1
Introduction

The purpose of computing is insight, not numbers.
Richard Hamming, *Numerical Methods for Scientists and Engineers*

Some questions:

- You are a working programmer given a week to reimplement a data structure that supports client transactions, so that it runs efficiently when scaled up to a much larger client base. Where do you start?
- You are an algorithm engineer, building a code repository to hold fast implementations of dynamic multigraphs. You read papers describing asymptotic bounds for several approaches. Which ones do you implement?
- You are an operations research consultant, hired to solve a highly constrained facility location problem. You could build the solver from scratch or buy optimization software and tune it for the application. How do you decide?
- You are a Ph.D. student who just discovered a new approximation algorithm for graph coloring that will make your career. But you're stuck on the average-case analysis. Is the theorem true? If so, how can you prove it?
- You are the adviser to that Ph.D. student, and you are skeptical that the new algorithm can compete with state-of-the-art graph coloring algorithms. How do you find out?

One good way to answer all these questions is: *run experiments to gain insight.*

This book is about *experimental algorithmics*, which is the study of algorithms and their performance by experimental means. We interpret the word *algorithm* very broadly, to include algorithms and data structures, as well as their implementations in source code and machine code. The two main challenges in algorithm studies addressed here are:

- *Analysis*, which aims to predict performance under given assumptions about inputs and machines. Performance may be a measure of time, solution quality, space usage, or some other metric.
- *Design*, which is concerned with building faster and better algorithms (and programs) to solve computational problems.

Very often these two activities alternate in an algorithmic research project – a new design strategy requires analysis, which in turn suggests new design improvements, and so forth.

A third important area of algorithm studies is *models of computation*, which considers how changes in the underlying machine (or machine model) affect design and analysis. Problems in this area are also considered in a few sections of the text.

The discussion is aimed at the newcomer to experiments who has some familiarity with algorithm design and analysis, at about the level of an undergraduate course. The presentation draws on knowledge from diverse areas, including theoretical algorithmics, code tuning, computer architectures, memory hierarchies, and topics in statistics and data analysis. Since "everybody is ignorant, only on different subjects" (Will Rogers), basic concepts and definitions in these areas are introduced as needed.

1.1 Why Do Experiments?

The foundational work in algorithm design and analysis has been carried out using a *theoretical* approach, which is based on abstraction, theorem, and proof. In this framework, algorithm design means creating an algorithm in pseudocode, and algorithm analysis means finding an asymptotic bound on the dominant operation under a worst-case or average-case model.

The main benefit of this abstract approach is universality of results – no matter how skilled the programmer, or how fast the platform, the asymptotic bound on performance is guaranteed to hold. Furthermore, the asymptotic bound is the most important property determining performance at large n, which is exactly when performance matters most. Here are two stories to illustrate this point.

- Jon Bentley [7] ran a race between two algorithms to solve the maximum-sum subarray problem. The $\Theta(n^3)$ algorithm was implemented in the fastest environment he could find (tuned C code on a 533MHz Alpha 21164), and the $\Theta(n)$ algorithm ran in the slowest environment available (interpreted Basic on a 2.03MHz Radio Shack TRS-80 Model II). Despite these extreme platform differences, the crossover point where the fast asymptotic algorithm started beating the

slow algorithm occurred at only $n = 5,800$, when both programs took two minutes to run. At $n = 10,000$, the highly tuned cubic algorithm required seven days of computation while the poorly tuned linear algorithm required only 32 minutes.

- Steve Skiena [26] describes a project to test a conjecture about pyramid numbers, which are of the form $(m^3 - m)/6$, $m \geq 2$. An $O(n^{4/3} \log n)$ algorithm ran 30,000 times faster than an $O(n^2)$ algorithm at $n = 10^9$, finishing in 20 minutes instead of just over a year. Even tiny asymptotic differences become important when n is large enough.

The main drawback of the theoretical approach is lack of specificity – a pencil-and-paper algorithm is a far cry from a working program, and considerable effort may be needed to fill in the details to get from one to the other. Furthermore, asymptotic analyses require greatly simplified models of computation, which can introduce significant inaccuracies in performance predictions.

Because of these gaps between theory and experience, some prefer to use an *empirical* approach to performance analysis: implement the algorithm and measure its runtime. This approach provides specificity but lacks generality – it is notoriously difficult to translate runtime measurements taken on one platform and one set of instances into accurate time predictions for other scenarios.

Experimental algorithmics represents a third approach that treats algorithms as laboratory subjects, emphasizing control of parameters, isolation of key components, model building, and statistical analysis. This is distinct from the purely empirical approach, which studies performance in "natural settings," in a manner akin to field experiments in the natural sciences.

Instead, experimental algorithmics combines the tools of the empiricist – code and measurement – with the abstraction-based approach of the theoretician. Insights from laboratory experiments can be more precise and realistic than pure theory provides, but also more general than field experiments can produce.

This approach complements but does not replace the other two approaches to understanding algorithm performance. It holds promise for bridging the long-standing communication gap between theory and practice, by providing a common ground for theoreticians and practitioners to exchange insights and discoveries about algorithm and program performance.

Some Examples. Here are some stories illustrating what has been accomplished by applying the experimental approach to problems in algorithm analysis.

- Theoretical analysis of Dijkstra's algorithm (on a graph of n vertices and m edges) concentrates on the cost of the decrease-key operation, which could be performed m times, giving an $O(m \log n)$ or $O(m + n \log n)$ worst-case bound,

depending on data structure. But experimental analysis has shown that the worst-case bound is overly pessimistic – the number of decrease-key operations is quite small for many types of graphs that arise in practical applications, such as network routing and roadmap navigation. In many real-world situations Dijkstra's algorithm exhibits $O(n+m)$ performance. See Cherkassky et al. [8] for experimental results and theorems.

- The history of average-case analysis of internal path length (IPL) in binary search trees, under a series of t random insertions and deletions, also illustrates how experiments can guide theory. The expected IPL in a random binary tree is $O(n \log n)$. An early theorem showing that IPL does not change with t was "disproved" by experiments (the theorem was correct but did not apply to the random model). Those experiments prompted a new conjecture that IPL decreases asymptotically with t. But later experiments showed that IPL initially decreases, then increases, then levels off to $\Theta(n \log^2 n)$. A more recent conjecture of $\Theta(n^{3/2})$ cost is well supported by experiments, but as yet unproved. See Panny [22] for details.
- LaMarca and Ladner performed a series of experiments to evaluate the cache performance of fundamental algorithms and data structures [15] [16]. On the basis of their results, they developed a new analytical model of computation that captures the two-tier nature of memory costs in real computers. Reanalysis of standard algorithms under this model produces much closer predictions of performance than the standard RAM model of computation used in classic analysis.

These examples show how experimental analysis can be used to:

1. Fill in the gaps between the simplifying assumptions necessary to theory and the realities of practical experience.
2. Characterize the differences among worst-case, average-case, and typical-case performance.
3. Suggest new theorems and guide proof strategies.
4. Extend theoretical analyses to more realistic inputs and models of computation.

Similarly, the experimental approach has made important contributions to problems in algorithm design and engineering. The term *algorithm engineering* has been coined to describe a systematic process for transforming abstract algorithms into production-quality software, with an emphasis on building fast, robust, and easy-to-use code. Algorithm design, which focuses on implementation and tuning strategies for specific algorithms and data structures (see Chapter 4 for details), is just one part of the larger algorithm engineering process, which is also concerned with requirements specification, interfaces, scalability, correctness, and so forth.

Here are a few examples showing how experiments have played a central role in both algorithm design and algorithm engineering.

- The 2006 *9th DIMACS Implementation Challenge–Shortest Paths* workshop contained presentations of several projects to speed up single-pair shortest-path (SPSP) algorithms. In one paper from the workshop, Sanders and Shultes [24] describe experiments to engineer an algorithm to run on roadmap graphs used in global positioning system (GPS) Routing applications: the Western Europe and the United States maps contain ($n = 18$ million, $m = 42.5$ million) and ($n = 23.9$ million, $m = 58.3$ million) nodes and edges, respectively. They estimate that their tuned implementation of Dijkstra's algorithm runs more than a million times faster on an average query than the best known implementation for general graphs.
- Bader et al. [2] describe efforts to speed up algorithms for computing optimal phylogenies, a problem in computational biology. The breakpoint phylogeny heuristic uses an exhaustive search approach to generate and evaluate candidate solutions. Exact evaluation of *each* candidate requires a solution to the traveling salesman problem, so that the worst-case cost is $O(2n!!)$ [*sic*–double factorial] to solve a problem with n genomes. Their engineering efforts, which exploited parallel processing as well as algorithm and code tuning, led to speedups by factors as large as 1 million on problems containing 10 to 12 genomes.
- Speedups by much smaller factors than a million can of course be critically important on frequently used code. Yaroslavskiy et al. [27] describe a project to implement the Arrays.sort() method for JDK 7, to achieve fast performance when many duplicate array elements are present. (Duplicate array elements represent a worst-case scenario for many implementations of quicksort.) Their tests of variations on quicksort yielded performance differences ranging from 20 percent faster than a standard implementation (on arrays with no duplicates), to more than 15 times faster (on arrays containing identical elements).
- Sometimes the engineering challenge is simply to demonstrate a working implementation of a complex algorithm. Navarro [21] describes an effort to implement the LZ-Index, a data structure that supports indexing and fast lookup in compressed data. Navarro shows how experiments were used to guide choices made in the implementation process and to compare the finished product to competing strategies. This project is continued in [11], which describes several tuned implementations assembled in a repository that is available for public use.

These examples illustrate the ways in which experiments have played key roles in developing new insights about algorithm design and analysis. Many more examples can be found throughout this text and in references cited in the Chapter Notes.

1.2 Key Concepts

This section introduces some basic concepts that provide a framework for the larger discussion throughout the book.

A Scale of Instantiation

We make no qualitative distinction here between "algorithms" and "programs." Rather, we consider algorithms and programs to represent two points on a *scale of instantiation*, according to how much specificity is in their descriptions. Here are some more recognizable points on this scale.

- At the most abstract end are *metaheuristics* and *algorithm paradigms*, which describe generic algorithmic structures that are not tied to particular problem domains. For example, Dijkstra's algorithm is a member of the greedy paradigm, and tabu search is a metaheuristic that can be applied to many problems.
- The *algorithm* is an abstract description of a process for solving an abstract problem. At this level we might see Dijkstra's algorithm written in pseudocode. The pseudocode description may be more or less instantiated according to how much detail is given about data structure implementation.
- The *source program* is a version of the algorithm implemented in a particular high-level language. Specificity is introduced by language and coding style, but the source code remains platform-independent. Here we might see Dijkstra's algorithm implemented in C++ using an STL priority queue.
- The *object code* is the result of compiling a source program. This version of the algorithm is written in machine code and specific to a family of architectures.
- The *process* is a program actively running on a particular machine at a particular moment in time. Performance at this level may be affected by properties such as system load, the size and shape of the memory hierarchy, and process scheduler policy.

Interesting algorithmic experiments can take place at any point on the instantiation scale. We make a conceptual distinction between the *experimental subject*, which is instantiated somewhere on the scale, and the *test program*, which is implemented to study the performance of the subject.

For example, what does it mean to measure an algorithm's time performance? Time performance could be defined as a count of the dominant cost, as identified by theory: this is an abstract property that is universal across programming languages, programmers, and platforms. It could be a count of instruction executions, which is a property of object code. Or it could be a measurement of elapsed time, which depends on the code as well as on the platform. There is one test program, but the experimenter can choose to measure any of these properties, according to the level of instantiation adopted in the experiment.

In many cases the test program may be exactly the subject of interest – but it need not be. By separating the two roles that a program may play, both as test subject and as testing apparatus, we gain clarity about experimental goals and procedures. Sometimes this conceptual separation leads to better experiments, in the sense that a test program can generate better-quality data more efficiently than a conventional implementation could produce (see Chapter 6 for details).

This observation prompts the first of many guidelines presented throughout the book. Guidelines are meant to serve as short reminders about best practice in experimental methodology. A list of guidelines appears in the Chapter Notes at the end of each chapter.

Guideline 1.1 *The "algorithm" and the "program" are just two points on a scale between abstract and instantiated representations of a given computational process.*

The Algorithm Design Hierarchy

Figure 1.1 shows the *algorithm design hierarchy*, which comprises six levels that represent broad strategies for improving algorithm performance. This hierarchical approach to algorithm design was first articulated by Reddy and Newell [23] and further developed by Bentley [6], [7]. The list in Figure 1.1 generally follows Bentley's development, except two layers–algorithm design and code tuning – are now split into three – algorithm design, algorithm tuning, and code tuning. The distinction is explained further in Chapter 4.

The levels in this hierarchy are organized roughly in the order in which decisions must be made in an algorithm engineering project. You have to design the algorithm before you implement it, and you cannot tune code before the implementation exists. On the other hand, algorithm engineering is not really a linear process – a new insight, or a roadblock, may be discovered at any level that makes it necessary to start over at a higher level.

Chapter 4 surveys tuning strategies that lie at the middle two levels of this hierarchy – algorithm tuning and code tuning. Although concerns at the other levels are outside the scope of this book, do not make the mistake of assuming that they are not important to performance. The stories in Section 1.1 about Bentley's race and Skiena's pyramid numbers show how important it is to get the asymptotics right in the first place.

In fact, the greatest feats of algorithm engineering result from combining design strategies from different levels: a 10-fold speedup from rearranging file structures at the system level, a 100-fold speedup from algorithm tuning, a 5-fold speedup from code tuning, and a 2-fold improvement from using an optimizing compiler, will combine *multiplicatively* to produce a 10,000-fold improvement in overall running time. Here are two stories that illustrate this effect.

- **System structure.** Decompose the software into modules that interact efficiently. Check whether the target runtime environment provides sufficient support for the modules. Decide whether the final product will run on a concurrent or sequential platform.
- **Algorithm and data structure design.** Specify the exact problem that is to be solved in each module. Choose appropriate problem representations. Select or design algorithms and data structures that are asymptotically efficient.
- **Implementation and algorithm tuning.** Implement the algorithm, or perhaps build a family of implementations. Tune the algorithm by considering high-level structures relating to the algorithm paradigm, input classes, and cost models.
- **Code tuning.** Consider low-level code-specific properties such as loops and procedure calls. Apply a systematic process to transform the program into a functionally equivalent program that runs faster.
- **System software.** Tune the runtime environment for best performance, for example by turning on compiler optimizers and adjusting memory allocations.
- **Platform and hardware.** Shift to a faster CPU and/or add coprocessors.

Figure 1.1. The algorithm design hierarchy. The levels in this hierarchy represent broad strategies for speeding up algorithms and programs.

Cracking RSA-129. Perhaps the most impressive algorithm engineering achievement on record is Atkins et al.'s [1] implementation of a program to factor a 129-digit number and solve an early RSA Encryption Challenge. Reasonable estimates at the time of the challenge were that the computation would take 4 quadrillion years. Instead, 17 years after the challenge was announced, the code was cracked in an eight-month computation: this represents a 6 quadrillion–fold speedup over the estimated computation time.

The authors' description of their algorithm design process gives the following insights about contributions at various levels of the algorithm design hierarchy.

- The task was carried out in three phases: an eight-month distributed computation (1600 platforms); then a 45-hour parallel computation (16,000 CPUs); then a few hours of computation on a sequential machine. Assuming optimal speedups due to concurrency, the first two phases would have required a total of 1149.2

years on a sequential machine. Thus *concurrency* contributed at most a 1150-fold speedup.
- Significant *system* design problems had to be solved before the computation could take place. For example, the distributed computation required task modules that could fit into main memory of all platforms offered by volunteers. Also, data compression was needed to overcome a critical memory shortage late in the computation.
- According to Moore's Law (which states that computer speeds typically double every 18 months), faster hardware alone could have contributed a 2000-fold speedup during the 17 years between challenge and solution. But in fact the original estimate took this effect into account. Thus no speedup over the estimate can be attributed to *hardware*.
- The authors describe *code tuning* improvements that contributed a factor of 2 speedup (there may be more that they did not report).
- Divide 6 quadrillion by $2300 = 1150 \times 2$: the remaining 2.6–trillion-fold speedup is due to improvements at the *algorithm design* level.

Finding Phylogenies Faster. In a similar vein, Bader et al. [2], [19] describe their engineering efforts to speed up the breakpoint phylogeny algorithm described briefly in Section 1.1.

- Since the generation of independent candidate solutions can easily be parallelized, the authors implemented their code for a 512-processor Alliance Cluster platform. This decision at the *systems* level to adopt a parallel solution produced an optimal 512-fold speedup over a comparable single-processor version.
- *Algorithm design* and *algorithm tuning* led to speedups by factors around 100; redesign of the data structures yielded another factor of 10. The cumulative speedup is 1000. The authors applied cache-aware tuning techniques to obtain a smaller memory footprint (from 60MB down to 1.8MB) and to improve cache locality. They remark that the new implementation runs almost entirely in cache for their test data sets.
- Using profiling to identify timing bottlenecks in a critical subroutine, they applied *code tuning* to obtain 6- to 10-fold speedups. The cumulative speedup from algorithm and code tuning was between 300 and 50,000, depending on inputs.

This combination of design improvements resulted in cumulative speedups by factors up to 1 million on some inputs.

Guideline 1.2 *When the code needs to be faster, consider all levels of the algorithm design hierarchy.*

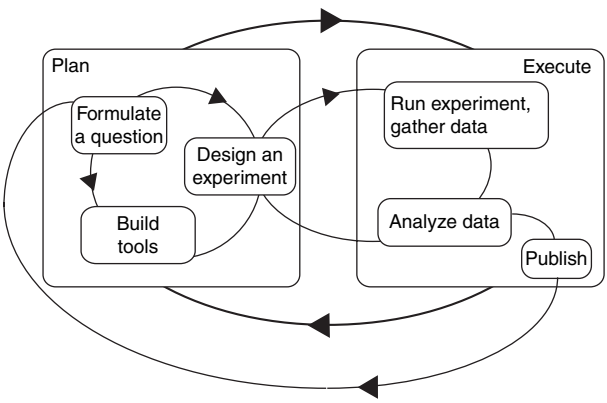

Figure 1.2. The experimental process. Experiments are carried out in cycles within cycles. Planning experiments alternates with executing them. In the planning stage, formulating questions alternates with building testing tools and designing experiments. In the execution stage, conducting experiments alternates with analyzing data. Individual steps may be carried out in different orders and are sometimes skipped.

The Experimental Process

The major steps in the experimental process are defined in the following list and illustrated in Figure 1.2. Despite the numerical list structure, the process is not sequential, but rather loosely cyclical: planning experiments alternates with conducting experiments; the three steps in the planning stage may be carried out in any order; and steps may be skipped occasionally.

1. Plan the experiment:
   a. Formulate a question.
   b. Assemble or build the test environment. The test environment comprises the test program, input instances and instance generators, measurement tools and packages, and data analysis software. These components might be readily available or might require considerable development time in their own right.
   c. Design an experiment to address the question at hand. Specify, for example, what properties are to be measured, what input categories are applied, which input sizes are measured, how many random trials are run, and so forth.
2. Execute the experiment:
   a. Run the tests and collect the data.
   b. Apply data analysis to glean information and insight. If the question was not answered, go back to the planning stage and try again.
   c. (For academic experimenters.) Publish the results. Ideally, your publication prompts several new questions, which start the process over again.

A single round of planning and experimenting may be part of a larger process in a project developed for purposes of design or analysis, or both. The nature