COMPONENT SOFTWARE

For articles on related subjects *see* CLIENT–SERVER COMPUTING; DISTRIBUTED SYSTEMS; JAVA; and SOFTWARE REUSABILITY.

In some sense, all software is made up of its component parts, typically modules and lines of code (*see* MODULAR PROGRAMMING). But the idea of being able to build complex software from ready-made modules of reusable code has long bedeviled software developers. The difficulty is not in writing code good enough to be reused, but rather in how modular to make it and in how to store and classify code so that another programmer might find and reuse it.

Component software is designed to solve programming problems in small, modular bits, rather than in large, monolithic masses—think of a small tent rather than a skyscraper. A *component* is a small, lightweight application, tool, or utility that is designed to do a particular task (and only that task) and to do it elegantly and well. In this sense, the Unix (*q.v.*) system is composed of many separate components that manipulate files, process text, and so on. These components are all separate programs, however, and in the newer sense of the term, components are pieces that are not self-sufficient, but can be assembled into complete programs.

As with all early markets, components started out (1995–1997) as tools for software developers, but in a broad sense of the term, including commercial independent software vendors (ISVs), value-added resellers (VARs), systems integrators, and information systems designers within an organization. Already we have seen some applications, mainly lightweight versions of personal productivity tools, e.g. word processors, spreadsheets (*q.v.*), charts. These applications are intended for three markets: for developers who will incorporate them into larger applications (this is the component as building block), for hardware OEMs (original equipment manufacturers) who will use them as part of the desktop for *networked computers,* and for leading-edge users who are ready to try something new.

Eventually, it is possible that we might see components become an alternative software market (or even a replacement software market). For that to happen, there will have to be a lot more work on transforming existing monolithic applications into components (a process that has now begun). Also, developers (internal or commercial) will have to build components for organizations' business processes, so they can be just another one of the building blocks. That, too, is beginning to happen.

A component might be written as an ActiveX control, a Netscape plug-in, or even as some C or C++ code. Most of the earlier applications fall into this category. New components being started now are likely to be written as Java applets, given the pervasive presence of the Internet, the explosive growth of intranet platforms as the new place for corporate application development, and the overwhelming interest in the development community to pursue the current hot platform. However, component programming is made possible through interface standards that define how components may interact—how one component may exchange information with another. Components that adhere to such standards may be written in any appropriate language. One such standard is CORBA (Common Object Request Broker Architecture); another is Microsoft's Component Object Model (COM or DCOM, for distributed programming); another, for Java, is Sun Microsystem's Java Beans.

In the late 1990s, many components were sold to OEMs and developers, but first tested as downloaded freeware from the Internet. In fact, components might not have happened at all without the Internet as a free and nearly frictionless marketplace. We expect individual developer Websites and Web-based component marketplaces (like the one that IBM already has) to be the model for selling individual components. We would also expect new pricing schemes to be implemented as software developers, used to substantial revenues from larger products, try out new bundling schemes to continue to grow their revenue as the business model for creating and selling software changes.

Bibliography

1998. Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* Reading, MA: Addison-Wesley.

Amy D. Wohl

COMPRESSION, DATA

See DATA COMPRESSION; and IMAGE COMPRESSION.

COMPUTATIONAL COMPLEXITY

For articles on related subjects *see* ALGORITHMS, ANALYSIS OF; ALGORITHMS, THEORY OF; FAST FOURIER TRANSFORM; INFORMATION-BASED COMPLEXITY; MATHEMATICAL PROGRAMMING; NP-COMPLETE PROBLEMS; SORTING; TURING MACHINE; and UNDECIDABLE PROBLEMS.

Once we have developed an algorithm (*q.v.*) for solving a computational problem and analyzed its worst-case time requirements as a function of the size of its input (most usefully, in terms of the *O*-notation; *see* ALGORITHMS, ANALYSIS OF), it is inevitable to ask the question: "Can we do better?" In a typical problem, we may be able to devise new algorithms for the problem that are more and more efficient. But eventually, this line of research often seems to hit an invisible barrier, a level beyond which improvements are very difficult, seemingly impossible, to come by. After many unsuccessful attempts, algorithm designers inevitably start to wonder if there is something *inherent* in the problem that makes it impossible to devise algorithms that are faster than the current one. They may try to develop mathematical techniques for *proving formally* that there can be no algorithm for the given problem which runs faster than the current one. Such a proof would be valuable, as it would suggest that it is futile to keep working on improved algorithms for this problem, that further improvements are certainly impossible. The realm of mathematical models and techniques for establishing such impossibility proofs is called *computational complexity*.

For example, sorting n keys is a computational task that can be easily accomplished in $O(n^2)$ time by naive exchange algorithms such as *bubblesort*, while more sophisticated techniques such as *quicksort* and *mergesort* bring the time requirements down to $O(n \log n)$. Can we do better, or is $n \log n$ an unsurpassable milestone for sorting? Another interesting example is matrix multiplication. For a long time it was assumed that one needs n^3 operations to multiply two $n \times n$ matrices. In 1969, Volker Strassen showed that two $n \times n$ matrices can be multiplied by an ingenious recursive algorithm in $O(n^{2.81})$ operations! Over the past thirty years this exponent has undergone a breathtaking sequence of improvements, and now stands below 2.4. Where is this sequence of improvements going to end? Can we multiply two matrices in $O(n^2)$ time? Can we prove a lower bound of the form $n^2 \log n$, or, even more ambitiously, $n^{2.2}$, for the matrix multiplication problem?

A third example is the *traveling salesman problem* (Fig. 1), a problem that is popular and well-studied, as well as notorious for its difficulty. It is trivial to come up with an algorithm which, given an instance of the traveling salesman problem with n cities, will find the optimum tour in time $O(n!)$—just check all possible permutations of the cities. This algorithm, is, of course, all but unusable for any but the smallest instances: even for a modest instance with $n = 30$ cities, the number of tours to be examined is larger than the size of the known universe (or its age in picoseconds). A more detailed examination of the algorithm will reveal that the true running time is $O((n-1)!)$, since the starting city can be fixed with no harm to the correctness of the algorithm. It took some cleverness (and several decades from the time the problem was posed in the 1920s) to find a faster algorithm, requiring "only" $O(n^2 2^n)$ steps; this algorithm, discovered by Michael Held and Richard M. Karp, uses a *dynamic*

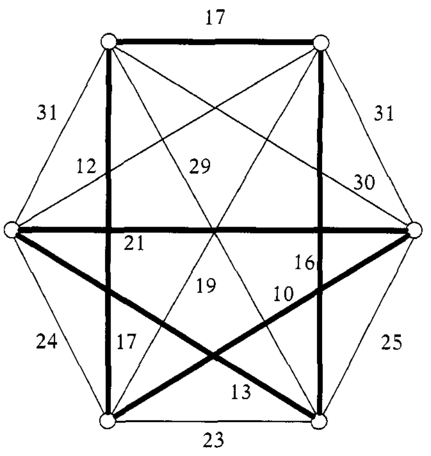

Figure 1. In the traveling salesman problem we are given a set of cities and the distances between them, and we seek the shortest closed tour that visits all cities. The optimum tour in this simple example is shown in bold, with total length 94. Because of the simplicity of its statement, its obvious appeal, and its maddening complexity, the traveling salesman problem has been studied extensively for decades, and it has been the testbed of every new algorithmic technique. Still, all algorithms known for it require exponential time in the worst case.

programming technique (*see* ALGORITHMS, DESIGN AND CLASSIFICATION OF) that patiently solves the problem for larger and larger subsets of the cities, using the results from smaller subsets to crack the larger ones, until the optimum tour of the set of all cities is finally identified.

Can we do better? To this date, there is no known algorithm that is guaranteed to solve the traveling salesman problem exactly for n cities faster than the dynamic programming algorithm. There are algorithms that are known *empirically* to solve quite large typical instances of the traveling salesman problem reasonably fast, and there are fast algorithms that somehow *approximate* the optimum solution, but there is no known algorithm that is guaranteed to return the optimum, and to do so in time that is *polynomial* in n—an algorithm with a running time such as $O(n^2)$, or $O(n^5)$. It is thus tempting to conjecture, and try to prove, that *the traveling salesman problem requires exponential time for its solution*, that all algorithms that solve it must spend exponential time for some infinite collection of instances.

As the reader may immediately suspect, the task of proving *negative results*, or *lower bounds* on the complexity of a problem, is usually a lot more intricate mathematically than just devising an efficient algorithm. Coming up with an efficient algorithm, however ingenious it may be, requires only that the algorithm be specified and analyzed. Proving a lower bound, however, necessitates that the prover must consider *the* *whole spectrum of all possible algorithms for the problem in hand,* and show that none of them does better than the specified bound; the difficulty of the task is obvious. Since its beginnings in the 1960s, computational complexity has been one of the most active research areas within theoretical computer science. However, despite hard work by some of the field's most gifted researchers, the development of sophisticated mathematical techniques, a few ingenious insights, and an ever-increasing understanding of the issue, it is fair to say that the difficulty of the task has heretofore prevailed: with very few and limited exceptions, lower bounds are still largely in the realm of conjecture.

General Models

The barrier separating polynomial algorithms from exponential ones, upon which we have stumbled in the case of the traveling salesman problem, is one whose significance goes beyond that problem. Polynomial-time algorithms, algorithms whose running time is bounded by a function like $O(n)$, $O(n^3)$, etc., form a substantial and important class of computations, broadly considered akin to the empirical concept of "practically feasible computation." Naturally, an $O(n^{1000})$ algorithm would hardy deserve to be called "practical", but such extreme polynomials never come up in practical situations. Typically, once a polynomial algorithm is discovered for a problem, a sequence of improvements ensues and the problem is eventually brought within the limits of practical computation. Unfortunately, there are many important problems for which, like for the traveling salesman problem, the best known algorithms are exponential in the worst case; it is these problems that have inspired the development of the main branch of the field of computational complexity, the one that deals with *general models of computation.* In contrast, problems such as matrix multiplication and sorting, for which the important open questions try to differentiate between different polynomial rates of growth, must be treated within specialized models of computation, within which there is some hope of making such fine distinctions.

The process of proving a lower bound on the complexity of a problem must start with a precise mathematical model for algorithms and their complexity. There are several useful mathematical models of algorithms, starting with the many variants of the Turing machine, proceeding to more down-to-earth models such as the *random access machine* (an abstraction of the von Neumann machine—*q.v.*), pointer machines, and many others. For each such model we have a way of evaluating the time required for the solution of an instance (in the case of the Turing machine, this is simply the number of steps the machine takes to come up with the final answer). This confusing diversity of models appears to add another layer of difficulty, besides the fundamental mathematical one, to the development of a theory of computational complexity. Fortunately, all these various models of computation have been proved to have computational powers that differ *only by a polynomial.* If a problem can be solved in polynomial time in any one of a wide array of models of computation, it can be solved in polynomial time in all of them. It is this fundamental fact, *the quantitative analog of the Church–Turing thesis* (*see* UNDECIDABLE PROBLEMS), that allows us to study the polynomial/exponential dichotomy in algorithms in a principled and model-independent manner. (It should be noted that this principle is not as universally accepted as the Church–Turing thesis; in fact, its most serious and credible challenge has come recently, as physicists and computer scientists have joined forces to define and study *quantum computing* (*q.v.*), a model of computation that exploits quantum mechanical phenomena to achieve, presently only in theory, apparent exponential speed-ups over conventional computers and models of computation.)

Complexity Classes

In computational complexity we classify computational problems into *complexity classes.* The most important complexity class is the set of all problems that can be solved by polynomial-time algorithms (by Turing machines, or algorithms in any other one of a broad set of standard models). This important complexity class is denoted P, for polynomial time. Actually, for reasons of convenience, uniformity, and tradition, complexity classes are comprised not of problems, but of *languages,* that is, sets of strings in some fixed alphabet such as $\{0, 1\}$ (*see* FORMAL LANGUAGES). Any computational problem of interest can be transformed into a corresponding language in a way that captures its complexity. For example, the traveling salesman problem can be captured by the language L_{TSP}, consisting of all strings of 0s and 1s which encode an $n \times n$ matrix of nonnegative integers (the distances between the cities) plus another integer B, such that there is a tour of the n cities of total length equal to B or less.

As we mentioned, it is widely conjectured that the language L_{TSP} is not in P. However, it does belong in a broader, albeit somewhat less natural, complexity class called NP, for *nondeterministic polynomial.* Any language in this class can be decided by a polynomial *nondeterministic Turing machine,* a hypothetical device that has the ability to make correct guesses. For example, to recognize a string in L_{TSP}, a nondeterministic Turing machine would correctly guess the optimum tour of the instance encoded, and check that its length is indeed below the given bound. A language belongs in the class NP if such a recognition algorithm —a guessing phase, followed by a polynomial-time checking phase—exists. This important class contains, besides all of P, the traveling salesman problem and many other notoriously difficult problems. It is widely believed that the class P is strictly included in the class NP (i.e. that there are problems in NP not in P); this conjecture, as yet unproven, is the most central, important, and well-studied problem in computational complexity. A proof of this conjecture would establish in particular that the traveling salesman problem cannot be solved by a polynomial-time algorithm; this is because L_{TSP}, along with a surprising variety of other languages encoding natural problems, has been shown to be *NP-complete*. A problem in NP is NP-complete if all other problems in NP reduce to it in polynomial time. If there are *any* problems in NP that require exponential time, all NP-complete problems *must necessarily* be among them.

Complexity classes go beyond NP. The class EXP contains, informally, all problems solvable by exponential-time algorithms. By a straightforward quantitative extension of the diagonalization proof which establishes that the *halting problem* is undecidable (*see* UNDECIDABLE PROBLEMS), it can be shown that there are problems in EXP that are not in P. EXP itself is a proper subset of the decidable languages. And it is known that EXP contains all of NP.

Complexity classes also deal with resources other than time, most significantly *space*. In analogy to P, PSPACE is the complexity class of all languages that can be recognized by a computer using an amount of memory (number of Turing machine tape squares, for example) that is bounded by a polynomial in the size of the input. Memory is a resource that is more powerful and robust than time (obviously, you can compute more things with 1,000,000 memory words and unlimited time, than you can with 1,000,000 instructions and unlimited memory). For example, PSPACE contains both P and NP (but is contained in EXP). Also, another sign of the robustness of space is that nondeterminism makes no big difference in the space domain, and nondeterministic machines can simulate deterministic ones with only *quadratic* increase in space (but exponential increase in time)—hence the absence of an NPSPACE class. Because of the power of memory as a resource, there are interesting tasks (such as the evaluation of formulas and the traversal of rooted trees—*q.v.*) that can be accomplished in *logarithmic* space, whereas no computational task of the kinds considered here, in which one must examine the whole input, can be carried out in logarithmic time. (Many computational tasks of the *database query* variety can of course be accomplished in logarithmic time, for example by binary search; there is a branch of computational complexity that studies these too.) The class of problems solvable in logarithmic space is often denoted L.

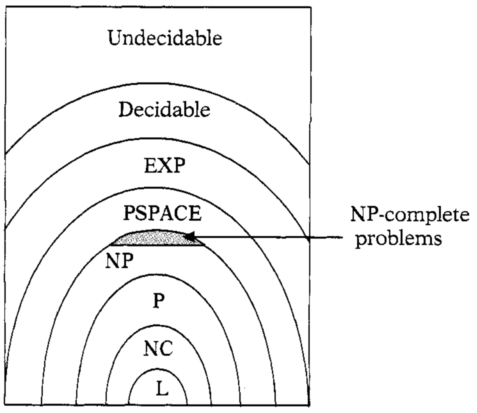

Figure 2. The complexity classes introduced in this article are depicted here as regions arranged according to the currently most prevalent view among experts in computational complexity. If a region A contains another region B, then the corresponding classes are known to contain one another in the same way. However, whether the containment is *proper*—that is, whether there is any space inside region A and outside region B—is for most of this map currently a subject of conjecture. There are a few known proper containments—for example, we can prove that EXP properly contains P, that PSPACE properly contains L, and that EXP is properly contained in the class of decidable problems.

There is an important and intriguing connection between space and *parallel time*: it turns out that, again within a broad range of "reasonable" models of sequential and parallel computation (*see* PARALLEL PROCESSING), the computational tasks that can be accomplished with a given amount of memory are closely related to the tasks that can be carried out in the same amount of parallel time—assuming that there are no limitations on the number of processors that are available. Another complexity class, called NC, is supposed to model feasible parallel computation more accurately: it includes all problems that can be solved in $O((\log n)^k)$ parallel time, for some fixed integer k, on polynomially many processors. This class is a subset of P, and is in fact believed to be a *proper* subset of P—as there seem to be many tasks which can be solved efficiently on sequential computers but cannot be successfully parallelized beyond some level. Alas, as with most of our more interesting insights related to complexity classes, that NC is different from P is currently yet another unproven conjecture.

Fig. 2 depicts the various complexity classes and their inclusions.

Other Aspects of Computational Complexity

In a certain sense, the first important results in computational complexity are the *undecidability* results proved by Alan M. Turing (*q.v.*), Alonzo Church (*q.v.*), Stephen C. Kleene, and many others in the 1930s and 1940s, establishing that certain problems cannot be solved by algorithms *at all*, however inefficient. When computers became available shortly thereafter, it became apparent that not all decidable problems deserve to be called "solvable," since excessive time requirements make many algorithms completely impractical. The current emphasis on polynomial time emerged in the 1960s from the experiences of researchers, most notably Jack Edmonds and Alan Cobham, trying to attack seriously hard problems in optimization and logic. The foundations of the present "complexity class" paradigm in computational complexity were laid by Juris Hartmanis and Richard E. Stearns in the 1960s.

Although research in computational complexity has ample internal unity, one can discern certain styles, trends, and research traditions. In the late 1960s a rich *axiomatic theory of complexity* was developed by Manuel Blum, in which complexity was studied in the abstract as a property relating recursive functions (*see* RECURSION) and computational resources that must obey a small number of common-sense axioms. *Structural complexity*, whose style is also influenced by the theory of recursive functions, studies complexity classes, various kinds of *reductions*, as well as the intricate connections between the two concepts. There have of course been several attempts to solve the major open problems of the field, of which the P vs. NP question is the most well-known and fundamental; many of these approaches redefine complexity in terms of *Boolean circuits* or other such primitive devices, in the hope of making the quest for lower bounds more concrete and tangible; other approaches to the P vs. NP question evoke the rich connections between computational complexity and mathematical logic. There is a research tradition of growing importance that uses computational complexity to study the foundations of cryptographic and other protocols (*see* CRYPTOGRAPHY, COMPUTERS IN), as well as of randomness, a study which often results in unexpected connections and insights into the more central problems in computational complexity. An interesting variant is *communication complexity*, which seeks to bound from below the amount of information that must be exchanged between two parties, each of which is in possession of a private input, in order to compute a complex function of the two inputs; communication complexity is often a useful tool in other subfields such as circuit complexity and VLSI complexity. There is also much research on using concepts from computational complexity, most often NP-completeness and its many variants, in order to understand better particular problems, and application areas, often problems in seemingly non-computational realms of the pure, applied, and social sciences.

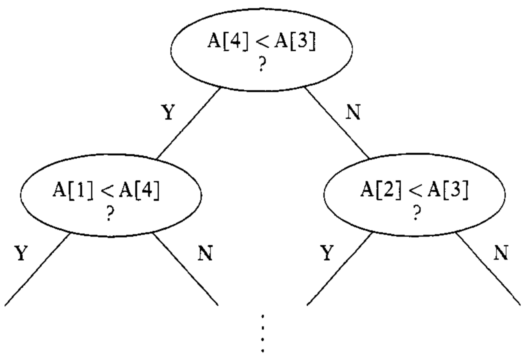

Figure 3. A lower bound for sorting by comparisons can be obtained within the *decision tree model* of computation. In this model, any algorithm that sorts n keys $A[1], \ldots, A[n]$ starts with a comparison of two keys (in our example, $A[3]$ with $A[4]$), branches to two new comparisons depending on the outcome, from these to two new comparisons, and so on. Thus, every algorithm in this model is a tree. (The analysis of the related model in which comparisons have three outcomes, $<$, $=$, and $>$, is very similar.) Notice that this model suppresses the instructions, present in any sorting algorithm, that move keys around according to the comparison outcome; since our goal is to prove a lower bound, such omission is legitimate. The complexity of this algorithm is captured by the *height* of the tree, the length of the longest path from the root to a leaf.

Each leaf of this tree must correspond to the outcome of the sorting algorithm, that is, to an ordering of the n keys. Since this algorithm must sort correctly all possible initial permutations of the keys, there must be at least one leaf corresponding to each possible permutation. Therefore, this tree must have at least $n!$ leaves.

We are very close to a proof of our lower bound: it is well-known (and easily provable by induction) that a binary tree that has height h can have up to 2^h leaves. So, a tree that has $n!$ leaves must have height at least the logarithm of $n!$ Now, a calculation shows that the logarithm of $n!$ is about $n \log n$, which completes the proof.

There are also branches of computational complexity studying the inherent complexity of problems such as matrix multiplication and sorting, in which the desired complexity distinctions are much finer than the fundamental one between polynomials and exponentials. Necessarily, such problems are studied within more specialized models of computation. For example, in order to study the complexity of sorting, we may want to consider a model of computation in which sorting algorithms are seen as *comparison trees*. In this model, an $n \log n$ lower bound for sorting *can* be proved (see Fig. 3). Complexity results within limited models of computation, such as the decision tree model, are often met with skepticism: there may be useful algorithms for solving the problem that are outside the limitations of the model. There are classical algorithms like *radix sort*, using primitives other than comparisons (such as array access); these algorithms succeed in sorting certain kinds of key arrays faster than $n \log n$. Moreover, there are recent algorithms which appear to sort general arrays of keys faster than $n \log n$ by manipulating the bits of the keys. The ultimate value of lower bounds in restricted models may be, ironically, that they point to the kinds of primitives that must be used in order to circumvent them! The related field of *algebraic complexity* seeks to answer complexity questions such as the matrix multiplication problem, within a model of computation in which the primitive operations are algebraic operations (additions, multiplications, and so on). *Information-based complexity* seeks to understand the complexity of computations involving real numbers in which the scarce resource is the amount of information on the precise values of the inputs required for carrying out the computation.

Finally, in computational complexity we do not attempt to evaluate the complexity of a single input or a single string; we are interested only in the complexity of whole problems and languages. Still, it is intuitively obvious that the bit string

$$x = 011011101110001011011011010101111011$$

is more complex than the string

$$y = 010101010101010101010101010101010101.$$

Kolmogorov complexity is an approach to computational complexity that attempts to capture this intuition by defining the complexity of a string to be the length of the shortest program (in some fixed programming language) that generates this string. For example, string y above is generated by the program `"print '01' 18 times"`, whereas there may be no such short program generating the string x. Kolmogorov complexity is a well-developed field which, interestingly, has often been a valuable tool to researchers in more mainstream aspects of computational complexity.

Bibliography

1979. Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-completeness.* New York: W. H. Freeman.

1992. Harel, D. *Algorithmics: The Spirit of Computing*, 2nd Ed. Reading, MA: Addison-Wesley.

1994. Papadimitriou, C. H. *Computational Complexity.* Reading, MA: Addison-Wesley.

Christos H. Papadimitriou

COMPUTATIONAL GEOMETRY

For articles on related subjects *see* ALGORITHMS, ANALYSIS OF; ALGORITHMS, DESIGN AND CLASSIFICATION OF; COMPUTER GRAPHICS; and GRAPH THEORY.

Computational geometry is the study of algorithmic problems involving geometry. Although the ruler and compass constructions of ancient Greek geometry were essentially algorithms for producing geometric objects, modern computational geometry begins with M. I. Shamos's 1975 Ph.D. dissertation, which solved several fundamental geometric problems and posed many more. Since the 1980s, computational geometry has been perhaps the most active area of algorithms research, and a recent bibliography lists over 8,000 relevant publications. The explosive growth of this field can be traced to the intuitive appeal of geometric problems as well as the wide range of practical applications.

Geometric problems arise in a variety of applications, some of which would not seem to have geometric aspects. VLSI circuits are described by overlapping rectangles of different materials. To prevent wires from short-circuiting, it is necessary to test designs so that no two rectangles intersect. The huge number of rectangles in a large circuit implies the need for fast intersection-detection algorithms. Mobile robots must find paths to a goal through rooms full of obstacles without bumping into anything. This can be more difficult than it might appear, as anyone who tries to move a piano through a door quickly discovers. Finite element methods (*q.v.*) used to simulate the performance of physical systems such as aircraft depend upon dividing the surface of the object into triangular regions, and effort spent in finding a "good" triangulation (such as the Delaunay triangulation described below) pays dividends in more efficient and accurate simulations. Database queries of the form "how many people are between 180 and 200 centimeters tall and weigh between 60 and 75 kilograms" can be thought of as asking how many points lie in a given rectangle, where the x-axis represents the height and the y-axis the weight. Finally, eliminating hidden lines and surfaces is typical of the geometric problems arising in computer graphics.

Computational geometry often deals with questions of how to compute various aspects of geometric structures. Many brute force algorithms for solving geometric problems can be improved upon by algorithmic techniques and sophisticated data structures (*q.v.*). To a larger extent than most traditional algorithmic problems, efficient solutions often rely on a *combinatorial* understanding of the problem, for example, knowing how many regions of a certain type can be formed by an arrangement of n lines.