Challenges for Theoretical Computer Science: DRAFT

(Last Updated 28 June 2000)

This draft report was inspired in part by a Workshop on Challenges for Theoretical Computer Science, held in Portland on May 19, 2000. As three of the four organizers of that workshop (Christos Papadimitriou, Avi Wigderson, and Mihalis Yannakakis) were traveling for significant portions of the time between the workshop and the time the draft was due at NSF, this draft has been primarily put together by the fourth organizer (David Johnson), based on input from panel participants and others, but without much opportunity for consultation with his co-organizers.

Given the short time frame allowed for creating this draft, certain compromises have been made, which we hope readers will forgive. We have opted to include more material at the expense of time for adequate selectivity, proof-reading, and polishing. We have attempted to do the best job we could with the material we had, rather than devoting extensive efforts toward filling the gaps in the presentation. Many challenges had to be translated from sketchy viewgraph summaries, and there has not always been time to fill in all the missing details adequately. Moreover, in adding what details we did, we have not in all cases had time to verify that we have correctly captured the proposer's intent.

We hope readers will nevertheless find this draft to be a stimulating beginning, and we welcome their comments and suggestions, which should be sent to David Johnson (dsj@research.att.com).

Introduction: Theory as the Scientific Foundation of Computing

Connections with Other Sciences

- Biology
- Physics
- Astronomy
- Political Science and Sociology
- Economics

Connections with the Web

- Searching and Data Mining
- Electronic Commerce and other Web Applications
- Security and Privacy

Connections with the Rest of Computer Science

- Software Engineering
- Computer Aided Verification
- Networking
- Database Systems
- Operating Systems
Central Issues for Theoretical Computer Science

- Cryptography and Security
- Combinatorial Algorithms
- Computational Geometry
- Parallel and Distributed Computing
- Complexity Classes
- Lower Bounds
- Logic, Semantics, and Programming Methodology
- Learning Theory and Statistical Inference

Introduction: Theory as the Scientific Foundation of Computing

If science is the search for the fundamental principles that govern the world around us and explain the phenomena we see, then a case can be made that Theoretical Computer Science (TCS) is the "science" underlying the field of computing. The formal and mathematical nature of TCS is especially appropriate for a science of computing, given that computation itself is essentially a discrete logical process. At any rate, it is clear (as this report will illustrate) that theory is an essential part of almost everything we do today in the field of computing and especially in the growing connections of computing to other scientific disciplines and the burgeoning new field of electronic commerce.

In this report, we shall illustrate the central role of TCS in computing in terms of a broad-ranging list of opportunities and challenges facing theory and theorists as we begin the 21st Century, ones that permeate the world of computing and its applications and at the same time emphasize and illustrate the importance of the fundamental issues that are at the core of the field. Although the role of theory (and indeed computing itself) can on occasion be merely as a tool in support of other areas, just as often theory's true role is that of guiding light, providing the models needed for understanding and the innovations that inspire whole new directions of endeavor.

The Principles of TCS

What is it about TCS that makes this possible? There are several basic principles that characterize the way TCS works.

- **A Need for Modeling and Abstraction.**
  The key to understanding a computational process or problem is to abstract away the irrelevant or less-important details and obtain a model that captures the essence of the matter. This model can then be studied, reasoned about, and as necessary modified to help us understand the possibilities and limitations of the processes/problems being modeled. Moreover, it allows the techniques and analyses developed to be more general and thus remain adaptable as the underlying technologies or applications change.

- **A Drive Toward Unification and Linkage.**
  A major goal of TCS is to discover fundamental links between seemingly disparate areas of computing. Once problems are stripped to their essentials, similarities between them become clearer, allowing us
to transfer ideas from one setting to another. General models thus can provide (and have provided) the intellectual underpinnings, the vocabulary, and basic tools for many different areas of computing. For example, the basic "random access machine" computer model has served well as an abstract computer model for use in designing algorithms for over three decades, while real computer architectures have undergone major evolutions in detailed structure and speed. The model doesn't capture everything, but it still is often the best abstraction with which to start and the cleanest way to describe the key ideas in an algorithm. As another example, the theory of NP-completeness shows that a wide variety of problems arising from all areas of computer science, from artificial intelligence, to operations research, to compilation, to networking, to VLSI CAD, are all in an important sense just equivalent variants on the same basic (difficult) problem. Another example of unexpected linkages is the development of Probabilistically Checkable Proofs, where fundamental implications about approximation algorithms were made possible using techniques that emerged from cryptography, the theory of randomized algorithms, average-case analysis and program correctness testing.

- **A Curiosity about Possibilities and Fundamental Limitations.**
  Given a problem, how hard is it to solve? What kinds of solutions are feasible? Given a computational model, what are the inherent limitations on its computational power. Given a formalism, what are the limits on what it can express? These are the sorts of questions that TCS asks again and again in different domains, and they are key to our understanding the potentialities of computing. Moreover, they are in essence mathematical. No experiment can prove a negative, can for instance show that a given algorithm cannot be improved. Substantiating the notion of computational intractability is thus the province of theory. This is a question that impacts all of computer science. Theoretical Computer Science is responsible for building a body of mathematical techniques for addressing such questions. The canonical example is the P vs NP question. Although direct progress towards answering this question has been slow, the theory of lower bounds has been developing a rich set of techniques, many of which have found other applications. An example where more progress has been made is that of randomized vs. deterministic computation. While not yet proven, strong evidence now exists that randomness does not speed computation dramatically, i.e., that $P = BPP$. This evidence was built up over years of research combining ideas from cryptography, structural complexity, the theory of randomized algorithms, and circuit complexity.

- **A View toward the Future.**
  A natural corollary of an interest in the limitations and possibilities of models is an interest in finding ways to circumvent the limitations and exploit the possibilities. Moreover, since computing is essentially a man-made phenomenon (or at least seemed so until recent developments in biology and quantum mechanics started suggesting that computation may indeed be inherent in the way Nature operates), one need not restrict attention only to what is doable today. One can consider what will be possible with new architectures, faster and bigger machines, future applications. These are things that TCS can examine before they exist because the relevant analytical tools are primarily mathematical rather than experimental. TCS has been inventing the future in this way for all its life. For example, Church and Turing gave formalizations of computability well before actual general-purpose computers were built. The theory of parallel computation pre-existed actual large-scale parallel machines. The theory of public-key cryptography was well-developed before the World Wide Web made it an everyday necessity. Currently, theoretical exploration of quantum computation is both motivating and clarifying the requirements for the possible construction of quantum computers. A more general example of this world view is theory's concern for asymptotic analysis, e.g., looking at the way running time grows as a function of instance size. Today many of the algorithms and data structures that were deemed of only theoretic interest when they were first introduced are now in daily use as instances that were once too large to be considered are now everyday occurrences.

**Factoring: An Illustrative Example**
As a more detailed illustration of many of these ideas, and especially of the linkages that can be exposed by TCS, consider the question of whether one can factor large integers efficiently. Factoring is the most famous example of a conjectured one-way function, a function that is simple to compute but intractable to invert. One-way functions are mainly known for their relevance to cryptography, but in fact their role in our understanding of other fundamental issues is far greater. This is because the possibility of easily defined but intractable problems changes our viewpoint on basic mathematical and philosophical notions which have been studied for centuries. Examples of such notions are Computation, Proof, Entropy, Randomness, Learning, Knowledge, Game, and Encryption. Introducing one-way functions into models of the above concepts changes them dramatically, because of the distinction in introduces between knowing information that determines a value and being able to compute that value in any practical amount of time.

To see the broad scope of this distinction, consider some implications of the simple complexity statement "Factoring is Hard on Average" (i.e., "Multiplication is a one-way function"). None of these implications are obvious, but are rather summaries of many different but related lines of theoretical research. [References to be added.]

- Essentially every cryptographic task can be digitally implemented.
- Classical computation cannot efficiently simulate (quantum) nature.
- Natural proofs cannot be used in circuit lower bounds.
- The amount of computational entropy in a source can be increased deterministically.
- Randomness can be efficiently simulated by deterministic algorithms.
- Efficient learning of natural concepts is impossible.
- Every mathematical theorem can be proved in a way that gives the hearer no knowledge beyond its statement.
- Deterministic games can have no winning strategy for either player.

Thus, factoring's intractability is a question with far-reaching implications for many areas of computer science, as well as physics, mathematics, economics, and society at large. These implications show the unity of TCS as an intellectual discipline. It is hard to separate the ideas from one sub-field from those in another, when the same fundamental issues underlie them all.

**The Organization of this Report**

This report is based in part on presentations prepared for and made at a one-day Workshop on Challenges for Theoretical Computer Science, held in Portland on May 19, 2000, and co-sponsored by DIMACS, SIAM, and SIGACT with financial support from the NSF. The goal was to remind theoreticians and others of the wide-ranging importance of and opportunities for TCS. Rather than do this by summarizing our many past accomplishments and their impact, the plan was to use the dawning of the new millennium as an excuse to look into the future, constructing a list of challenges that illustrate the new (and continuing) issues with which theory is inextricably linked.

Special emphasis was placed on pointing out the connections between TCS and other fields, many of which might not have been imagined a decade ago. Thus the workshop was organized into four panel discussions. The first concerned the central issues with which TCS is concerned, ones that underlie a variety of different application areas and ones that deal with the fundamental nature of computation itself. The remaining three panels considered connections between TCS and other subdisciplines of computer science, connections to other sciences, and connections to the Web and electronic commerce. The workshop was a lively affair, and highlighted many important challenges which we have subsequently augmented with the help of additional contributors.

As might be expected, the challenges reflect a variety of concerns, from fundamental questions of broad
significance to highly-specific (but important) applied queries, from requests for models and analytic
techniques to pleas for help in making certain theoretical concepts or algorithms more "practical," from
highly-speculative new proposals to long-standing open problems. (What is interesting is that this sort of
range was present in each of the four panels.) For this draft, we have followed the organization into panels,
but have chosen a different order of presentation, starting with the "Connections" challenges and going from
outside-in, rather than starting with the central issues and proceeding from inside-out, as was done at the
workshop. There are several reasons for this.

First, for readers coming fresh to this material, the significance of the central issues presented in our final
section will be much clearer once one has seen the many ways in which those issues underlie the overall field
of computing, as illustrated in the "Connections" challenges. These central issues constitute the core of TCS,
and we want people to see why they are important. Second, one of the main goals of this report is to illustrate
the wide reach and relevance of theory, so it seems natural to highlight that first. Finally, readers in the TCS
community will already be familiar with many of the central challenges confronting TCS, but may be less
aware of the exciting new developments connected to other fields and the opportunities they present both for
applied and fundamental work. Thus it again makes sense to put those first.

We stress that the order of sections should not be taken to imply any value judgments about what research
topics are most important. We strongly believe in the scientific value of studying the central problems of TCS
for their own sake. The combination of a focus on the core problems together with an openness to the outside
world of industry and science has made TCS the dynamic and productive field it has become. The most
successful makers of connections (both in solving other disciplines' problems and in integrating them into the
core of TCS) were people who were trained in and who contributed to the core of TCS - in other words
people who could model, abstract, unify, make connections, and use the diverse techniques of our field. Only
continuing (curiosity-driven) research in the core areas of TCS will ensure the continuation of the wellspring
(of ideas and people) by which theory has been impacting the field of computing and the ever-broader reach
of its connections.

We also should point out that our intent here is not to list all the important challenges or cover all the areas
where theory has the potential for impact. Space and time constraints would make such a goal impossible.
However, we do hope to provide a representative sample of challenges, and to capture some of the most
stimulating ones. In what follows, we have tried to describe each challenge in sufficient detail that the issues
are exposed, but in terms that we hope will be meaningful to the non-expert. In several of the subtopics under
the "Central Issues" section, where there is much history lying behind the new challenges, we have also
included an opening preamble giving some of this historical perspective. For the final draft, we hope to
include references for each Challenge, so that interested readers can find out more. We also hope to fill in
many of the obvious coverage gaps still remaining in this preliminary draft, and welcome suggestions on how
to do this.

---

**Connections with Other Sciences**

**Biology**

[This section drafted based on contributions from Anne Condon, Dannie Durand, Dick Karp, and Alex
Schaffer, as well as the April 1999 Report on Challenges for Theory of Computing]

**Challenge: Genome Sequencing**

[Drafted 6/9, Revised 6/19]
By the time you read this, victory may have been declared for the Grand Challenge of determining the DNA sequence for the human genome, a combined victory for both the technology developed for gathering data about the genome and the algorithms invented to infer structural details from that data. However, this is just a first step. Can we now use what we have learned to develop much faster methods, perhaps bootstrapping off of the knowledge we have gained about known genomes and how they are structured? With such techniques we might be able to sequence genomes for large numbers of individuals, thus obtaining much more detailed knowledge of the variability of the species. We could also sequence a wide variety of different species, helping us to infer evolutionary relationships among organisms. There could also be important disease-control benefits if we were able to sequence a newly-discovered disease bacterium in a day.

**Challenge: Understanding Evolution**

[Drafted 6/9, Revised 6/19]

The comparison of DNA and protein sequences in different species is an increasingly important tool for understanding the evolutionary relationships among species. Such comparisons have all but replaced the classical comparisons based on visible features, as it is more reliable and more clearly linked to the mechanisms of mutation. Moreover, by measuring the difference between two genes, that presumably corresponds to the number of mutations since they diverged, one can get a rough notion of the amount of time since two genes diverged. However, despite better data, one is still left with the very difficult problem of inferring the relationships after the fact. These are typically depicted by phylogenetic trees that indicate how species branched off from ancestral species, but there is much controversy about how such trees should be constructed. This is further complicated by the fact that for most interesting optimization criteria, the problem of constructing the tree that `best' explains the data is an NP-hard problem and so can only be tackled by heuristics that only give near-optimal results (and can be abused: the now abandoned `Single Eve' hypothesis came about due to the misinterpretation of the results obtained by running a local-search heuristic). Challenges:

- Can theoretical computer scientists, who have developed many of the algorithms for constructing phylogenetic trees, help clarify the issues by developing better probabilistic models for the evolutionary process and metrics for comparing trees or quantifying the robustness of the information deducible from them?
- Can phylogenetic tree computations help us gain insight into such hypothesized evolutionary mechanisms as gene doubling, etc.?

**Challenge: Understanding Genetic Diseases**

[Drafted 6/9]

Genes typically have two or more common forms called alleles and are subject to further variation due to mutations. These variations are the cause of thousands of genetic diseases (or predispositions to diseases) such as diabetes, cancer, and arteriosclerosis. By understanding how and which variant forms of genes influence disease it will eventually be possible to develop personalized medical treatment based on detection of variant forms of genes in individual patients. But first, we need to determine which genes or combinations of genes are responsible.

In some cases the only available approach is to look at the genes of affected individuals, their relatives, and control populations. Here we are given the family tree (not necessarily a true tree, due to inbreeding) with the individuals labeled as to whether they have the disease, and information about the genes of some but not all of the people in the tree. The goal is to derive likely hypotheses about causality, which is essentially an optimization problem, one that gets increasingly complicated as the number of genes involved increases. Here better algorithms would help as would complexity theoretic results, that might help us identify those diseases that are less-likely to be deciphered with current technology, thus allowing us to invest limited research funds on more promising searches.
Challenge: Developing the Computational Model of the Cell
[Drafted 6/9, Revised 6/12, 6/19]

Many processes that go on in living cells can be viewed in computational terms. DNA strands can in a sense be viewed as the tapes of multi-headed Turing machines, from which the designs for proteins (the genes) are read and the proteins themselves then produced. The rate at which proteins are produced and activated is different in different cells and at different times, depending on factors such as the ambient environment of the cell and chemical signals from other cells, the mechanisms for which are becoming much better understood (see for instance the June 2000 issue of SCIENTIFIC AMERICAN). Thus the problem of understanding cell function becomes more and more that of understanding the underlying communications protocols, and communications protocols are one of the basic computational mechanisms with which theoretical computer science concerns itself. Insights to be gained from looking at the cell in this way may help us understand such fundamental questions of molecular biology as how genes and proteins act in concert to control cellular processes such as cell division, metabolism, respiration, and DNA repair.

Challenge: Understanding the DNA Programming Language
[Drafted 6/9, Revised 6/19]

If one views the DNA in our chromosomes as constituting a "program" for building us (a plausible analogy), then it is natural to want to understand the underlying programming language involved. The DNA sequence in the vicinity of a gene typically has several structural features. These include exons and introns, a promoter region where an enzyme called RNA polymerase must bind in order to perform transcription, and binding sites where proteins or complexes of proteins bind to the DNA to enable or inhibit transcription. Deterministic or stochastic properties of these sequences can be used identify genes and parse their sequences. Here are some specific questions:

- There can be more than one way to encode a specific protein. What rules, if any, govern the choice of encoding? Is something being optimized (presumably by evolution)? What measures for fault-tolerance and error-recovery (if any) are included?
- What are the common "bugs" that mutations can introduce into gene programs and their effects?
- What is the purpose (if any) of the large quantities of "junk DNA" that does not appear to code for any proteins?

Challenge: Protein Folding
[Drafted 6/9, Revised 6/22]

In one sense a protein is simply a sequence of amino acids. This is how proteins are encoded in our DNA and how they are initially constructed in our bodies. However, once this linear chain is first constructed, it automatically folds into a complicated shape (determined by the sequence in ways we do not yet fully understand), and it is the geometric properties of this shape that to a large extent determine how the protein functions.

The problem of predicting how a protein folds, given its amino acid sequence, is one of the premier problems of science. Once the 3-dimensional structure of a protein is known, it becomes possible to design drugs that inhibit or enhance a protein's activity by fitting into niches in the surface of the protein. It may also be possible to design new proteins with useful properties. But first we need algorithms both for determining the geometric structure from the sequence and, perhaps even more difficult, to determine sequences that give rise to desired structures (a problem similar in nature to the self-assembly problem listed below). We also need to understand how and which small changes in a gene's sequence can cause major changes in its 3-dimensional structure. Limited progress has been made, but the main challenges remain before us.

One particular theoretical challenge is that, in general, the problem of determining the minimum-energy
folded structure for a protein is now known to be NP-hard. This suggests various interesting possibilities:

- P = NP and Nature knows and exploits this (unlikely, but something with exciting consequences if we can figure out the algorithms Nature uses).
- Although P does not equal NP, Nature can solve NP-hard problems efficiently using the massively parallel computation inherent in the protein folding process (also unlikely, but something with exciting consequences for biomolecular computing if true).
- Evolution has given us proteins whose structure is sufficiently restricted so that the folding problem is easier to solve (this suggests trying to discover just what those restrictions are and why they work).
- Proteins don't necessarily fold to their minimum-energy configurations, but instead just go to a very attractive local optimum in the energy landscape (in which case we also have important design criteria left to discover).

**Challenge: Databases for Bioinformatics**

[Drafted 6/12]

The proliferation of biological data and the need for its systematic and flexible storage, retrieval, and manipulation is creating opportunities in the database field. Current genomic databases are heterogeneous, distributed, and semistructured or with schemas that are in flux, thus offering novel challenges in database design, including its more fundamental aspects. The study of gene expression, protein structure, and cell differentiation are introducing quantities of data far beyond the "mere" gigabyte of data represented by the human genome. There are new challenges for indexing of sequences and 3-dimensional structures, for lab workflow storage and process modeling, and for processing queries that involve specialized approximate pattern matching and complex geometric relations such as molecule docking. These problems could benefit much from the theory community's extensive experience with string matching, scheduling, query optimization, and computational geometry.

**Challenge: Exploiting DNA Arrays and Their Variants**

[Drafted 6/19]

A relatively new biological testing mechanism is the `DNA array," in which DNA fragments are arranged in a grid and mechanisms are provided so that if an external DNA fragment matches up with one of the fragments in the array, light is given off. This allows a large number of tests to be performed in parallel, with the genetic makeup of a collection of fragments can be interpreted by examining the pattern of light coming off the array. This basic testing scheme, and likely future variants on it, give rise to a variety of computational questions.

- How does one efficiently derive conclusions from the results of a text using an array?
- How should arrays be designed so as to maximize the accuracy of readings obtained from it, i.e., how should the various DNA fragments be distributed to limit false positives and false negatives?
- Given a collection of tests that is larger than can fit on a single array, how can one best partition the tests between arrays and order the use of the arrays so as to minimize the biological work?

**Challenge: Biomolecular Computation**

[Drafted 6/9, Revised 6/12, 6/19]

Biomolecular computing is a field that seeks to harness and expand the powerful toolbox of chemical and enzymatic processes for manipulating biomolecules, with the goal of exploiting these processes as a means performing computational tasks. It has grown out of the original "DNA computing" work of a theorist, Len Adleman, who provided a practical demonstration that DNA molecules could be harnessed to do parallel computation, in his case to solve a small instance of the NP-hard Hamiltonian Circuit problem. Since then it has expanded to include work on building nanomolecular structures from DNA molecules, and several other
computing paradigms such as DNA computations on a chip and other hybrids between biomolecular action and traditional silicon-based devices (see for instance the June 2000 issue of SCIENTIFIC AMERICAN).

The potentialities of biomolecular computing are not yet clear, but one promising area would be in the parallel simulation of various biochemical processes. From a theoretical point of view, we need to better understand the power and limitations of various proposed models, including "classical" parallel computation with non-interacting DNA strands, whiplash, self-assembly, systems of communicating cells, and more. In addition, designing the molecular components of a DNA computation raises new variants of old information theoretic questions. For instance, can one design "large" sets of minimally hybridizing DNA strands? How does one simulate errorless models using error-prone biomolecular operations? Such questions potentially involve an interplay of molecular chemistry, computational geometry, and combinatorial optimization.

**Challenge: Nanotechnology and Molecular Self-Assembly**

``Nanotechnology" refers to manufacturing of objects so small that they must essentially be assembled molecule by molecule. Much has been written in the popular press about the potentialities for nanotechnology, but we are still a long way from making it work in a general way. A key manufacturing technique is likely to be "self-assembly," where once the basic units are manufactured, they put themselves together automatically according to their structure and geometry (and the rules of chemistry and physics). Such self-assembly can be viewed as an algorithmic process, taking on aspects of computational geometry and parallel processing, as shown in the work of Berger and Shor, who provided a geometric explanation of the self-assembly of virus shells that goes on in nature and of Rothemund and Winfree, who showed how squares of any given size can construct themselves from a fixed set of base tiles.

The challenge is to develop a fuller understanding of the basic principles involved, an algorithmic toolkit of basic self-assembly subroutines, and perhaps a complexity theory of self-assembly that can help guide practitioners as to what is and is not possible.

**Challenge: Understanding the Brain**

A collections of biological processes that have been viewed in computational terms for decades concerns the actions of the brain. Until recently, much of the hypothesizing in this area has had no firm base. However, as we gain a better understanding of how neurons work, how they signal and respond to signals and how they are interconnected, we are reaching the point where we can accurately address the question of how the brain does its computing. As we refine our computational model for brain function, we can begin to ask just which algorithms the brain might use for such tasks as perception, learning, and data storage. These are questions that may call on many areas with theoretical computer science, from learning theory to distributed and parallel computing.

---

**Physics**

[This section drafted based on contributions from Daphne Koller, Peter Shor, Naftali Tishby, John Watrous, and Peter Winkler.]

**Challenge: Understanding Quantum Mechanics**

``Anyone who can contemplate quantum mechanics without getting dizzy, has not properly understood it" -
Niels Bohr.

Quantum mechanics has existed as a mathematical system for explaining physical processes for perhaps three quarters of a century. It is fair to say, however, that the recent results from theoretical computer science that attempt to exploit quantum mechanical principles for both computation and cryptography have been a great stimulus to our understanding the implications and meaning of several fundamental quantum mechanical phenomena, such as superposition of states, decoherence, and EPR pairs. By trying to exploit (or overcome) such phenomena, we learn more about them, their implications, and their limitations. Indeed, viewing quantum systems as inherently computational in nature may prove to be a fruitful way of looking at them, with the potential for new insights into fundamental physics. The growing number of successful collaborations between theoretical physicists and theoretical computer scientists is a promising development.

**Challenge: Make Quantum Computing a Reality**

[Drafted 6/12]

Until theoretical computer scientists such as Peter Shor became involved, there seemed to be insuperable fundamental obstacles confronting would-be builders of quantum computers, such as the difficulty of exact measurement and the principle of decoherence. With the development of quantum error correcting codes and fault-tolerant gates, we can now see our way around such obstacles. Attention has thus turned to experimental physicists and their attempts using various technologies to create ensembles of qubits and other components of quantum computers.

Nevertheless, if we are to attain the goal of practical quantum computation, we will need to continue working on the algorithmic side as well. We need to increase the efficiency and practicality of fault-tolerant quantum computing schemes. We need more accurate error models and improved methods for analysis of fault-tolerant systems in the presence of such errors. Moreover, depending on what physical technologies are used, specific challenges may arise, such as the difficulty of getting a quantum computer based on liquid NMR to the initial all-0 state given currently achievable values of polarization.

**Challenge: Solve New Problems with Quantum Computers**

[Drafted 6/12, Revised 6/19]

Currently there are two main classes of problems that can be solved more quickly on a quantum computer (if we can build one) than we know how to do on a classical computer: The number-theoretic problems of factoring and discrete logarithm computation, which can be sped up by a super-polynomial factor (i.e., more than any polynomial) by the algorithms of Shor, and searching problems that can be sped up from $O(n)$ to $O(\sqrt{n})$ by algorithms of Grover. Are there other, distinctly different types of problems for which quantum computers offer a potential speedup? Possible candidates:

- Simulation of specific physical systems
- Graph isomorphism (a hard problem that, like factoring, is not likely to be NP-hard)
- Some version of the short vector in a lattice problem
- Some classical NP-complete problem

More specifically, to support the experimentalists, find interesting tasks for small (e.g., 100 qubit) quantum computers to do.

**Challenge: Prove the Security of Quantum Key Exchange**

[Drafted 6/22]

In addition to its potential to assist computation, quantum mechanics also has the potential to enhance security. By exploiting the Heisenberg Uncertainty Principle, we may be able to obtain unbreakable security
for certain applications. One key cryptographic problem that has been addressed in this way (both theoretically and experimentally) is the problem of distributing keys (to be used for instance as one-time pads or as the private information used in more complicated systems). The first and best-known protocol for "quantum key exchange" is what is now called BB84, introduced by Bennett and Brassard in their paper "Quantum Cryptography: Public-key Distribution and Coin Tossing," Proc. IEEE Int. Conf. on Computers, Systems and Signal Processing, IEEE, 1984, 175-179. This protocol has been the subject of much study, both experimental and theoretical.

Much of the theoretical work was directed at constructing a proof that BB84 was secure, which was in a sense accomplished by Mayers in "Quantum Key Distribution and String Oblivious Transfer in Noisy Channels," Advances in Cryptology: Proc. Crypto '96, Springer-Verlag, New York, 1996, 343-357. The current work all assumes, however, that in the physical implementation one uses a single photon source. In practice, all that is currently available is a "weak coherent source," one that is a superposition of states including (i) a high-amplitude state including 0 photons, (ii) a smaller amplitude state containing one photon, (iii) a still-smaller amplitude state containing two photons, three photons, etc.

Challenge: Prove that BB84 is secure for weak coherent sources.

**Challenge: Develop Quantum Complexity Theory**
[Drafted 6/12, Revised 6/27]

Complexity-theoretic aspects of quantum computation are currently not well understood. For instance, while most well-studied "classical" complexity classes such as P, NP, BPP, and PSPACE can be characterized in many different ways in terms of seemingly unrelated notions, the same cannot yet be said of the main quantum complexity classes. In particular, we currently know of no way to characterize the class of problems that can be computed in polynomial time by quantum computers in terms of classical resources. How does this class (called BQP), compare to P, NP, the polynomial-time hierarchy, etc.? Can quantum computers solve NP-hard problems in polynomial time, or would unexpected consequences (such as P = NP) result? Finally, can we use facts learned from quantum complexity theory to learn new things about classical complexity theory? Discoveries in this vein could prove to be equally or even more beneficial to computer science than the physical realization of quantum computers.

**Challenge: Develop Quantum Information Theory**
[Drafted 6/12]

There has been much recent theoretical work on exploiting quantum phenomena for communications purposes, both in the case of secure communication (quantum cryptography) and simply more efficient communication. However, many questions of fundamental importance in quantum information theory remain open. For instance, to what extent are channel capacity, minimum entropy, and other quantities for quantum channels "additive"? What are the relationships among various measures of entanglement?

**Challenge: Bounded Rationality Physics**
[Drafted 6/12, Revised 6/19]

In economics, there is a concept called "bounded rationality" that comes into play as we realize that economic agents cannot in practice optimize their behavior if this involves solving computationally intractable problems. Does the same concept have implications for the physical universe? As seen in the Biology section of this report, the physical process of protein folding has to cope with the NP-hardness of the general problem it has to solve. Other highly-complex physical processes also may have to deal with such obstacles. What other physical systems are capable of encoding NP-hard (or worse) problems? Possibilities include general relativity and other classical field theories, turbulence and classical chaos theory, statistical mechanics of disordered systems, etc. If such hard instances exist in theory, is it possible that they could arise...
(or be constructed) in practice (i.e., in real systems)?

**Challenge: Computational Statistical Mechanics**  
[Drafted 6/19]

In recent years there has been an increasing number of collaborations between theoretical computer scientists and physicists interested in statistical mechanics. Microsoft Research has an entire department devoted to this sort of work. The synergy is due to the fact that both groups of researchers are interested in the same sorts of problems, but bring different sets of tools and ways of thought to them. The classical physics problem that serves as a motivation is the behavior of matter (from gases to crystal lattices) in the presence of heat, magnetism, or other external forces.

One fundamental quantity that physicists are interested in is the "partition function" for a given model, which is needed if probabilities are to be normalized. For one well-studied model, the Ising model for spin glasses, we now have almost completely characterized the complexity of computing the partition function. It has long been known to be polynomial-time computable for planar lattices, and now, due to results of Istrail (STOC 2000) it is known to be NP-hard for any non-planar lattice. One challenge is to extend this work to other important models. Also, where NP-hardness holds, we need to develop heuristics that give good estimates of the true value of the partition function.

Another fundamental quantity associated with a model is "mixing time" in the presence of a heat bath. In physics this corresponds to the time it takes for a mixture or solution to reach equilibrium. In mathematical terms, it is the time for a Markov chain, started in a specific state, to come sufficiently close to its equilibrium distribution. There has been much recent work by theoretical computer scientists on the topic of "rapid mixing" (processes that get close to their equilibrium distribution quickly) because of its use in the design of randomized algorithms. Can these developments be exploited to help us better understand physical systems?

Additional problems on the borderline between statistical mechanics and theoretical computer science include getting good estimates on various fundamental constants such as

- The limiting multiple of $n$ in the exponent of the number of self-avoiding random walks (from 0) of length $n$ on the plane grid (cf. Randall and Sinclair).
- The threshold probability for various forms of site and bond percolation (cf. Bollobas and Stacey).
- The limit of $n^{-1/3}$ times the length of the longest chain in the coordinate partial order determined by random points in 3-space.

**Challenge: Understanding Phase Transitions in Computational Complexity**  
[Drafted 6/12, Revised 6/19]

In recent years researchers have uncovered phenomena in combinatorial optimization problems that are analogous to physical phase transitions, or more specifically to the models of such transitions studied by theoretical physicists using statistical mechanics. For instance, if one randomly generates instances of the NP-hard 3-Satisfiability problem (3SAT) in an appropriate way, they make a sharp transition from easy to hard as one increases the ratio of clauses to variables. Similar phenomena occur for probability spaces related to other combinatorial optimization problems.

What are the common principles underlying such behavior? What mechanisms cause it to occur? Are there any practical situations in which it occurs? Do the phase boundaries always correspond to regions where if there are any solutions there is likely to be at most one (as in 3SAT)? Can we exploit what we are learning about phase transition to develop a general complexity theory of hard and easy instances? Can we exploit it to design better approximation algorithms?
Astronomy

[This section drafted based on contributions from Richard Anderson.]

**Challenge: Discovering Astronomical Structure**  
[Drafted 6/12]

Sky surveys are providing us with large quantities of astronomical data about the universe in 3-dimensional form (even 4-dimensional form as we gain more information about time and velocity). Can techniques from computational geometry help us identify structure in such data sets? With billions of stars to consider, this is a domain where our past concerns for "asymptotically efficient" algorithms may be especially relevant.

**Challenge: Astronomical Simulation**  
[Drafted 6/12]

One of the great successes for algorithms in scientific computation has been in solving the gravitational N-body problem, where the Barnes-Hut and Greengard-Rohklin algorithms have allowed us to speed up computations from the naive $O(N^2)$ to $O(N^{4/3})$, making possible simulations of astronomical systems involving millions of stars. However, even as computer memories become cheaper and multiprocessor machines become larger and faster, our desire for larger and more sophisticated astronomical simulations outstrips them and new algorithms and data structures appear to be required, ones that can be tuned to exploit the computer architecture that is to be used for the simulation. In addition, checkpointing strategies and data structures need to be devised that can help us to recover from system failure with little waste of time, despite the massive amount of information being handled. Our goal is a physical model that explains the observed universe, but the amount of data we have that needs explaining continues to grow rapidly as the results of sky surveys etc. Astronomers also want to move beyond simulations that consider only gravitational effects generated by the stars to include the results of collisions and near-collisions between stars as well as the hydrodynamics effects caused by the presence of interstellar gas.

Political Science and Sociology

[This section drafted based on contributions from Aleksandar Pekec and Fred Roberts.]

**Challenge: Designing Political Districts**  
[Drafted 6/19]

There are many different criteria involved in finding a "fair" political districting. Are the districts approximately equal in size? Homogeneous? Geographically coherent? Do they keep natural regions together?

Finding solutions that even approximate optimality on a few of these criteria turns out to be a complex computational problem. Recent developments in large scale discrete optimization help both theoretically and algorithmically and some computational experiments indicate that there are promising heuristics for solving these problems. However, the problems are only now coming to be precisely defined to the point where an approach by methods of TCS would be useful.

**Challenge: Preventing Strategic Voting**  
[Drafted 6/27]

It has long been known that not voting for what you really believe in can often bring about a result that is
closer to what you want than "sincere" voting. There is a long literature in political science having to do with ways to make an election easy to manipulate and other ways to protect against manipulation. Some ways to manipulate an election include certifying additional voters who might agree with your position, decertifying voters who might disagree with your position, certifying additional candidates who might take away votes from candidates you don't like, decertifying candidates who might take away votes from candidates you like. There are some results that say that for some voting rules and some manipulation procedures, it is computationally intractable to manipulate an election. For others it is computationally tractable. More needs to be done along these lines. In general, the challenge is to determine the complexity of various methods of "strategic voting" and design voting rules to protect against them.

**Challenge: Computing Power Indices**  
[Drafted 6/27]

Power indices have been used in political science and economics in a wide variety of applications. For instance, they have been used to measure power and determine equity ("one man one vote") under modified rules of procedure in the Electoral College, the Australian government, the United Nations, various county boards of supervisors, and professional society elections. They are also used to allocate costs to different users in shared projects. Examples of such applications include allocating runway fees to different users of an airport, highway fees to different size trucks, costs to different universities sharing library facilities, fair allocation of telephone calling charges, charges to different accounts for projects that cut across accounts, etc. Among the most popular indices for measuring power are the Shapley-Shubik power index, the Banzhaf power index, the Coleman power index, etc. These indices sometimes need to be calculated for huge voting games, yet in most cases it is computationally intractable to compute them. Approximation algorithms and and other effective heuristics are needed.

**Challenge: Deducing Social Influence**  
[Drafted 6/27]

Models of how opinions form and change are very important in sociology and psychology and also have important marketing applications, with particular emphasis nowadays on communication on the Internet and how the opinions get formed and business/products get ranked there. Models of social influence often involve computational components, being built around Markov chains, dynamic processes on graphs, etc. Among the issues are: Does an individual evolve over time to a stable opinion? Do all the individuals in a group evolve to the same stable final opinion? Which individuals have the real power and influence in the sense that a group's final opinion is fundamentally determined by the initial opinions of these individuals? We only know the answers to these questions for very specific processes and social network topologies. As yet we have very few algorithms for identifying which social networks and influence processes yield stability and which individuals have the real power and influence.

---

**Economics**

[This section drafted based on contributions from Joan Feigenbaum, Ming Kao, and Scott Shenker.]

**Challenge: Continue Exploring the Impact of Bounded Rationality**  
[Drafted 6/13, Revised 6/20]

In classical economics and game theory, participants are assumed to have unbounded computational power, and thus can be assumed to use optimal strategies, so long as such strategies exist. In the real world, participants do not have this luxury. How does this change things? One direction of research is to assume that participants have limited computational power or information, a concept that is called "bounded rationality"

To date most of the work in this area has considered such restrictions as bounding memory, or requiring computations to be done by finite automata. Less-studied, and at least as interesting, would be considering the restriction to polynomial-time computations (under the assumption that P does not equal NP). In general however, there are many important questions here to be explored.

**Challenge: Develop a Theory of Algorithmic Mechanism Design**

[Drafted 6/13, Revised 6/22]

Game theory offers an interesting contrast to the models of distributed computing traditionally favored by computer scientists. In computer science, we often strive to design protocols that work in the presence of actively malicious agents, both to protect ourselves against arbitrary system failures and because we do not assume that we know our adversaries' motivations or that they will respond to incentives in predictable ways. In game theory, the agents don't want to make our life difficult; rather they are "selfish maximizers" who respond predictably to succinctly statable and quantifiable incentives. Protocols may thus use incentives (such as payments or price reductions) to manipulate the behavior of the participants toward the desired end. This idea is captured in the concept of a mechanism. A mechanism consists of four things:

1. For each agent a *family of strategies* (in the game-theoretic sense)
2. An *output function* that maps a tuple of strategies to an outcome
3. A *payment function* that maps a tuple of strategies to a tuple of payments (one to each agent)
4. For each agent, a *welfare function* that maps an (outcome,payment) pair to a real number

The goal of a mechanism design problem is to construct a mechanism that will produce a desired output assuming agents act so as to maximize their individual welfare. For instance, if the output function is real-valued, one might want to produce as large (or as small) a value as possible for the output function. Examples include load balancing in a time-shared multiprocessor system, minimizing bandwidth usage in a packet routing network, and maximizing revenue in a multicast scheme. Indeed, many distributed commercial tasks that we might wish to accomplish over the Internet give rise to mechanism design problems.

Desirable properties of mechanisms include possession of *dominant strategies* (each agent has one strategy that optimizes its welfare no matter what strategies are used by the other agents), being *strategyproof* (in mechanisms where a strategy consists solely in revealing one's utility function, truth-telling is a dominant strategy), and efficiency, both in the computational complexity of the various functions involved and, in the case where the agents are connected by a network, the amount of communication needed to compute those functions.

Economists' work on mechanism design has, for the most part, ignored computational issues. We need to develop a complexity theory of mechanism design, especially distributed mechanisms: find the right computational model(s), the right resource(s) to measure, the right notion(s) of reduction, canonical hard problem(s), efficient algorithms for problems to which hardness results do not apply, etc. In addition, can we design mechanisms that assure various levels of privacy to the participants? Can privacy issues be "layered" on top of game-theoretic and computational efficiency issues, or must we take into account the interactions between the two in our initial mechanism designs?

References (to preliminary work along these lines):

- Feigenbaum, Papadimitriou, and Shenker, "Sharing the Cost of Multicast Transmissions," STOC 2000,
Challenge: Algorithms for Computational Finance

[Drafted 6/13]

In recent years, the financial industry has spent many computer cycles (and absorbed many theoretical computer scientists) in attempting to profit from hidden implications of financial data (for instance, by arbitrage, derivative pricing, etc.). Public research in this area has been held back by the desire for (and value of) secrecy in the business. However, this is clearly an area where algorithmic advances can have big payoffs.

To give one example, consider the problem of risk management and derivative pricing for very large portfolios, a problem proposed by Peter Carr at the Bank of America. The statistical behavior of a single stock is considered simple and it is relatively easy to price derivatives written on a single stock. However, even with just two stocks certain questions become \#P-hard, and if a portfolio has a large number of stocks things get even more complicated. For instance, pricing a call option on N stocks would involve N binomial trees, each accounting for one dimension. How are the time complexity and accuracy affected by dimension? Can heuristics give useful results?

For some initial work along these lines, see Kao, Nolte, and Tate, "The Risk Profile Problem for Stock Portfolio Optimization," STOC 2000, 228-234.

A second issue in computational finance is "overmodeling." This is a serious problem and can occur in two basic forms. On the one hand, a model can be too simplistic to be relevant to the current market condition. For example, the standard approach of modeling stock prices by a log-normal distribution is useful only for deriving closed-form formulas of option prices. On the other hand, a model can be too dependent on the current market condition to adapt to rapidly changing market trends. The disastrous near-collapse of the Long Term Capital Management in 1998 was reported to be a result of such inflexible models. Overmodeling is typically caused by the analytical difficulty of the mathematical finance question involved. A fundamental challenge for theoretical computer science would be to invent efficient algorithms to enable accurate and robust modeling. We anticipate that such algorithmic advances will also lead us to formulate novel finance problems which significantly depart from traditional paradigms of finance research. (Contributed by Jason Tigg formerly at Greenwich Capital Markets and Pete Kyle formerly at Morgan Stanley.)

Connections with the Web

[This section drafted based on contributions from Joan Feigenbaum, Ming Kao, David Karger, Jon Kleinberg, Alon Levy, Michael Luby, Moni Naor, Sridhar Rajagopalan, Prabhakar Raghavan, Scott Shenker, and Jeff Ullman.]

Searching and Data Mining

Challenge: Understanding and Exploiting the Graph Structure of the Web

[Drafted 6/19, Revised 6/21]

The Web now constitutes perhaps the biggest (and most important) directed graph we have yet seen in practice. Here web pages correspond to vertices, and a link embedded in page \textbf{A} and pointing to page \textbf{B} corresponds to an arc from \textbf{A} to \textbf{B}. The obvious challenges are to understand, model, and exploit this structure. Examples of modeling work in this area include
• Broder, Kumar, Maghoul, Raghavan, Rajagopalan, Stata, Tomkins, and Wiener. “Graph Structure in the Web,” *WWW*9, 2000.

In addition, perhaps the most influential paper on exploiting this graph-theoretic structure (it was the most-cited paper at the 2000 WWW web conference) is Kleinberg's 1998 SODA paper, "Authoritative Sources in a Hyperlinked Environment," that introduced the concepts of "hubs" and "authorities" and showed how useful rankings of the responses to web queries could be obtained by computing eigenvectors of relevant subgraphs of the Web. The popular Google web search engine uses a similar (but simpler) approach to ranking responses.

Nevertheless, we have just begun to scratch the surface in this area. What are the other meaningful (and useful) graph-theoretic properties of the Web, and how (algorithmically) can we identify them quickly in the dynamically changing environment that today's Web represents? (For a more specific challenge along these lines, see below.)

A related question: The proliferation of distributed content-searching and replication services such as Napster and Gnutella introduces a "parallel web" of peer-to-peer communication among internet hosts. What are the characteristics of this graph, how does it evolve, and how does it segment itself into components?

**Challenge: Finding Frequently Occurring Subgraphs of the Web Graph**

[Drafted 6/26]

Collections of hubs and authorities on the Web often form the "core" of a community of pages concerned with a common topic. In graph-theoretic terms, this can be viewed as follows: A dense directed bipartite subgraph of the Web is strong evidence that individual creators of pages are creating links to a common set of prominent resources. More generally, one finds that certain types of frequently occurring subgraphs of the Web graph have a natural organizational or sociological meaning in the context of the Web as a whole.

Kumar et al. (in Kumar, Raghavan, Rajagopalan, and Tomkins, “Trawling the Web for Emerging Cyber-Communities,” *8th World Wide Web Conference*, 1999) have developed heuristics to enumerate complete bipartite subgraphs from a large crawl of the Web graph; using these heuristics, they uncovered more than 100,000 "cyber-communities," many not yet known to directories such as Yahoo!.

A general challenge is to identify other types of frequently occurring subgraphs, understand their significance in the overall structure of the Web, and use such insights to provide improved access to particular kinds of information. From an algorithmic point of view, suppose we are given a large directed graph G (such as the Web), and a "template" subgraph H; we would like to efficiently find all (or many) isomorphic copies of H that occur in G. For which graphs H is such a problem feasible? It is reasonable to consider the case in which H has fixed, constant size; but the enumeration should run in time near-linear in the size of the underlying graph G, and ideally use a bounded number of passes through some reasonable representation of G. Even for subgraphs H for which the problem is provably intractable, good heuristics (as in the work of Kumar et al.) are of great interest.

More generally, one would like to find subgraphs that are "approximately isomorphic" to H; and one would like to be able to implicitly specify a family of template subgraphs (e.g. all bipartite subgraphs of at least a certain edge density, perhaps with size constraints) and find subgraphs of G isomorphic to any member of the family.
**Challenge: The Web of Queries**

[Drafted 6/19]

Today there are various services that will keep you informed of new developments in areas of interest to you by keeping track of changes in relevant web pages. As such services grow, one can envision the growth of an augmented version of the graph associated with the Web. In this new graph the vertices are not only web pages, but also queries that depend on webpages (and possibly on the results of other queries). There is an arc from a query to a webpage or another query if the result of the first query depends on the contents of the webpage or the result of the second query. Should this "Web of Queries" develop, we will need to confront such questions and issues as

- Detecting changes in queries and answers, as well as implicit dependencies among them.
- Efficient distributed processing of large numbers of queries in response to changes in webpages.
- Optimizing the tradeoff between precomputation and the running time required when the answer to a query is actually requested.
- Maintaining consistency between answers as the contents of the Web change dynamically.
- Extracting useful information from the graph structure of the query web.

**Challenge: Extracting Associations and Other Structure from Web Pages**

[Drafted 6/19, Revised 6/26]

Given that web sites do not typically obey well-understood design principles, the problem of extracting information from many pages in order to build databases that summarize the contents of many pages offers is formidable. We need to develop tools for describing the structure of Web pages and other information sources and efficiently integrating them into a relational database or other regular structure. (Complicating the problem: Today many sources have "input restrictions." You must fill in values for certain fields/attributes on a web form or the source will not answer your query.) Some interesting approaches include the way Sergey Brin mined the Web for book-author facts, in

http://www-db.stanford.edu/~sergey/extract.ps

and the technology of Levy, Mendelzon, Sagiv, and Srivastava ("answering queries using views"), Duschka, and others for using Datalog to build integrated products. These references, others, and an overview of the subject can be found through

http://www-db.stanford.edu/~ullman/cs345-notes.html

**Challenge: Answering Search Queries More Quickly**

[Drafted 6/26]

The typical web search involves a query that averages 2.3 words. A basic goal of a web search is to find documents that contain many copies of all (or as many as possible) of the query words. Sophisticated search engines may apply additional criteria on top of the search, but term occurrence always plays a crucial role. Search engines solve millions of these queries per second. A natural algorithmic challenge presents itself: to make this term-matching calculations fast.

Two simple approaches present themselves. On the one hand, when the query arrives, we can check each document to see how many occurrences of the terms there are. This takes time linear in the corpus size (number of documents). Alternatively, we can preprocess the corpus, computing, for each pair or triple of terms, which documents have the largest occurrence of those terms. This allows us to answer queries in constant time. However, it requires that we use space quadratic or cubic in the number of possible terms to store the results, which is far too much in practice.
So the challenge is to develop a scheme that trades off between these two points, exploiting the large-but-not-infinite storage space typically available to give better than linear-time responses. A common technique in practice is the inverted index: one lists, for every single term, the documents that contains that term; then, to answer a query, we need only take the lists for the terms in the query and intersect them. This reduces the query time to be proportional to the number of documents containing the query terms. But this can still be quite poor if the query terms are all common. Is there a more sophisticated scheme that can provide guaranteed speed regardless of what the query terms are?

Note that this problem is related to the computational question of searching for nearest neighbors on high dimension (the dimension being the number of distinct words in the corpus), although here we have the (non-standard) restriction that each query vector contains only 2 or three non-zero entries.

**Challenge: Next Generation Searching**

[Drafted 6/20]

In the future, there will be an increasing demand for search engines that are increasingly specialized and corpus-specific, for instance search engines specialized to legal case histories, academic literature, patents, recipes, etc. This is different from search engines that simply look for a certain type of file, such as MP3 files of downloadable music. Here we are interested in search engines that exploit the organization and specialized semantics of the data.

What are the common principles that apply to such specialized search engines and the models of data they exploit? What sort of algorithmic toolkit is needed? How can data be organized to facilitate such searching?

**Challenge: Determining the Geography of the Web**

[Drafted 6/21, Revised 6/27]

Web addresses are typically of two forms. First, there is the URL (Universal Resource Locator) such as sigact.acm.org that provides a linguistically meaningful address (once you have become accustomed to the conventions). Corresponding to this is the IP address, currently a 32-bit string that we typically represent as a sequence of four numbers between 0 and 255, separated by periods, as for instance 135.205.36.19. It is the second sort of address that is used by Internet routers, and a complicated distributed system of Domain Name Servers (DNS) is used by the Internet for looking up the IP address corresponding to a given URL. There is, however, a third address that is of interest: the network location of the web site (not necessarily where it is geographically, but where it is in the graph whose vertices are computers and other computational devices, and whose links are communications lines). The actual geographic location corresponding to an IP address constitutes a fourth address (and one that may not be constant, in the case of mobile users).

This network location is implicitly encoded in the IP address, which originally at least was supposed to present a hierarchical description of the networks or domains that contain the location. Today, however, a correct interpretation of the encoding may involve information that is distributed in routers all over the world and cannot be directly queried. Locations with almost identical IP addresses can be very far apart, due to changes in the network and other idiosyncrasies. When a router receives a packet whose destination is not within that router's own backbone network, it must rely on information from neighboring networks that advertise which ranges of IP addresses they know about or think they know how to forward to. (This information is constantly changing and updates are exchanged between neighboring backbone networks using the complicated BGP protocol.) The packet may then be forwarded on through a whole sequence of networks, with each network learning only about the next one in the path. Additional information will presumably be necessary if we are to infer geographical locations.

There is much potential value in knowing more precise information about the network and geographic location of an IP address. In particular, content distribution systems, such as the one Akamai provides, would
like to be able to direct a web site visitor to a cache that is close to the user (in the network sense), so as to 
eliminate or reduce the delays in content delivery that are caused when the content has to traverse many 
network links. Knowing geographic locations can be of value in marketing.

Challenge: Design algorithms that efficiently construct and update the current physical map of the Internet 
(the graph structure in which nodes are IP addresses and edges are communications links), using the 
protocols and mechanisms currently available for obtaining this information, together with whatever 
additional informations are available to help pin down geography. (Note that Akamai recently announced an 
information service (``EdgeScape'') based on a partial solution to this problem.)

**Electronic Commerce and other Web Applications**

**Challenge: Real-World Multiparty Function Computation**

[Drafted 6/19, Revised 6/22]

There has been much theoretical work over the past 15 years on secure multi-party function computation, in 
which a group of agents, each with its own input, jointly compute a function of all the inputs while revealing 
nothing to any participant besides the value of the function and any information that can be derived from that 
and the agent's own input value. Much of this work has been generic; the protocols work for any function and 
assume that any pair of agents can communicate directly with each other. Protocol design goals include 
minimizing the total amount of communication and the number of rounds of communication and maximizing 
the number of ``cheaters'' that can be thwarted by the honest participants.

Challenge: Develop (non-generic) efficient, secure, multiparty protocols for specific functions of interest.

Some examples that have already been studied include elections, where the function value is the identity of 
the winner of the election, and auctions, where two functions are computed: the winner of the auction and the 
amount the winner must pay. In practical protocol designs, there will often be constraints on the 
communication paths between parties, i.e., it may not be the case that A and B can communicate directly 
without having to go through one or more third parties.

References (to two recent protocols of this form):

on E-Commerce*, 1999

**Challenge: Designing Secure Electronic Trading Systems**

[Drafted 6/19]

Although many people are today actively trading stocks over the Internet, in doing so they must essentially 
rely on the trustworthiness and security of the Website through which they trade. Shyum Sunder at the Yale 
School of Management asks if one can design Internet-based trading systems or protocols that provably have 
the following properties.

- Far-away traders are not disadvantaged by communications delays (fairness).
- Traders can verify that their orders were executed according to their instructions (anti-manipulation).
- Traders can detect whether someone's trading program is winning excessively (market efficiency).

**Challenge: Develop a Theory of Web-Site Design**

[Drafted 6/26]
Relational databases are a huge success of theory, namely the relational model developed by Codd. The formal underpinnings of the relational model have enabled the design of systems with well-understood query languages and sophisticated techniques for query optimization. An even greater opportunity lies in the design of web sites, because a web site is also a database, except for a few differences. First, the data is not as well structured as in the relational world (often referred to as semi-structured data). Second, users interact with web sites through a combination of querying and browsing a hypertext structure, rather than only by querying. To date, we have no theory of web-site design, and tools for constructing and maintaining web sites provide only ad-hoc solutions. We need to:

- Develop a theory of the data model underlying web sites. The theory should encompass both the underlying content of the site and the navigational structure imposed over this content.
- Develop a theory underlying the optimization of web-site performance. Part of the problem is to automatically select which data needs to be materialized and cached in a web site based on analysis of the web-site structure and user browsing patterns.
- Develop methods for specifying and verifying integrity constraints on web sites.


**Challenge: Designing Adaptive Web Sites**  
[Drafted 6/26]

The hypertext structure of a web site is relatively static, and is designed for accommodate all possible users. However, after observing usage patterns of the site, it may be desirable to restructure a site to be more effective. Furthermore, we may want to automatically restructure the site for certain classes of users, or even for individuals. The challenge is to develop methods for automatically restructuring web sites for different users over time. The challenge involves two parts:

1. Detecting patterns of usage the suggest interesting restructurings, and
2. Automatically restructuring the site in a consistent manner.


---

**Security and Privacy**

**Challenge: Preventing Denial of Service Attacks and Related Problems**  
[Drafted 6/19]

A "denial of service attack" on the Web consists of flooding a site (or a subnetwork) with packets whose sole purpose is to overload the local system, thus hampering (or preventing) access by legitimate users/customers. Can router protocols be developed to prevent or disarm such attacks before substantial performance degradation can occur? Note that the problem is especially complicated because of the possibility of *distributed* attacks, where the malicious packets come from corrupted machines at many different locations.

A related question has to do with a fundamental weakness in the underlying IP protocols used by the Internet that can potentially be exploited by greedy (as opposed to simply malicious) users. The standard TCP protocol requires that all users reduce their sending rate when the links they are sharing become congested. However, there is nothing to prevent users from modifying their own copy of the TCP protocol so that it does
not reduce its rate as much as required (or at all). In this way, unscrupulous users could obtain more than their fair share of the bandwidth. Can router protocols be developed to force users to "play fair"?

**Challenge: Designing "Security Services"
[Drafted 6/22]**

Identify basic "security services" that should be available on the Internet and explore how to combine such services into secure protocols. Servers would provide a specified interaction and come with specific but limited guarantees of security. Such guarantees could be enforced through cryptographic means, reputation, or an escrow licensing deposit that would be confiscated if the guarantee were not met. Examples of such services include time-stamping (e.g., using the protocol from Haber and Stornetta "How to Time-Stamp a Digital Document," *Journal of Cryptology*, 3 (1991), 99-111), public-key certification, re-mailers, bulletin boards that guarantee non-erasure of posted messages, and certified random number generators (i.e., a provider of authenticated, randomly chosen numbers). The purpose of such intermediaries would be for efficiency (replacing communication and interaction between participants), availability (many servers for each task would exist, so protocols could continue despite any one failure), and simplicity (general tasks might have simpler protocols if such services were available).

**Challenge: Privacy and Access Control in Information Retrieval Systems
[Drafted 6/22, Revised 6/23]**

Modern IR systems manage and index very large (hundreds of millions) hyperlinked document corpora. The theory community has been instrumental in developing new algorithmic methods which qualitatively improve the search and mining results generated by these systems by exploiting the abovementioned graph-theoretic view of the Web. Graph algorithms are used to deduce which vertices of the Web graph are likely to contain useful or relevant information and which are not. While classical IR methods evaluate each document on its own, the new methods also consider the graph theoretic context within which the document appears, thus leveraging the collective wisdom of the entire web or corpus.

The assumption in this work is that the entire corpus is accessible to everyone using the system. Real corpora, though, often contain documents which are only accessible by some of the people using the system. The challenge is to find ways of dealing with privacy and access control requirements without sacrificing the quality and performance of modern searching and data mining methods.

**Challenge: Semantic Zero-Knowledge Proofs
[Drafted 6/20]**

Web searchers are typically looking for information "about X" where aboutness is a rather fuzzy notion. Humans can verify that given documents satisfy their information needs, and cooperative information retrieval systems can do a pretty good job of selecting documents that satisfy a specified information need. But suppose the given document costs money: how do you convince humans to pay for information without first showing it to them and eliminating the incentive? Zero knowledge proofs suggest a means, but they work only when the language of discourse is boolean logic. They might be used to prove that a document contains certain words, but this proves nothing about whether the document even makes sense in English. We need something fuzzier.

Develop a protocol that lets a server with a given document convince a client that the document is "about" topic X, without revealing the document to the client.

**Challenge: Public-Key Watermarking
[Drafted 6/21, Revised 6/22]**
In the Web context, “watermarking” is a process by which certain information about the owner or originator of a file (text, picture, sound, video) is embedded in the data in such a way that it can be used to trace back the source of any copy later discovered. For such a scheme to work, it must be the case that any computationally feasible attempt to render the watermark inoperative will also corrupt the original file sufficiently to render it unusable or at least significantly less valuable. Current watermarking schemes are “private-key” schemes: The owner of a document needs a secret key to run the watermark-insertion procedure, and the same key must be input to the watermark-detection procedure. For E-commerce applications, it would be desirable to have “public-key” watermarking schemes, in which there are pairs of keys, one used for marking (and kept private by the owner) and the other used for detection (and publicly distributed so that customers and middle-men can verify documents' provenance without having possession of the owner's private key).

Challenge: Develop a public-key watermarking scheme, or prove that none can exist.


**Challenge: “Good Enough” Intellectual Property Management**

[Drafted 6/19, Revised 6/27]

Critical to the development of electronic commerce is the management of digital Intellectual Property (IP). Technology has challenged the status quo of IP-management in many ways. Widespread use of personal computers and Internet communication creates vast opportunities for producers, distributors, and consumers of digital IP of all forms, but it also threatens to make copying and modification of that IP completely uncontrollable. Oft-repeated, accepted wisdom dictates that it is, of course, impossible to prevent copying or modification of digital material that is delivered to ordinary PCs, because a determined adversary can always "capture the bits at the rendering stage," but that, of course, it's not necessary to protect the material perfectly -- a protection scheme that is "good enough for the business model" will do. Develop the tools to make rigorous statements about "good enough" IP-management. Two aspects of real-world IP-management that should be part of our formal models but currently aren't are the dollar value of the digital assets under consideration and the expected commercial lifetime of those assets.

In thinking about how to formulate "good enough protection," it might help to keep the example of credit cards in mind. The credit-card system fails all of the formal definitions of "security" that the theory community uses. Yet the system is a big success: Card issuers make hefty profits and customers love credit cards and use them all the time. Changing the card system to make it fit our current definitions of security would probably be a bad idea, but coming up with a new formalism in which we could state that the current system is "good enough" might be a useful exercise.


**Challenge: Exploit Human Ability Considerations in Security Design**

[Drafted 6/27]

It is often said that the human being is the weakest link in the security of a system. The challenge is to incorporate assumptions on the ability and behavior of people into the design of the system in such a way that experiments and mathematical proofs can rigorously assure the security of the system.

An example of an area where such an investigation has been carried to a large extent is passwords, where protocols have been designed based on the assumption that the user has selected a password of not too large
entropy, yet the execution of the protocol should not make the adversary's probability of success larger than the probability of guessing the password.

A different direction in which human abilities can be exploited is to make sure that humans are the ones requesting a resource. Applications might include discouraging SPAM email or visits by meta-search engines trawling the Web. The solution is to give a puzzle that humans can easily solve, but that computers will have trouble with. Such ideas are already used in an ad hoc manner on the Web. The challenge here is to develop an automated Turing Test, which an adversarial program is unlikely to pass, even if it knows the testing program (but not the random bits it may be using).

**Challenge: Theory and Applications of Moderately Hard Functions**

[Drafted 6/20, Revised 6/27]

The theory of hard functions (e.g. one-way) has been a great success story and we have developed a good understanding of which tasks require what assumptions. In contrast, the theory of moderately hard functions has been neglected. This theory has many applications - combating junk mail (see a different approach in the previous challenge), preventing denial of service attacks, running auctions, and signing contracts (using a timed-commitment that can be opened within a certain amount of time) to name a few. There are many interesting research questions that are largely unexplored in this area. Some of the questions mirror the hard functions world and some are unique to this area. Relevant questions include: Can such hardness be amplified by asking for computations to be iterated? Can we use results about P-completeness to show immunity from parallel attacks?

---

**Connections with the Rest of Computer Science**

[This section drafted based on contributions from Ed Clarke, David Harel, Anna Karlin, Gerard Holzmann, Daphne Koller, Butler Lampson, Jeff Ullman, and Uzi Vishkin.]

---

**Software Engineering**

**Challenge: Software Specification**

[Drafted 6/21]

One would normally think that the true test of a software system is how well it meets the requirements set out for it. Unfortunately, in a typical software design process for a large system, requirements are initially quite ill-specified, having been constructed by teams of non- (or ex-) programmers, who produce `Requirements` documents that are long, written in informal natural language, and full of redundancies and inconsistencies and hence poorly adapted to providing a standard against which the resulting software can be measured.

Precise formal methods do exist for specifying desired behavior, such as message sequence charts (MSC's), but these do not have sufficient expressive power to completely specify what we might want from a system (basically, they just specify the types of behaviors we would like to be possible). Mechanisms for adding expressivity, such as live sequence charts have only recently been introduced, but it is not yet is clear that they are exactly what is needed, nor is there currently any automatic way for going from an informal specification to a formal one.

Challenge: Devise formal specification techniques for large reactive software systems (ones that have to deal with external events like mouse clicks or network interfaces) that can adequately express the functionality and behavior desired and with such additional desiderata as
- Efficient algorithms for testing the self-consistency of a specification.
- Sufficient clarity and simplicity that the people currently constructing requirements documents might be persuaded to use the formal techniques instead, thus clarifying the original designs and saving a step in the software design process.
- Efficient algorithms for assisting the process of telling whether the specifications are met by a given implementation, for instance by automatic generation of effective test data from the specification or by facilitating model-checking or full verification procedures.

These are impossible goals given that without severely restricting what the specifications can express, much of what we want to accomplish involves solving undecidable problems. However, any progress would be worthwhile. For a survey of current work in these directions, see Harel, "From Play-In Scenarios to Code: An Achievable Dream," *Proc. Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science 1783, Springer-Verlag, 2000, 22-34, and the references contained therein.

**Challenge: Automatic Transformation of Specification to System Design**

[Drafted 6/21]

Even if it is completely formal, a specification is a very different type of an object from a system design (a high-level description of an implementation), which might be expressed in terms of *state-charts* or some other automata-theoretic model. The specification talks about overall system behavior and response, the system design views the system as made up of components, and specifies how those components work and interact. It is a long step from the first to the second, but to the extent that we can automate it we save ourselves work (and opportunities for the inadvertent introduction of bugs).

Challenge: Design algorithms and tools for converting specifications into system designs. If necessary, simplify the task by imposing restrictions on what sorts of specifications are allowed (and how they are expressed) and what types of systems designs are allowed. If possible, arrange it so that the resulting systems designs come with proofs of correctness.

For more information on the current state-of-the-art, see the Harel paper cited in the previous challenge and its references.

**Challenge: Understanding Incomplete Programs**

[Drafted 6/27]

Software systems are generally build by designing a system architecture, consisting of interdependent parts. Many software projects have failed, or run significantly over time and over budget, because the early decomposition of a complex system into parts overlooks some important design objective. A challenge for software theory is to develop an understanding of incomplete programs. This theory should provide the basis for testing and evaluating designs that may be partially built, before person-years have been invested in complete implementation of a system.

**Challenge: From System Design to Code (and in between)**

[Drafted 6/21]

Given a formal design for a system in terms of state-charts of some other formalism, the problem of obtaining a full implementation in terms of some programming language can be viewed simply as a high-level compilation task. It does, however, involve it's own algorithmic issues and is not nearly as well studied. For instance, one view of what we are doing is constructing a distributed system, with the various systems components as communicating processes, so many fundamental theoretical issues are involved even before one starts worrying about things like efficiency. Even constructing high-level interpreters for a system design, as would be desirable for instance if we wish to test it before going to all the work of detailed
programming, can be non-trivial.

Again, much work has been done in this arena already, but it is an area with many remaining opportunities for theoreticians. For more information on the current state-of-the-art, see the Harel paper cited in the previous challenges and its references.

**Challenge: Querying the Software System**  
**[Drafted 6/27]**

Modern software systems are complex entities that are difficult to understand. An important challenge is to devise a query language for asking questions about an existing software system, along with appropriate algorithms for answering such queries. In effect, the challenge is to view an existing software system as a "database" that can be queried to understand properties of the system.

---

**Computer Aided Verification**

**Challenge: Next Steps for Symbolic Model Checking**  
**[Drafted 6/21]**

``Symbolic model checking'' is an algorithmic method for formally checking system designs that is widely used in the computer hardware industry and is beginning to show significant promise also in software verification and other areas and has been embodied in several commercial products. It has its roots in theoretical work on temporal logic from the early 1980's and before, but its current practical form arose when this work was wedded to the use of another formal concept, the "ordered binary decision diagram" (OBDD), to model hardware designs. It achieves its success in part by abandoning the general goal of proving a design completely "correct" for the more limited goal of proving that specific unwanted behaviors cannot occur. For such questions, it either verifies the claim or provides a specific example of how the unwanted behavior can occur, thus providing valuable feedback to designers. (In this way it can be more effective at finding bugs in designs than months of testing.)

For their work in inventing symbolic model checking, Bryant, Clarke, Emerson, and MacMillan won the 1998 ACM Kanellakis Theory and Practice Award. For a recent survey and tutorial on the field including many examples that illustrate the practical successes of symbolic model checking, see


Despite the successes, however, the fact remains that the basic problem that symbolic model checking attempts to solve is PSPACE-hard or worse. The algorithms used have running times that grow exponentially in the size of the system being checked, and although the constant of proportionality in the exponent is small, it is not that small. Thus the applicability of the method is currently limited to systems (or subsystems) of moderate size. We really need to be able to handle systems with 5000 state variables or more, which is an order of magnitude larger than what we can handle now. We'll need to handle even bigger systems if we are to have a hope of extending this approach successfully from checking hardware to checking software. Some possible directions:

- Develop practical model checking algorithms for reasoning about *hybrid systems* (systems involving both analog and digital components), *real-time systems*, and *security protocols*.
- Develop better heuristics for ordering the variables when constructing OBDD's. The size of an OBDD
can often be dramatically reduced given the right variable ordering.

- Develop automatic *abstraction* techniques for modeling subunits by less-complex designs that capture their main properties with fewer state variables.
- Devise more powerful *compositional reasoning techniques* so that large problems can be decomposed into more tractable subproblems and the answers for those composed to provide answers to the full problem.
- Find good techniques for reasoning about *parameterized* systems, for instance where the number of identical processes is a parameter.
- Develop practical symmetry-based reduction techniques to take advantage of the fact that hardware and software systems frequently embody many symmetries or near-symmetries.
- Combine *model checking* and *theorem proving*. Work is just beginning on this approach, but there may be useful synergies, and such an approach may be necessary for software verification.
- Develop techniques for extracting finite state *synchronization skeletons* from concurrent programs.
- Develop model checking algorithms for *infinite state systems* (including software with complex data types). This may require several of the other techniques on the list.

**Challenge: Model Checking without OBDD's**

[Drafted 6/21]

In "bounded model checking" one uses algorithms for SATISFIABILITY instead of OBDD's. We construct a boolean formula that is satisfiable if and only if there is a specific finite path of length $k$ in the underlying machine (representing an occurrence of undesirable behavior). We then look for longer and longer paths by incrementing $k$. After some number of iterations, we may conclude that no such path exists and the specification holds. For example, to verify safety properties, the number of iterations is bounded by the diameter of the finite state machine modeling the system being checked. See


For this approach to succeed, it needs to be successfully wedded to good algorithms for SATISFIABILITY. Two possibilities are worth exploring. First, we could use exact procedures, such as Davis-Putnam, for which remarkably fast implementations exist (exponential running times, of course, but small constants of proportionality in the exponent). These would be guaranteed to find errors if they exist (and we are willing to wait for the algorithm to terminate). A promising second approach might be to use some of the very effective heuristics recently developed for SATISFIABILITY, which can handle much larger instances. These may overlook some errors, but might still be quite effective at finding bugs.

**Challenge: Modeling Source Code**

[Drafted 6/26]

If we are to verify software, it will undoubtedly be easier to reason about formal models for the software than the software itself. To do this, however, we need practical techniques for deriving a sound abstract representation of a program from the program source text guided by a formal specification of (some of) the program's requirements, for instance based on "program slicing" algorithms.

---

**Networking**

**Challenge: Specifying, Constructing, and Verifying Communications Protocols**

[Drafted 6/21]
Although the research community interested in communications protocols does not have a strong overlap with the software engineering community, they face some of the same problems, although here, because of the relative simplicity of many communications protocols, there is some hope of constructing true proofs of correctness, for instance using I/O Automata as one's basic model. Unfortunately, much of this is still done by hand.

Challenge: Automate as much as possible the process of constructing and verifying communications protocols.

For a survey of current work in this area, see [references needed].

Challenge: Routing, Congestion Control, and Failure Recovery on the Internet
[Drafted 6/26]

Current routing protocols in use on the Internet, such as OSPF (Open Shortest Path First) and MPLS (Multi-Protocol Label Switching) have been designed so as to offer some control over congestion and recovery from link or node failures. However, we do not yet have good models for how best to use these controls, and with both protocols more control requires more communication between routers, more computation, and more local storage. This introduces scalability problems as the subnetworks governed by the protocols grow larger, a problem currently addressed by splitting the network into sub-areas and restricting the routes between the sub-areas.

How can we model and optimize performance (minimize latency and packet loss) in such a situation, taking into account cost, real-world (heavy-tailed) traffic models, network topology, link failure probabilities, etc.? How best can we in addition provide "quality-of-service" (QOS), which typically means at least the appearance to paying customers that a specified amount of bandwidth has been dedicated to their use?

More generally, are there better routing mechanisms than the ones currently being considered? More fundamentally, are real-time adaptive routing mechanisms feasible? Both MPLS and OSPF require a certain amount of set-up time, so that as presently constituted they must be optimized based on some static measure of the traffic demand matrix, either based on past measurements or forecasts; they cannot react to the instantaneous traffic changes common on the Internet. Indeed, network experts and operators are quite leery of real-time adaptive routing protocols, based on experiences of major "thrashing" when this was tried in the early ARPANET. Is it possible to develop such protocols that provably are free from disastrous behavior of this sort?

Challenge: Design a Replacement for the BGP protocol
[Drafted 6/21]

The BGP protocol (Border Gateway Protocol) mentioned in the Web section of this report has the task of helping subnetworks of the Internet to properly rout packets to other subnetworks. It unfortunately seems to have been devised with no precise specification in mind, and indeed it is difficult even after the fact to determine a meaningful specification of precisely what problem it is trying to solve. It has many defects that might have been avoidable, such as the ability to get caught in cyclic behavior where deliverable packets never reach their destinations.

Challenge: Provide a precise specification of the problem that BGP should be solving and design a network protocol that correctly implements this specification.

Database Systems

Challenge: Dealing with Large-Scale Data and Secondary Storage
[Drafted 6/20, Revised 6/22]

Many applications, both scientific and commercial, now can involve data sets that are so massive that they not only don't fit in main memory but don't even fit in the available disk space and must be stored on tertiary devices such as robotic tape storage (or even worse, on tapes that must be manually located and mounted). As databases become larger, the importance of taking the memory hierarchy into account becomes more and more crucial, and we need more accurately to model the differing costs of accessing data and storing intermediate results in cache, RAM, disk, tape, etc. when attempting to reduce the cost of executing database queries.

Challenge: Invent new ways of executing queries on large-scale data, using such secondary-storage models of cost, and possibly exploiting parallel and distributed processing when available. Especially, consider complex queries involving grouping and aggregation and multiway joins.

Challenge: Query Optimization
[Drafted 6/20, Revised 6/22]

A "query plan" is a detailed algorithm for constructing the answer to a query using basic database operations such as projections, joins, etc. Many plans can exist for a query stated in a language such as SQL. Query optimization is the process of finding the most efficient plan (or at least an adequately efficient plan). Much work has been done on special cases of query optimization, but much remains to be done, especially when the complex operations mentioned above are present, and when the availability of materialized views allows substitutions of views for certain portions of the query.

Even less is known about optimization for object-oriented and semi-structured data models (e.g., XML documents) than for the relational model, and for optimizing queries when subqueries are to a variety of different sources on the Web.

Background can be found in Garcia-Molina, Ullman, and Widom, *Database System Implementation*, Prentice Hall, 1999: ch. 2 for the model, ch. 6 for query execution, and ch. 7 for query optimization.

Operating Systems

Challenge: Protecting File Systems
[Drafted 6/22]

Formalize and provide examples of security for ``permissions'' in file systems. How can we model the ways programs access files, in order to prove that a set of programs cannot be used as a back-door to gain access to confidential files? Some notions of intentionality should enter into the definition. The definition must distinguish between a mail program being used as a back-door by an adversary to gain access to a file, and the mail being sent to the owner of the file, who responds by sending the file. Both have the same access pattern, but in the second the owner has given the requester ``permission'' to view the file.

This problem will become even worse as we increase our use and reliance on "mobile code," programs that our systems automatically accept from the network and then execute (JAVA applets are the current most common instantiation of this idea). How do we model the secure execution of untrusted code? Browsers currently attempt to do this, but the frequent reports of holes in their security schemes suggest that better
modeling and analysis are needed.

**Challenge: Optimizing Data Access**  
[Drafted 6/26]

For many tasks performed by operating systems, the main bottleneck is the time spend obtaining data from storage devices, be it instructions and data on pages of memory, files on disk or tape, or web pages on servers. Significant improvement in response times can often be achieved by tailoring the layout of data on these storage devices to the patterns of access commonly encountered, or when that is not possible, by re-ordering the access requests so as to minimize the time spent going from the end location of one access to the beginning of the next (this can be viewed as an instance of the "Traveling Salesman Problem").

The challenges are to model these processes and design

- Automatic techniques for "optimal" dynamic rearrangement of data in response to access patterns and
- Fast online reordering techniques for access requests so as to minimize overall latency, given a fixed data layout.

---

**Computer Architecture**

**Challenge: Exploiting Instruction Level Parallelism**  
[Drafted 6/20]

Recently-developed processors harness more and more hardware for improving performance by a variety of ways, including instruction-level parallelism. For instance, the *Vtune* package, [http://developer.intel.com/vtune](http://developer.intel.com/vtune), allows for performance tuning of application programs for each of Intel's recent processors, and AMD has similar tools.

1. Evaluate how well such processor-specific tuning fits algorithmic models we use, or could use.
2. Could PRAM algorithmics, or another algorithmic approach, lead towards a research agenda for new computer systems, where the responsibility of the programmer could be cleanly stated? Proceed to validate whatever you propose through modeling and various modes of quantitative analysis.

---

**Artificial Intelligence**

**Challenge: Inference Algorithms for Probabilistic Models**  
[Drafted 6/26]

Probabilistic modeling is the task of building a probability distribution that is an abstracted model of the world. We can then use that model to answer queries about the world based on partial observations. When we get observations, we condition the distribution in our model on those observations, getting a posterior distribution. We can then query the probability of an event in the model and use that as an estimate for the likelihood of that event in the real world.

For example, a hidden Markov model (which is very similar to a finite state automaton with probabilistic transitions) is typically used as the model for speech. You have a joint distribution over all possible trajectories of words, represented as a Markov model P(next word | current word), a distribution over trajectories of phonemes within a word, and a distribution over the acoustic signal associated with each phoneme. That defines a joint distribution over all word, phoneme, and acoustic signal trajectories. You can
then condition on a measured acoustic signal and get a probability distribution over the spoken words that might have generated that signal, taking into consideration the likelihood of one word to follow another, as well as how each word sounds. This technology is what is used by all state of the art speech recognition systems.

There are many probabilistic models that are substantially more complex than this one, and a fundamental algorithmic question in all of them is how you derive the posterior distributions from the observed measurements. These are computationally difficult inference problems and the best we can typically hope for are approximate solutions. What are the most effective algorithmic techniques, and how can we adapt them to exploit the structure of particular models? Does randomization help?

**Challenge: Geometric Object Recognition**

[Drafted 6/26]

A long-standing open problem in computer vision is that of recognizing objects in an image using a database of 3D models. We need to match both features and pose simultaneously, which potentially leads to a huge combinatorial blow-up. Some things that might help:

- Identify of invariants of objects that are preserved in all (likely) image configurations *and* are easy to measure.
- Define a (no doubt complex and non-Euclidean) distance function covering both objects and images.
- Develop data structures allowing fast approximate queries under this metric, so that we can quickly find objects that are ``near matches.''

---

**Scientific Computing**

**Challenge: Better Algorithms for Mesh Generation**

[Details to be filled in later]

**Challenge: Better Preconditioning Techniques**

[Details to be filled in later]

**Challenge: Eigenvalue Problems and Graph Partitioning**

[Details to be filled in later]

---

**Central Issues for Theoretical Computer Science**

[This section is based on contributions from Bernard Chazelle, Joan Feigenbaum, Maurice Herlihy, Russell Impagliazzo, Dexter Kozen, John Mitchell, Eva Tardos, Avi Wigderson, and Andy Yao. The section on Parallel and Distributed Computing is closely based on a draft by Herlihy and that on Logic, Semantics, and Programing Methodology is closely based on a draft by Kozen. An important overall resource was the following draft report prepared by the ``core theory'' panel chaired by Wigderson](Postscript).

---

**Cryptography and Security**

[Drafted 6/25, Revised 6/27]
Cryptography is the study of codes; or more generally, the use of secrets to control access to information. Only with the advent of formal notions of intractability has cryptography had a solid foundation. Although there was some previous work along these lines, the use of intractability as a basis for cryptography was made explicit in "New Directions in Cryptography" by Diffie and Hellman in 1977. This paper introduced the idea and basic applications of public-key cryptography, and was shortly followed by the announcement of a factoring-based public-key cryptosystem by Rivest, Shamir, and Adleman.

Together with work done in the early 1980's by Blum, Goldwasser, Micali and Yao, these papers set the agenda for theoretical cryptographic research. They sparked a flurry of activity: researchers formalized intuitive ideas of security, often requiring intricate modeling; introduced amazingly powerful new tools (public-key encryption, secure pseudo-random generation, bit commitment, zero-knowledge, secure multi-party computation); proved that, under reasonable intractability assumptions, almost any conceivable cryptographic task had a solution; explored the relationships between cryptographic tasks, and in some cases identified minimal assumptions needed for such tasks.

By the early nineties, theorists had an impressively complete picture of cryptography's possibilities. It became clear to the community that the time was overripe for implementing this vision, since implementation and use of cryptography was still fairly primitive. (Actually, the move towards implementing ideas from cryptographic theory started in Europe in the late eighties, whereas legal constraints delayed progress in the United States.) The military and banks used private-key encryption routinely, but public-key encryption and provably secure protocols were thought impractical for routine use. Theorists around this time began to move towards transferring theoretical ideas into practice. The new area of "practice-oriented provably secure protocols" drew on the ideas and techniques developed by theorists, but refined and clarified the models, analyzed protocols that were already in use, and replaced crude qualitative reasoning with intricate quantitative analysis. It should be noted that the pioneers in this new direction included many of the most important researchers in theoretical cryptography and their students. This wave of research came just in time. The Internet created a huge market for secure electronic commerce and encryption, and applied theorists were there to help guide the development of protocols for these needs.

Cryptographic theory is now in the position of being pulled rather than pushed. Research is often driven by society's immediate security needs. Unlike most areas of theory, there is a relatively short turn-around time in implementing new ideas. Industry absorbs as many Ph. D.'s in cryptography as graduate and begs for more. This has made cryptography a vital, dramatically growing, and exciting field. However, basic theory in cryptography is not growing at the same rate as applied theory. The well from which applied cryptography draws inspiration is in danger of running dry. Since society's security needs are so many, and since there is fierce competition for patents and products, there is the temptation to address those problems for which current ideas work rather than to push the limits.

This is not enough, however. There is still much work to be done in designing constructions for central tools like pseudorandom generators, digital signature schemes, and encryption schemes that are secure against chosen ciphertext attack. For these we have yet to obtain truly practical solutions that are still provably secure (based on believable intractability assumptions). Moreover, we do not yet have a good handle on the system security issues that are the main security threats. It is a truism that cryptography is only a small part of systems security, and that is a fair statement concerning the state-of-the-art. However, there is no a priori reason why systems security issues should not be formalizable and why there should not be a unified theory that includes protocol and systems security. Increasingly, cryptographic tools are being introduced at a systems level, as in Internet II, but without clear definitions of what attacks these tools are intended to counter.

The difference between protocol and systems security is that protocol design assumes that communication channels are vulnerable to attack, but the communicating processes are themselves secure. In other words, for the honest participants, the specified protocol is followed without error, secret information and internal
computation are opaque to attackers, and the hardware used to run the protocol is in the physical possession of the user at all times. While these assumptions may be valid for the duration of a fixed transaction, like a purchase via the Internet, they are invalid for most computer use. People install software on their systems which they do not understand, operating systems have bugs, information about internal processes are available to outsiders (e.g., timing attacks), and people share the computers that store their confidential information. The Internet's main security problems arise from settings where the end-to-end assumption is not valid: e.g., intellectual property protection; computer viruses; and security for agent-based and distributed computing.

**Challenge: General Tools for Proving Protocols Secure**

[Drafted 6/22]

Develop simple, sound formalisms for reasoning about protocol security, and extend these to enable proofs of security at the system level. While some formalisms exist that can find bugs in protocols, no general formalism that actually proves protocol security (from assumptions about the cryptographic tools used) has been given. Such a formalism would be a first step to developing automated tools for verifying correctness of cryptographic protocols. Hopefully, such tools could combine with similar tools for verifying correctness of communication protocols. This would allow us to avoid the problem that, while an underlying cryptographic protocol might be secure, bugs in implementation might make the resulting software insecure.

Systems issues that we will need to deal with include "man in the middle" attacks, as well as the preservation of security under concurrent execution. In most cases, the current security notions refer to a stand-alone execution of a protocol, and do not guarantee that security is preserved when many instances of various protocols are executed concurrently, a very real threat.

**Challenge: Automate the Design of Cryptographic Protocols**

[Drafted 6/25]

Simplify and automate design of efficient end-to-end protocols. This involves going beyond analysis of specific protocols to developing meta-tools for design and analysis. The goal of this direction is to allow users to design their own secure protocols for self-defined tasks without consulting a Ph. D. in computational theory. To this end, can we make security properties compositional, so that it will be easier to put together protocols out of component parts?

**Challenge: Expand the Reach of Zero-Knowledge Proofs**

[Drafted 6/27]

Zero-knowledge proofs and arguments have proved themselves as important building blocks in the construction of secure protocols. However, the arsenal of problems that have practical zero-knowledge proofs is limited.

- Find more problems that (a) have efficient zero-knowledge proofs and (b) are useful primitives in protocols.
- Find a general construction of zero-knowledge proofs where the number of cryptographic operations is smaller than the circuit-size for verification. The existence of such protocols is suggested by the work of Kilian (STOC'92), but he has to go through the theory of general probabilistically checkable proofs and so there is no saving in the total computational work.

**Challenge: Designing Secure Public Functions**

[Drafted 6/22]

Traditionally, security of cryptographic functions is based on a key that allows the function to be computed
but is unknown to the adversary, who only is allowed to interact with the function as a black box. But for many applications, e.g., an agent to retrieve medical or stock information, the complete description of the function needs to be given out. How do we formalize notions of security in these contexts? Some examples of such formalizations are collision-intractable functions (functions \( f \) such that given an input \( a \), it is difficult to construct a second input \( b \) such that \( f(b) = f(a) \)) and correlation-intractable functions. For the former, there are plausible constructions, but not for the latter.

An alternative approach would be to provide an incomprehensible program for computing the function, so that even if the function were to weak to meet the above restrictions, a user would be hard-pressed to discover the fact. For this we would need to formalize two additional notions and develop techniques for reasoning about them.

1. We need functions that are sufficiently unstructured that it would be difficult determine them from simply computing them for a moderate number of inputs.
2. We need a scheme for encrypting programs that does not change the function they compute and yet is not efficiently invertible.

**Challenge: Find Public-Key Cryptosystems that are Safe from Quantum Computers**

[Drafted 6/20, Revised 6/27]

The currently-used public key cryptosystems depend for their security on the intractability of factoring or computing discrete logarithms, both of which can be accomplished in polynomial time on a quantum computer (assuming we can build one). The proposed public-key cryptosystems based on hardness of the shortest vector problem do not appear to be practical. Are there practical public-key cryptosystems that do not rely on the hardness of problems known to be solvable in polynomial time by quantum computers?

Even better (presumably) can we design a practical public-key cryptosystem that is secure unless \( P = NP \)?

---

**Combinatorial Algorithms**

**Challenge: Find Polynomial-Time Algorithms for Long-Open Problems**

[Drafted 6/23]

Determining whether a polynomial-time algorithm exists for a problem is a generic ```challenge.` When confronted with a new problem, an algorithm designer's first step is usually to see whether the problem can be solved (exactly) in polynomial time. Although this does not in itself guarantee that there are practical algorithms for the problem, it is typically the first step on the way to finding such algorithms. (And if we fail to find a polynomial-time algorithm, this may be the first step on the path to proving that the problem is NP-hard and so practical algorithms that solve all instances quickly are unlikely to exist.)

Today, after almost 30 years of the theory of NP-completeness, most problems of known interest have been classified as to whether they are polynomial-time solvable or NP-hard. For certain fundamental problems, however, the questions of the existence of polynomial-time algorithms have been open for so long that they have become challenges for the field. Here is a short list of some of the most important:

- **Factoring:** Given that most researchers suspect this problem is not only hard in the worst case but on average (as was discussed in the Introduction), it is unlikely that a polynomial time-algorithm will be found for it. However, it is also not likely to be NP-hard. Or more precisely, the equivalent decision problem (Given a number \( N \) and a bound \( B \), does \( N \) have a prime factor larger than \( B \)?) is not likely to be NP-complete since it is in the intersection of NP and co-NP and cannot be NP-complete unless NP
is closed under complement.

- **Primality Testing:** This is the problem of testing whether a given integer $N$ is prime. Like factoring, this problem has significant commercial importance, since one must generate large primes in order to construct the keys in the RSA cryptosystem. Current schemes can only generate numbers with the guarantee that with high probability almost all of them will be prime. So this is a problem we would like to be easy. There are promising indications: Not only is it in both NP and co-NP, both it and its complement (given $N$, is it composite?) can be solved in random polynomial time, which leads many to believe that eventually we will find a (deterministic) polynomial-time algorithm for primality testing.

- **Graph Isomorphism:** This is another problem that cannot be NP-hard unless an unlikely hypothesis is true (in this case, that the "Polynomial Hierarchy" collapses). Moreover, a polynomial-time algorithm would be of great value, for instance in searching databases of chemical structures to determine whether a given chemical structure is already known. However, it and many other apparently-hard isomorphism problems are all of equivalent complexity, and so a polynomial-time algorithm would be considered quite surprising.

- **3-Processor Scheduling:** This is the problem of determining whether a system of unit-time tasks with precedence relations between them can be scheduled on three parallel processors to meet a given deadline. The variant in which there are only two processors is in P, and the variant in which the number of processors is part of the input is NP-hard. For any fixed number of processors exceeding 2, the problem is open, so the real question is what is the largest fixed number $m$ of processors such that the problem can be solved in polynomial time. This is a fundamental problem in scheduling theory that has been open for over 25 years and can serve as a representative for many similar questions about where the transition from polynomial-time solvability to NP-hardness occurs in a parameterized problem.

**Challenge: Find a "Strongly Polynomial-Time" Algorithm for Linear Programming**

[Drafted 6/23]

Linear Programming (optimize a linear function subject to a collection of linear inequalities) is a problem to which perhaps more computer cycles are applied than any other combinatorial optimization problem. Two classes of algorithms are used:

- Variants on the Simplex algorithm of Dantzig, which has worst-case exponential time but typically runs in low-order polynomial time on real-world instances.
- Interior Point Approaches such as Karmarkar's algorithm that in principle can run in worst-case polynomial time although the proofs of polynomial time may not survive the modifications (such as long step sizes) made to the algorithms to get them to run faster in practice.

Each type of approach dominates the other for some classes of instances, although as instances get larger and more dense, the interior point approaches win more often. They are one of the great successes of Theoretical Computer Science and it's point of view. Practitioners were relatively satisfied with the Simplex approach, despite its worst-case weaknesses, but theoreticians could not resist the question of whether Linear Programming was polynomial-time solvable. This led first to Khachian's discovery of the Ellipsoid Method, which ran in polynomial time but was impractical. The standard theoretician's quest for an algorithm with better asymptotic running-time bounds then led to Karmarkar's algorithm, which was practical. This introduction of competition for the Simplex method then inspired a great flourishing of theoretical and experimental work in the field, leading to major improvements in performance for both Simplex and Interior Point approaches.

However, a theoretical question still remains. When we say that Karmarkar's algorithm runs in polynomial time, we don't simply mean that the running time is bounded by a polynomial in the number $n$ of variables and the number $m$ of constraints, but by a polynomial in these two constraints and the number of bits needed to represent the instance. That is, instances with larger coefficients in the inequalities and objective function
may take longer time, even if the numbers of variables and constraints remain the same.

Now of course this unavoidable: One must read the instance, whose size itself may not be bounded by a polynomial in \( n \) and \( m \) given that the coefficients can be arbitrarily large. However, linear programming can be viewed as a problem in computational geometry, and for such problems a computational model that is commonly used and has given many useful insights in the past is the "real number random access machine (RAM)," in which one treats all numbers as having unit size, and all arithmetical operations as taking unit time. In this model, polynomial time means polynomial in \( n \) and \( m \). Of course, being polynomial in this sense would be of little value in the real world if in fact the algorithm ran in exponential time on a real-world computer. So what is really desired is what is called a "strongly polynomial-time" algorithm, one that runs in polynomial time both on the standard computer model and on the real number RAM. The Challenge is to find one.

The closest we have today is an algorithm of Tardos, which is strongly polynomial for instances in which all the coefficients in the constraints are in essence restricted to the values 0, 1, or -1.

**Challenge: Quantifying Approximability for NP-Hard Problems**

[Drafted 6/25]

Given that an optimization problem is NP-hard and hence efficient algorithms that are guaranteed to find optimal solutions are unlikely, a natural fall-back position is to search fast "approximation algorithms" that are guaranteed to find near-optimal solutions, for various definitions of "near." There is a natural hierarchy of possibilities:

1. **Fully polynomial time approximation scheme:** An algorithm that, given an instance and an epsilon > 0, always produces a solution within a factor of 1+epsilon, and does so in time bounded by a polynomial in the instance size and 1/epsilon.
2. **Polynomial time approximation scheme:** Same as above, except that now the running time need only be polynomial in the instance size for every fixed epsilon > 0.
3. **Constant ratio guarantee:** A polynomial-time algorithm that for some \( r > 1 \) and any instance, is guaranteed to produce a solution within a factor of \( r \) of optimal.
4. **Polylog ratio guarantee:** A polynomial-time algorithm that for some constants \( c,d > 0 \) and any instance, is guaranteed to produce a solution within a factor of \( c \log n)^d \) of optimal, where \( n \) is some key combinatorial parameter of the instance, such as the number of vertices in a graph or the number of tasks in a scheduling problem.
5. **Sublinear ratio guarantee:** A polynomial-time algorithm that for some constants \( c > 0 \), \( 0 < d < 1 \), is guaranteed to produce a solution within a factor of \( cn^d \) of optimal, where \( n \) is as before.

It is hard to think of anyone desiring an algorithm of type (5), but in fact there are problems, such as Clique and Graph Coloring, where this is the best one can hope for assuming P does not equal NP, and indeed for clique one cannot even have a polynomial-time algorithm of this type under only slightly stronger assumption. Indeed, for each of the five classes there are optimization problems whose best possible approximation algorithms lie in that class, assuming P does not equal NP. Many of these results have come in the last decade, with the lower bound results following as originally unexpected consequences of the characterization of NP in terms of probabilistically checkable proofs and its refinements. See for instance [need reference].

Given an NP-hard optimization problem, the natural challenge is to classify it as to its approximability class, and given that, to find the the best possible guarantee within that class that can be provided assuming P does not equal NP (a process involving both algorithm design and complexity-theoretic proofs of lower bounds). The algorithmic design challenge has given rise to many simple and broadly-applicable techniques (such as rounding followed by dynamic programming, geometric divide-and-conquer, the rounding of solutions...
obtained from linear or semi-definite relaxations, primal-dual methods, geometric embeddings, etc.).

And for some specific problems, we have pinned down the answer fairly precisely. Several important problems have long remained open, however. Here are some illustrative examples:

- The (asymmetric) "Traveling Salesman Problem" (TSP): In this problem one is given a set of \( n \) cities, and for each ordered pair \((i,j)\) of cities a distance \(d(i,j)\). The goal is to find a tour that visits all the cities and returns to its starting city, and minimizes the total distance traveled. This is an important problem that has applications from manufacturing and scheduling to computational biology. As stated, the problem does not have an approximation algorithm in any of the five above classes. However, if the "triangle inequality," which says that the the shortest distance from one city to another is the direct path, holds, then a polynomial-time \(O(\log n)\) approximation algorithm based on repeated matchings is known. A constant ratio guarantee might be possible, however. All we currently know is that no polynomial-time algorithm can guarantee a ratio less than 1.025 unless P = NP.

- The symmetric TSP: If the triangle inequality holds and the distance matrix is symmetric \((d(i,j) = d(j,i)\) for all pairs of cities), the situation is simpler, in that an algorithm due to Christofides is known to guarantee a ratio of 3/2. However, a major gap still remains, as all we currently know is that no algorithm can provide a guarantee better than 1.0078 unless P = NP. (If in addition the cities points in the plane and the distances are determined by one of the standard metrics such as Euclidean or Rectilinear, the classification is much tighter: the TSP has a polynomial-time approximation scheme but not a fully-polynomial time approximation scheme (assuming P does not equal NP.)

- Graph Partitioning Problems: These form the basis of divide-and-conquer type methods, and much attention has been devoted to good approximation algorithms for them. Yet, the best algorithms provide only logarithmic approximation guarantees (e.g., for sparsest cut, balanced cut, multi-way cut), and constant factor approximation algorithms, or every approximation schemes might be possible.

- Graph Coloring: We know that graph coloring in general is very hard to approximate, but one might hope that in special cases where only a few colors are needed, we might be able to do better. Unfortunately, even if we restrict ourselves to 3-colorable graphs, where the only NP-hardness results known simply rule out algorithms that use 4 or fewer colors (unless P = NP), the best polynomial-time algorithms currently known can only guarantee \(O(n^d)\) colors for \(0 < d < 1\).

- Closest Vector in a Lattice Problem. Here we are given the basis for an \(N\)-dimensional integer lattice and an arbitrary integer point \(x\) in \(N\)-space, and ask for the closest lattice point to \(x\) under the Euclidean metric. Here the game is changed somewhat, since we are asking for an approximation in terms of the numbers involved, rather than some combinatorial parameter of the instance, and the best we can expect, assuming P does not equal NP, is a polynomially-bounded approximation. However, the best approximation currently known is exponential.

- Bin Packing: Given a sequence of \(N\) numbers between 0 and 1, partition them into a minimum number of sets, each of which sums to at most 1. Here the game is changed in a different way. For this problem there is an algorithm due to Karmarkar and Karp that always produces a solution whose difference from optimal is bounded by a constant times \((\log N)^2\). The best lower bound we have, however, says that no polynomial-time approximation algorithm can guarantee a difference of 1 unless P = NP.

**Challenge: Understanding Metaheuristics**

[Drafted 6/23, Revised 6/25]

As seen above, many real-world optimization problems are not only NP-hard but also hard-to-approximate. Even when theoretically-good approximation algorithms are available, they may be too inefficient or
customers may want solutions that are even better than those that they provide. Thus, although the heuristics may well be used as a starting point, practitioners have for the most part been turning to various local improvement techniques, either straight local search or one of the ever-growing list of "meta-heuristic" variants on it, such as

- Simulated Annealing
- Genetic Algorithms
- TABU Search
- Neural Nets
- Greedy Randomized Adaptive Search Procedure (GRASP)
- Go-with-the-Winners
- Etc.

Currently the only way to tell whether an algorithm of this sort will be effective for a given problem is to implement and run it. We currently do not have an adequate theoretical understanding of the design of search neighborhoods, rules for selecting starting solutions, and the effectiveness of various search strategies. Initial work has given insight in narrow special cases, but no useful general theory has yet been developed. We need one.

**Challenge: Faster Polynomial-Time Algorithms**

[Drafted 6/25]

Over the years an important theme in theoretical computer science has been to search for ever faster (in an asymptotic sense) algorithms for problems of key importance in computing, such as matrix multiplication and may commonly encountered graph-theoretic problems. Many algorithms of great practical value have resulted from this search, from interior point algorithms for linear programming to "push/relabel" algorithms for Min Cost Network Flow. For many widely-applicable problems, however, our best algorithms still fall significantly short of the ultimate goal of linear-time operation, although we currently don't know of any theoretical obstacles to attaining such performance. (At present, theory does not provide techniques for proving specific superlinear running-time bounds on algorithms in machine models that accurately reflect the potentials of real computers.) Thus the question arises, *can* linear running time be achieved for such problems. Some specific challenges (there are many other important ones as well):

- **Multiplying $N$ by $N$ Matrices.** For linear time, we would have to do this using only $O(N^2)$ arithmetic operations. The naive method takes proportional to $N^3$ operations, while the best currently-known theoretical algorithm uses $O(N^{2.376})$ operations, and the best practical algorithm (originally only thought to be of theoretical interest but actually outperforming alternative approaches for $N$ as small as 100 when the matrices have few non-zero entries) is Strassen's $O(N^{2.81})$ algorithm. (Note that the computational model here is the "real number RAM" mentioned above and we are actually talking about strongly polynomial time; ideally we would also like the algorithm to be linear in the size of the input when the lengths of the binary representations of the matrix entries is taken into account.)

- **Assignment Problem:** Given a weighted bipartite graph, find a maximum weight set of edges, no two of which share a common endpoint. Here the best running-time bound known has for many years been $O(N^3)$, versus the $O(N^2)$ of linear time.

- **Min Cut/Max Flow:** Given a network with identified source and sink vertices and capacities on the edges, what is the maximum amount of flow that can be pushed from source to sink? There are a variety of currently "best" algorithms for this problem depending on the relative values of $N$ and $M$, the numbers of vertices and edges in the network (for instance, see Cherkiyan, Hagerup, and Mehlhorn, "An o(n^3)-Time Maximum-Flow Algorithm," *SIAM J. Comput.* 25 (1996), 1144-1170). For none of these possibilities, however, does the best algorithm come close to the $O(M)$ that would be needed for a $N/\log N$.

- **Single Source Shortest Paths:** Given a weighted graph and a specified vertex, find the distances from
that vertex to all other vertices. Here we can solve the problem in time $M \log N$ using Dijkstra's algorithm, much closer to linear time, but still not there.

- **Minimum Spanning Tree**: Given a weighted graph, find a minimum cost subtree that includes all the vertices. For this problem we are almost as close to linear time as one can possibly get there without being linear, but there is still a gap. The best current algorithm, due to Chazelle, "A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity," *J. ACM* (in press), has running time $O(m \alpha(m))$, where $\alpha(m)$, the inverse-Ackermann function, is an incredibly slowly growing function that pops up as a seemingly inherent aspect to many problems in algorithms and geometry.

If we cannot achieve linear-time for these problems, how fast are the ultimate limits on what we can do?

**Challenge: Develop a Science of Algorithm Testing**

[Drafted 6/27]

Promising theoretical results about an algorithm often do not suffice to convince practitioners of the algorithm's usability. If the algorithm is to have practical impact, it typically must be implemented and tested. At present, however, we do not have a solid science about how to do this. Testing algorithms is different from testing paints or running shoes. In principle, there is a wide variety of questions one might which to answer, such as how does performance depend on instance type, on instance size, where are the cross-over points with its competitors. Moreover, in the case of approximation we face bi-criteria questions about potential tradeoffs between running time and quality of solution, where the tradeoffs may themselves depend on instance type and size. How can one best illustrate the effect of various parameters and other design decisions made in the algorithm's implementation? More fundamentally, how does one provide evidence correctness - that the algorithm as implemented is indeed the one intended?

Although in recent years a collection of rules-of-thumb has been accumulating for how to address some of these questions, we currently only know how to apply them on a case-by-case basis. Some specific questions:

- What constitutes an adequate set of test instances? How does this depend on the types of questions being asked?
- What are the standard questions that should be asked?
- How can we insure comparability between tests by different implementers on different machines?
- What are effective methodologies for performing exploratory experimental analysis in hopes of finding provable hypotheses about algorithmic performance?
- What is the proper role for experimental work in complementing theoretical results?

**Challenge: One-Pass, Space-Efficient Algorithms for Massive Data Streams**

[Drafted 6/19, Revised 6/22]

Massive data sets are increasingly important in a wide range of applications, including observational sciences, product marketing, and monitoring and operations of large systems. In network operations, raw data typically arrive in "streams," and decisions must be made by algorithms that make one pass over each stream, throw much of the raw data away, and produce small data structures for further processing. Examples of such streams are the "Netflow" data produced by Cisco routers that record information about each individual flow of packets that pass through the router and "click streams" that record the sequence of choices made by users (often large numbers of users) at the various servers hosting a given web site.

As in the above two examples, network-generated massive data streams are often "distributed": Several different, physically separated network elements may receive or generate data streams that, together, comprise one logical data set; to be of use in operations, the streams may have to be analyzed locally and their synopses sent to a central operations facility. Develop a comprehensive algorithmic toolkit to deal with
the enormous scale, distributed nature, and one-pass processing requirements on network-generated massive data streams.

References (to preliminary work along these lines):

- Henzinger, Raghavan, and Rajagopalan, ``Computing on Data Streams,'' DEC SRC TR 1998-011

Notes: (1) Space-efficient, one-pass algorithms will, of necessity, be randomized and approximate. In particular, it often suffices to determine whether a massive data stream is ``close'' to one that has the desired property or ``far'' from any that has the desired property, for some appropriate notion of ``close'' and ``far.'' Another important application in which approximate results may be satisfactory is that of clustering the data contained in streams to identify groups of similar users, etc. These types of approximation are not considered, for example, in the classical theory of one-way automata, and that's one of the main reasons that a new theory is needed. (2) An attractive algorithmic design strategy is ``stream through the data once, extracting a small subset, and then do an intensive computation on that subset.'' Two well-known successful applications of that strategy are:


---

**Computational Geometry**

Computational geometry's role is to be a two-way pipeline between algorithmic investigations of a theoretical nature and applied geometric computing. Recent work has given us good reasons to cheer but also many directions for future work.

First, the cheers: Virtually all of the classical problems posed 22 years ago (the field is widely considered to date from the time of publication of Shamos' thesis in 1978) have been solved. Some of the successes were merely the low-hanging fruits to be expected of any new discipline; others were deep achievements of lasting consequence. A brief review of some of them can serve to inform the rest of the discussion.

- The study of arrangements has led to a full understanding of Davenport-Schinzel sequences and near-optimal zone theorems (eg, how high is the combinatorial complexity of the free-space within which a robot can move?).
- Linear optimization with a fixed number of variables (parametric searching, ``LP-type'' formulation) is solved. For example, one can solve linear programs and can find the Lowner-John ellipsoid of $n$ points in $\mathbb{R}^d$ deterministically in linear time for any fixed $d$, a rather amazing fact!
- Data structures have been developed for configuration searching: range searching, point location, intersection, visibility structures. Computational geometry has contained the most exciting recent
``action'' on data structures.

- Convex structures are well-understood. For example, optimal algorithms are known to compute convex hulls and Voronoi diagrams in any fixed dimension.
- Triangulation and optimal decomposition of shapes into canonical components.

One of the most stunning accomplishments has been in the area of geometric sampling. Work on (de)randomized algorithms has led to optimal constructions of sampling tools for spaces of bounded Vapnik-Chervonenkis dimension. The discrepancy method (ie, the linkage between discrepancy theory and computational sampling) lies behind several of the most important breakthroughs (eg, convex hulls, diameter, range searching). Could these breakthroughs have been achieved years earlier? In view of the algorithmic sophistication and the long chain of results that make the discrepancy method effective, the answer is, most likely, no.

Most of the problems studied in CG have trivial polynomial-time solutions (because the dimension is often kept fixed). As with data structures, the name of the game is not to separate complexity classes but to nail down the true complexity of a problem. (The motivation for doing so is not just practical: it is primarily that only an optimal algorithm can vouch for our true understanding of what's going on. By analogy, think of those (logn)^2-depth sorting networks: to shave off the extra log factor required rewriting the whole book on sorting. A similar story has happened time and again in CG.)

From that (theoretical) perspective, randomization does not seem to help at all. Virtually every randomized algorithm we know has a deterministic version. At the same time, the only optimal deterministic solutions for some of the most important problems in CG are derandomizations. So, randomness appears to be an indispensable methodological tool but not really a complexity tool.

**Challenge: Open Problems in Computational Geometry**

[Drafted 6/25]

What's left to do in computational geometry? As is all areas within theory, lower bounds lag behind algorithms. We begin this list with perhaps the most conspicuous gap between upper and lower bounds in CG.

- What is the complexity of certifying general position: given a bounded-degree polynomial $p$ in $Q[X_1...X_d]$ and $n$ reals, do any $d$ of them form a zero of $p$?
- Can we find a singly-exponential algorithm for computing a triangulation of a real semi-algebraic set (or at least its homology groups)?
- How hard is geometric searching in a non-fixed number of dimensions? (extending current work on NN search to general searching).
- Find novel representation of shapes. In particular, we want representations that allow statistical inference, learning, matching and classification of shapes. We should bear in mind that standard probabilistic distributions are by and large irrelevant to real-world CG.
- Solve the ``big'' open problems in combinatorial geometry, eg, the size of $k$-sets, and single-face complexity for algebraic varieties.
- Generalize to computational topology (especially, a ``multiscale'' version that captures both geometry and topology).
- Address output-sensitive algorithms (eg, hidden surface removal).

---

**Parallel and Distributed Computing**

The basic challenge here is that the reality of distributed computing is changing, and so the theory of
distributed computing had better change along with it. Simplifying somewhat, distributed computing and parallel computing are converging. A classical distributed system might consist of a collection of workstations that communicate by exchanging messages over a slow and unreliable network. By contrast, a classical parallel system might consist of a number of processors (or threads) that communicate through a relatively fast and reliable shared memory. Technology trends are forcing these two domains closer together.

The evolution of network communication will radically change distributed computing. Network bandwidth and reliability is increasing while latency is decreasing. Wireless communication will soon be ubiquitous, and eventually each toaster will have its own IP address. At the software level, shared memory programming models are displacing traditional message-passing models. While message-passing will always have some uses (such as hard-core scientific programming), mainstream distributed computing is moving toward a shared-memory model. Network file systems make a plurality of file systems appear as one, and distributed shared memory systems promise a world where workstations can coordinate without the bother of explicit message-passing. The demand for intellectual piracy has produced systems such as Napster and Gnutella that tie together the file systems of thousands of individual machines. These systems resemble the shared-memory models of classical parallel computing.

Multiprocessor architectures are moving in the opposite direction. Processors become faster each year, and memory speeds, while increasing, cannot keep up. The net effect is as if, each year, memory were to become slower and farther away. The multiprocessors of antiquity provided a simple memory model in which memory reads and writes occurred atomically in the order they were issued. These systems made extensive use of caching and message-passing interconnect, but these mechanisms were hidden at the architectural level. This level of abstraction was kind to programmers, but it has become too slow and too expensive. Instead, modern processors support a bewildering variety of weaker memory models that reveal the existence of underlying caching and network-based communication. These models vary widely in terms of clarity and rigor, but they increasingly resemble the message-passing models of classical distributed computing.

This convergence provides distributed computing with two kinds of challenges: applying distributed techniques to new problems that arise as a result of this convergence, and borrowing techniques from the parallel domain to address distributed problems.

One challenge is to apply intellectual tools developed for distributed systems to parallel systems. Modern parallel architectures are really classical distributed systems. We can treat them as space-bounded automata that communicate by message-passing. For results to be effective, we have to take the space bounds seriously - in hardware, space is still expensive. Notions of fault-tolerance have always been central to the distributed computing community, while the parallel computing community has not paid as much attention to this subject. As parallel machines become larger and more distributed, fault-tolerance will necessarily assume a more central role in parallel systems.

The distributed computing community has well-developed theories in many areas, including atomicity and serializability, network algorithms, replication lock-free synchronization, competitive analysis of anything that moves, and security. The distributed computing community can have an impact on parallel architectures by applying this rich legacy to a theory of how to implement useful shared-memory abstractions on top of networks of (severely) space-bounded automata.

Another promising area is a comprehensive theory of multiprocessor synchronization. Modern architectures typically provide efficient but weak and cumbersome synchronization primitives, while programmers want efficient but powerful and simple primitives. To bridge this gap, we need a new complexity theory for synchronization. For example, compare a single-word compare-and-swap, a $k$-word compare-and-swap, and a $(k+1)$-word compare-and-swap. The larger the number of words, the easier to design parallel data structures, but the more complex the underlying hardware. Right now, we do not understand the range of possible trade-offs.
Modern processors use deep pipelines and aggressive speculative execution. The classical theory of distributed computing has a well-developed theory of transaction systems, optimistic concurrency control, optimistic recovery, discrete-event simulation, and distributed checkpointing. It seems there should be some connection.

For distributed computing, like all areas of theory, changes in technology present both a challenge and an opportunity. The challenge is to deepen our understanding of new phenomena. The opportunity is to develop new and heretofore unimagined areas of theory.

**Complexity Classes**

In this section we will talk about challenges having to do with the fundamental nature of computation. What is the value of various computational resources, such time, space, nondeterminism, randomization, etc. Such questions can typically be modeled in terms of *complexity classes*: sets of problems that can be solved given a certain amount of the resource under consideration. Questions about the computational power of various resources can then be viewed as questions about containment relations between classes. Here we shall present a sampling of the most important, beginning with the most famous open problem in the field.

**Challenge: The Power of Nondeterminism -- Does P = NP?**

[Drafted 6/26]

Nondeterminism in this sense can be viewed as the ability to guess correctly in the time it takes to write down the guess. P is the class of decision problems solvable in deterministically in polynomial time; NP is the class of decision problems solvable *nondeterministically* in polynomial time, more precisely, those decision problems such that if the answer is ``yes,'' then there exists a short proof of this fact (the guess) that can be verified in polynomial time. By definition P is contained in NP, but it is almost universally believed that the containment is proper, i.e., that there are problems where finding a short proof is superpolynomially more difficult than verifying one once it has been found.

The resolution of this challenge will have wide ranging implications for computing, as one can readily see from the number of times the phrase ``assuming P does not equal NP'' has occurred in this report. Indeed, the significance of this 30-year-old open problem is widely recognized both in and outside the field. Indeed, it is first on the list of seven open problems in mathematics for the solution of which the Clay Mathematics Institute is offering $1,000,000 prizes. See [http://www.ams.org/claymath/](http://www.ams.org/claymath/).

Note that there are ways that this problem might be resolved that would still leave us with challenges. For instance, it could be the case that P = NP, but the algorithms for solving NP-hard problems in polynomial time take time proportional to n^100 or worse. Even worse, one could conceive of a proof that shows that P = NP using non-constructive proof techniques, i.e., a proof that shows that polynomial-time algorithms exist for NP-hard problems without in fact exhibiting any. Finally, we could be left in an even worse limbo: some theorists hypothesize that the problem of P versus NP might be independent of the axioms of set theory and hence in principle *not* resolvable.

At any rate, little progress has been made on this problem, except to show that certain techniques, such as diagonalization, won't work, and indeed that it is unlikely that any ``natural proof'' in the sense of Razborov and Rudich, ``Natural Proofs,'' *J. Computer and System Sciences*, **55** (1997), 24-35) can do the trick.

Other problems about the power of nondeterminism also remain open. For instance, does L, the class of problems solvable in logarithmic space, equal NL, the class of problems solvable nondeterministically in log space. Note that we know by an old result of Savitch that anything solvable nondeterministically in space f(n)
can be solved deterministically in space $O(f(n)^2)$, which implies that deterministic polynomial space does equal nondeterministic polynomial space, but there is still a gap to be closed when we consider this issue at the logarithmic space level.

**Challenge: The Power of Randomization -- Does P = BPP?**

[Drafted 6/26]

BPP is the class of decision problems solvable by randomized algorithms that for all instances have probability at least 2/3 of getting the answer correct. (2/3 can be replaced in this definition by any constant greater than 1/2. The key point is that with such an algorithm, one can guarantee correctness with arbitrarily high precision simply by running the algorithm over many times using independent sets of random numbers.

The area of randomization and derandomization of algorithms is one of the most fascinating and productive in TCS. The question it addresses is: is the ability to make random choices of use in making computation more efficient? Many basic problems, such as primality, have probabilistic algorithms that are dramatically more efficient than the best known deterministic methods. However, the consensus is emerging that such probabilistic algorithms can (in most models) be non-trivially simulated by deterministic ones. However, probabilistic algorithms still seem to be a very important tool in algorithm design.

This field is one of the great success stories of TCS, and represents a combined effort of combinatorics, structural complexity, cryptography, geometry, and circuit complexity. There are numerous techniques to derandomize specific algorithms, to use many fewer random bits than obvious in general classes of probabilistic algorithms, and to allow them to run if the sources of randomness are extremely weak.

Based on these successes, many now believe that P = BPP. Note that this would not mean that randomization is useless, merely that it cannot provide a superpolynomial speedup over a deterministic algorithm. Can we prove BPP = P? This seems much harder. However, the related weaker statement that BPP does not contain EXP, the set of problems solvable in exponential time seems possible. All this statement says is that some problems in EXP cannot be solved efficiently (with randomness), and we already know by diagonalization arguments that they cannot all be solved efficiently without randomness.

As to the related problem of L versus BPL (deterministic logspace versus probabilistic logspace), here again equality is suspected, and we may be very close to proving BPL=L without any assumptions!

**Challenge: The Power of Complementation -- Does NP = co-NP?**

[Drafted 6/26]

This question concerns a perhaps non-intuitive computational resource: complementation. Decision problems are problems that have either yes or no answers. Is there an inherent computational difference between asking whether the answer is "yes" or asking whether it is "no"? As might be expected, this has as much to do with logic and proofs as to do with computation. Consider a problem in NP, such as "given an instance of SATISFIABILITY, does it have a satisfying truth assignment?" If the answer is yes, there is a short proof, consisting of the desired proof assignment itself. If the answer is no, however, it is not so obvious that a short proof exists that will convince one of the fact. If we define co-NP to be the class of all complements of problems in NP (problems in NP with the roles of yes and no reversed). If NP does not equal co-NP, as most believe, then the above apparent asymmetry is real.

One reason theorists believe that NP does not equal co-NP is that equality would have unlikely consequences, such as the collapse of the polynomial hierarchy (a hierarchy based on the computation resource of alternation). Unfortunately, if NP does not equal co-NP, then P cannot equal NP (since P is closed under complement), so we are not likely to prove this fact soon.
Assuming that the two classes are not equal, then an interesting third class will exist and be distinct from each: the intersection of NP and co-NP. This class contains P, but again is not thought likely to equal it, since several non-trivial problems that are not thought to be polynomial-time solvable lie in this intersection, the most famous being the decision problem version of factoring (given $N$ and $B$, does $N$ have a prime factor larger than $B$?).

Note that the analog of this problem for logarithmic space has been solved. As shown (independently) by Immerman and Szelepcsényi, $L = \text{co-L}$.

**Challenge: The Power of Space -- Does P = PSPACE?**

[Drafted 6/26]

This question concerns the power of space (memory) as a computational resource, where PSPACE is defined to be the class of problem solvable in polynomial space. In polynomial time we can only use polynomial space, so that P is clearly contained in PSPACE. It is also easy to see that NP is contained in PSPACE, so that $P = \text{PSPACE}$ would imply $P = \text{NP}$. The question P versus PSPACE question remains open, however, and its significance has grown over the years as PSPACE has been shown to be equivalent to many other independently-defined complexity classes, such as ones based on time-bounded alternation (APTIME), interactive computation (IPSPACE), and "games against nature." Many presumably intractable problems in such diverse fields as game theory, automata theory, and logic are "PSPACE-complete" and hence will not have their true intractability confirmed until we succeed in proving that P does not equal PSPACE.

---

**Lower Bounds**

**Challenge: Develop Unnatural Proof Techniques for Proving Lower Bounds**

[Drafted 6/26]

Razborov and Rudich, in their paper "Natural Proofs," *J. Computer and System Sciences*, 55 (1997), 24-35, defined a class of proof techniques that seems to capture most of the ideas researchers have to date used to prove lower bounds on the size of circuits (and hence on the running time of algorithms). Moreover, they showed that based on reasonable assumptions such as the one that factoring is hard on average, the powers of natural proofs are strictly limited. For instance, they won't be usable to prove P does not equal NP.

An important challenge is thus to invent new proof techniques for proving lower bounds that are not natural (or adapt old unnatural techniques not previously used with much success in this domain).

---

**Logic, Semantics, and Programming Methodology**

These topics are of a more linguistic flavor than complexity theory or algorithms and are represented more heavily at logic and programming language theory conferences such as LICS and POPL than STOC and FOCS. Nevertheless they constitute an integral part of theoretical computer science and there is considerable interaction between logic and complexity.

We shall begin with a brief survey of some of the topics that make up this area, and conclude with some illustrative challenges. As with complexity theory and algorithms, one could enumerate many instances of the impact the field has had on computing practice (and we will mention a few prominent ones), but here we will focus primarily on more theoretical issues at the core of the topic. Because of space and time limitations, we will touch on only a few highlights out of a variety of possibilities. In particular we will not dwell on classical program verification, which is covered in part elsewhere in this report. Of course, basic questions
such as deductive completeness, compactness, Loewenheim-Skolem phenomena, complexity of various decision procedures, and the relative expressive power of programming and assertional constructs are always of interest.

**Deduction as Computation**

It has been known since the earliest days of theoretical computer science—the time of Goedel, Church, Turing, Curry, Schoenfinkel, Post, and Kleene—that logic and computation are inextricably linked. A proof is a sequence of applications of logical rules, similar to the execution of a program. The interplay between these two subjects is a pervasive theme and shows up time and time again in various guises.

Perhaps the ultimate expression of this relationship is the Curry-Howard Isomorphism, or the propositions-as-types principle. In this view of logic, a proposition is a type (in the programming language sense), a constructive proof of that proposition is a program that has that type, and proof-checking is just ordinary type-checking. For example, a conjunction "phi AND psi" is regarded as a product type "phi · psi"; a proof of the conjunction is a pair \((p,q)\) where \(p\) is a proof of \(phi\) and \(q\) is a proof of \(psi\). An implication "\(phi \rightarrow psi\)" is regarded as a function type; a proof of the implication is a function taking proofs of \(phi\) to proofs of \(psi\). A constructive proof in natural deduction style is essentially a program that computes something, and an explicit representation of the program in the form of a lambda term can be extracted automatically from a natural deduction proof. This far-reaching relationship extends all the way up through second-order logic and the polymorphic typed lambda calculus. Formal systems of mechanized mathematics such as Nuprl are based on this theory.

**Duality**

Another widespread and fundamental phenomenon, which is related to the above, is the algebraic/topological duality of specifications and implementations. The classical (non-dynamic) form of such a duality is the Stone duality between Boolean algebra (propositional logic) and certain topological spaces called Stone spaces. The phenomenon also arises quite often in computer science, but here the algebraic structures are typically program logics and the dual objects are typically automata of one form or another. For example, there is a duality between set constraints used in program flow analysis and a type of tree automata. Relationships such as this sometimes lead to remarkable confluence of activities; in the case of set constraints, it led to the unification of two extensive and previously disjoint bodies of research in program analysis and type inference. Duality has been observed on many different levels in programming language semantics as well, especially in denotational semantics. Such dualities give much insight because they allow one to look at problems from different viewpoints. In some cases this leads to faster algorithms as well, or decision procedures where none were known.

**Fixpoints and Induction**

Inductively defined objects are omnipresent in computer science: inductively defined datatypes such as lists, trees, and stacks, programming language syntax and semantics, etc. Perhaps the single most heavily used proof technique in computer science is induction. It is ironic that the most prominent classical logic, first-order logic, is inadequate for dealing with inductively defined objects in a natural way. It is only through the grace of the coding power of the natural numbers that it is possible at all. Over arbitrary structures, first-order logic is not powerful enough to define even something as fundamental as transitive closure. One gift of computer science to the field of logic, besides the realization of the importance of induction, is the definition of a variety of logical formalisms that can deal naturally with inductively defined objects.

**Finite Model Theory**
There are several examples, beginning with Fagin's characterization of NP, of characterizations of complexity classes by logical formulas over finite models. Several major open problems in complexity theory translate to relative expressiveness results of various logics on finite ordered models. For example, the P vs. LOGSPACE problem is equivalent to the relative expressibility of the mu-calculus and transitive-closure logic. By now the subject of finite model theory is a well-developed subject and has generated considerable interest among pure logicians.

Type Theory

Types are one of the greatest contributions of the field of programming languages. Types allow programmers to express their intentions as to the meanings of variables and expressions and impose a statically discernible structure to otherwise raw data. Type checking determines that expressions are consistent with this intended usage, which imposes a discipline on the programming process and allows more errors to be caught earlier.

The theory of types is burgeoning well beyond its original roots. The theory of types touches on the very foundations of mathematics and some foundational systems have been developed that can serve as a foundation for mechanizing mathematics: NuPrl, Martin-Lof type theory, the Calculus of Constructions. These systems are reflexive in that they can talk about themselves, an essential feature.

Types and typing have been recently been shown to apply in unexpected areas. For example, much of program analysis in compiler theory can be formulated in terms of type-checking. Classical program verification using Hoare logic with partial correctness assertions is essentially type checking. As the theory becomes more and more developed, it becomes applied in more unexpected situations, such as storage management and resource usage analysis, which previously might have been thought to be too dynamic to handle statically with types.

One aspect of types that programmers sometimes complain about is that one must specify them. However, modern functional programming languages such as ML have a type inference mechanism that will infer types from usage. This gives strong typing without having to specify types explicitly, still being able to identify type inconsistencies at compile time.

Object Oriented Programming

In traditional programming, control flow is primary and data are secondary. In object oriented programming, it is the opposite. Object oriented features are extremely useful in providing encapsulation and protection mechanisms and promoting modularity and code reuse. The theory of object oriented programming is currently a very active area of research, with at least three major conferences devoting significant portions of their time to the topic, some exclusively. The semantics of objects, especially their interactions with types, are not fully understood.

Alternative Programming Paradigms

Programming language theorists have developed and studied a rich variety of alternative programming paradigms: constraint-based languages, functional languages, rewrite systems, logic programming. Advanced functional languages such as ML and Haskell have many advanced features such as polymorphic type inference. They are considered the next step beyond the current popular generation of programming languages such as Java or C++. Programming language theorists are ever exploring the structure and meaning of such languages in the desire to simplify the programming process.

Logic and Automata
A significant advance in complexity of logics of programs is the development of automata-theoretic decision procedures. The exact complexity of many logics has been established over many years of work by many authors. The first use of this technique was the Buechi/Rabin result on the decidability of second order theories of successor using automata on infinite strings and trees. This has been further developed by several researchers over many years, and by now it is the primary technique for obtaining efficient decision procedures for satisfiability problems. At present the complexities of most major program logics are known exactly, and some of these proofs are quite deep, involving nondeterministic and alternating automata on infinite objects with various kinds of acceptance conditions.

Other related problems such as model checking and synthesis of models from specifications are handled in this framework as well. Model checking has has a large and visible impact on the industry.

Specification Languages

There is a tendency toward higher and higher level programming constructs, so that the programming process becomes more of a declarative instead of imperative exercise. This development, along with such techniques as partial evaluation and abstract interpretation, clouds the distinction between specification and programming.

*******************************************************************************

Challenge: Extend Programming Language Semantics
[Drafted 6/27]

The denotational semantics of functional programming languages is well understood. However languages that combine functions with assignment, storage allocation, objects, and concurrency are beyond the scope of current methods. A major challenge is to develop a grand unified field theory of programming languages, providing a single semantic framework that can be used to develop program analysis methods, program correctness methods, and other methods that require a precise understanding of the meaning of a computer program.

Challenge: Extend the ``Formulas-as-Types'' Paradigm
[Drafted 6/27]

The formulas-as-types correspondence has been the basis of many advances in programming languages over the past 10-15 years. This paradigm, which relates computation to logic, is well understood for functional programming languages with relatively complex type systems (including polymorphism, generic program modules, and restricted forms of objects). However, it remains an important challenge to extend this paradigm to object-oriented languages, imperative languages, and concurrency. Advances in this direction will provide improved static checking of program properties, leading to significant improvements in our ability to find errors in programs, and improved formal logics for program verification.

Challenge: Devise a Theory of Programming Language Expressiveness
[Drafted 6/27]

Virtually all programming languages in common use are equivalent in the coarsest sense: in each language, the set of definable functions on the integers (or any set recursively isomorphic to the integers) is the set of partial recursive functions. However, most programmers make distinctions between languages. For example, some languages are better for quick programming of simple tasks while others may be better for producing large-scale maintainable software systems. An important and difficult challenge is to devise a theory of programming language expressiveness that makes meaningful distinctions between practical programming languages. One possible way to distinguish between languages is by comparing the ability to localize
program design decisions. Another possible dimension that would shed light on software development practice is to compare languages according to the ability to express operations on program modules.

Learning Theory and Statistical Inference

(Still to be Written)