law and miniaturization, today's computers operate upon the same basic principles as the Turing machine and the relay computers developed in the UK, the US and Germany in the 1940s. Quite different principles may be employed to meet our needs for computation by the end of the current century. The need for exploring new models of computation is evident. Continued miniaturization will bring us, within another decade or two, to the point at which a transistor must be expressed within a single atom. The potential avenues being explored are opened up by studies of computation that attempt to understand its physical content in the most general sense: Is it subject to the laws of thermodynamics; do living organisms or the processes of genetic expression compute; does a quantum mechanical system have the power to compute quantities that are different in their expressive power than the standard number system?

Rolf Landauer opened the subject of the thermodynamics of computation in a 1961 paper, which asked if the irreversibility of each computing step would lead inevitably to the generation of heat. Consider adding two bits (mod 2). One starts with two bits, and ends with one bit, leaving no record of what the initial two bits were. Landauer's proposal was that this erasure would be accompanied by the generation of kT of heat, (where k is Boltzmann's constant, and T is the temperature of the computing element). While this is a tiny amount of heat, of no practical consequence compared to other electrical losses, Charles Bennett showed in 1973 that it could be reduced to an arbitrarily small value by performing the calculation reversibly. One need not, after all, erase the input bits, but can copy them to a "history tape," from which the computation could later be run backwards. John Hopfield in 1974 and Jacques Ninio in 1975 realized that the copying of genetic material from DNA subchains to RNA messengers for the production of proteins is in fact a reversible computation. In cells,

the gain from reversible computation seems to be accuracy, a sort of "kinetic proofreading," which trades off speed for orders of magnitude decreases in error rates, rather than simply saving energy. Subsequent efforts have identified many events occurring in biochemistry which have impressive computational power due to the fact that chemical transformations occur in parallel at many parts of a molecule. To date, however, no reversible or biochemical computing model with the generality and ease of programming of today's general purpose computer has emerged.

Interest in the computing capabilities of simple quantum mechanical systems has been driven from two directions. R. P. Feynman in 1982 suggested that since conventional computer algorithms employed to simulate the evolution of quantum systems were extremely complex and slow, it would be better to simply isolate and control quantum systems so that their states could simulate the behavior of more complex quantum phenomena. Then in 1994, Peter Shor showed that in principle, a quantum computer could factor large numbers in subexponential time. Because quantum procedures do not transform into one another in the same way as classical algorithms, this falls short of showing that a quantum computer can solve all NP-Complete problems in subexponential time, but it nonetheless has lead to an explosion of interest in quantum computing which continues to the present. Methods of quantum search, encryption, and communication with quantum error-correcting codes have been developed, at least in principle. Quantum computing proceeds in three steps: initializing a quantum system into a known state of its potentially many degrees of freedom; allowing it to evolve while interacting with external influences which do not destroy its internal coherence; and finally making a measurement, which destroys the quantum state by reducing it to classical numbers. The text by Neilson and Chuang gives a very readable and thorough overview of this rapidly evolving field. There are obviously problems of finding general, easy-to-program ways of expressing computations in this framework. More important is the fact that it is difficult to keep a quantum mechanical system isolated from unintended interaction with the rest of the world for long enough to complete a computation. Nonetheless, quantum computations involving a few quantum bits ("Qubits") have been performed and encrypted quantum communication over distances of many km have been achieved. At the present rate of improvement of a Qubit or two every year, it will not be too many decades before we may be forced to replace RSA encryption with its fully quantum mechanical successor.

References:

Note: in some cases, I have not tried to cite the first paper presenting a critical idea, but have instead included a later and more comprehensive reference by the originating authors.

R. E. Tarjan, "Data Structures and Network Algorithms," Society for Industrial and Applied Mathematics, Philadelphia, PA 1983

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. L. Stein, "Introduction to Algorithms, 2nd Edition," MIT Press, Cambridge, MA 2001.

D. E. Knuth, "The Art of Computer Programming," in three volumes, Addison-Wesley, Boston, MA. Begun in 1962, current editions 1997-99.

A. E. Aho, J. E. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Boston, MA (1974).

Two useful web data collections are:

http://philip.greenspun.com/ancient-history/history-of-computer-science
http://www.math.uwaterloo.ca/~shallit/Courses/134/history.html

And an excellent overview reference to the history of computing with emphasis on the machines and companies involved is:

Paul E. Ceruzzi, "A History of Modern Computing," MIT Press, Cambridge, MA 1998.

S. A. Cook, "The Complexity of Theorem Proving Procedures," Proceedings of the 3rd ACM Symposium on Theory of Computing," 151-158 (1971).

Richard M. Karp, "Reducibility among combinatorial problems," in R. E. Miller and J. W. Thatcher, eds., "Complexity of Computer Calculations," 85-103 Plenum Press, London, 1972

Michael R. Garey and David S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman, New York 1979.

S. Kirkpatrick, C. D. Gelatt, Jr. And M. P. Vecchi, "Optimization by Simulated Annealing," Science **220**, 671-680 (1983).

M. Mezard, G. Parisi, M. Virasoro, "Spin Glass theory and Beyond," World Scientific, Singapore (1987).

M. Mezard, G. Parisi, and R. Zecchina, "Analytic and Algorithmic Solutions of Random Satisfiability Problems," Science **297**, 812-815 (2002).

E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Commun. ACM 13, #6 (6 June 1970).

C. J. Date, “An Introduction to Database Systems,” seventh edition, Addison Wesley, Boston, 2000.

James H. Ellis, two internal reports, prepared for the British Communications – Electronics Security Group, documents the developments in the “closed community” of British encryption analysts from 1969 until about 1974. These were written in 1986, but remained classified until their release in 1997. Copies are available at: http://www.cesg.gov.uk/site/publications/media/ellis.pdf , and http://www.cesg.gov.uk/site/publications/media/possnse.pdf .

W. Diffie and M. E. Hellmann, “New Directions in Cryptography,” IEEE Transactions on Information Theory, IT-22, #6, Nov 1976.

R. L Rivest, A. Shamir, and L. Adelman, Commun. ACM 21, 120-126 (1978)

On discovery of ever-larger prime numbers, see a useful website: http://www.utm.edu/research/primes/ maintained by Chris Caldwell.

The work on primality testing mentioned in the text is described with more detail in Knuth, vol 2, pp 391 – 412.

R. Solovay and V. Strassen, SICOMP **6**, 84-85 (1977).

M. O. Rabin, “Probabilistic Algorithm for Testing Primality,” Journal of Number Theory, **12**(1) 128-138 (1980).

D. A. Huffman, “A Method for the Construction of Minimum-redundancy Codes,” Proc. IRE **40**(9) 1098-1101 (1952).

C. E. Shannon, ``A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379-423 and 623-656, July and October, 1948.

J. Ziv and A. Lempel, ``A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, Vol. 23, pp. 337--342, 1977.

J. Ziv, and A. Lempel, "Compression of individual sequences via variable-rate coding", IEEE Transactions on Information Theory, Vol. 24, pp. 530-536, 1978.

William B. Pennebaker and Joan L. Mitchell, “JPEG Still Image Data Compression Standard.” Van Nostrand Reinhold, Princeton, NJ (1993).

For current information on the latest JPEG and MPEG standards, see the respective organizations’ websites, at http://www.jpeg.org and http://mpeg.telecomitalialab.com/ .

S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof systems,” SIAM J. Computing **18,** 186-208 (1989).

RW Landauer, "Irreversibility and heat generation in the computing process," IBM J. Res. Dev., v.5, pp.183-191, (1961)

Bennett, C.H., "Logical Reversibility of Computation" *IBM J. Res. Develop.* **17**, 525 (1973)

Hopfield, J. J. *Proc. Natl Acad. Sci. USA* **77**, 5248-5252 (1980). Summarizing work on transcription accuracy done in the 1970s.

Hopfield JJ: **Kinetic proofreading: A new mechanism for reducing errors in biosynthetic processes requiring high specificity.**
Proc Natl Acad Sci USA 1974, **71:**4135-4139

Jacques Ninio, "Kinetic amplification of enzyme discrimination," Biochimie 57, 587-595 (1975).

C.H. Bennett and R. Landauer "Fundamental Physical Limits of Computation", *Scientific American* **253:1** 48-56 (July 1985).

Rather than reference the original papers on quantum computation by Bennett, Shor, Grover, et seq…, I advise the reader to consult an excellent compendium of the first 25 years' work in this field: M. A. Nielsen and I. L. Chuang, "Quantum Computation and Quantum Information," 676 pp., Cambridge University Press, 2000.

Appendix A:

Comparing Morse Code to a Huffman encoding:

Let's look at Morse code for the English alphabet with the hindsight of Shannon's theory of information and Huffman's prescription for building a frequency-based code. The first two columns of Table 1 give frequencies of occurrence of each letter in a sample of text. The following column contains the Morse code in dots and dashes for each character. The time in "beats" that it takes to transmit each character over a telegraph is calculated as follows: each dot takes one beat, each dash two beats. Between letters, we must pause for one beat. The resulting lengths are given in the next column of the table. The average length of a letter in Morse code in beats is 4.51, using their frequencies to weight the average.

Next we construct a binary tree representing the frequencies of the letters. The prescription is as follows: Find the two least frequent letters, and form a pair from them. Here Z occurs 3 times and Q occurs 5 times. Sum their frequencies and treat the pair as if it were a new character in the alphabet with weight equal to the sum, in this case 8. The same procedure leads to the pair, J and X, with total weight 13. Each pair has less weight than the next rarest letter, K, so we combine the pairs into a new object with weight 21, which we then combine with K. In this fashion we build a binary tree from the bottom up. The final result is shown as Figure 1. To form a Huffman code, we now read off the path down the tree from its top to each letter, treating a step to the low side (red lines in Fig. 1) as a "0" and a step to the right, or higher side (black lines in Fig. 1) as a "1". For example, the code for "E" is "100." To decode, we trace paths on the tree, extracting each letter. It is not necessary to insert pauses between letters as Morse code required – this is a so-called "prefix code," in which no code is the prefix of another letter's code. The last column of the table reports the length of the Huffman codes for each letter. The weighted average of these is 4.15.

Is this optimal? Shannon's work gives us the answer. The minimum code rate that the data permits is its entropy, defined as the expected value (averaged over all the symbols, weighted by their probability of occurrence) of $-p \log_2(p)$. Calculating this for the symbols and frequencies in Table 1 we find that the entropy of the data set is 4.13, and this provides a lower bound on bits per symbol. So Huffman code is quite close to optimal, and the Morse code is not that bad, either. It is certainly easier to memorize Morse code, with a maximum length of 4 characters than the up to 10 bit strings required by the Huffman procedure, but a computer would probably prefer to use the Huffman code.

Table 1:

Letter	Frequency of Occurrence	Morse code	code length	Huffmann code	code length
a	368	".-"	4	"1111	4
b	65	"-…"	6	"011100	6
c	124	"-.-."	7	"01010	5
d	171	"-.."	5	"10111	5
e	591	"."	2	"100	3
f	132	"..-."	6	"01011	5
g	90	"--."	6	"00001	5
h	237	"…."	5	"0100	4
i	286	".."	3	"1010	4
j	6	".---"	8	"1011000110	10
k	19	"-.-"	6	"10110000	8
l	153	".-.."	6	"01111	5
m	114	"--"	5	"00011	5
n	320	"-."	4	"1101	4
o	360	"---"	7	"1110	4
p	89	".--."	7	"101101	6
q	5	"--.-"	8	"1011000101	10
r	308	".-."	5	"1100	4
s	275	"…"	4	"0110	4
t	473	"-"	3	"001	3
u	111	"..-"	5	"00010	5
v	41	"…-"	6	"1011001	7
w	68	".--"	6	"011101	6
x	7	"-..-"	7	"1011000111	10
y	89	"-.--"	8	"00000	5
z	3	"--.."	7	"1011000100	10

Figure 1: (on separate paper)