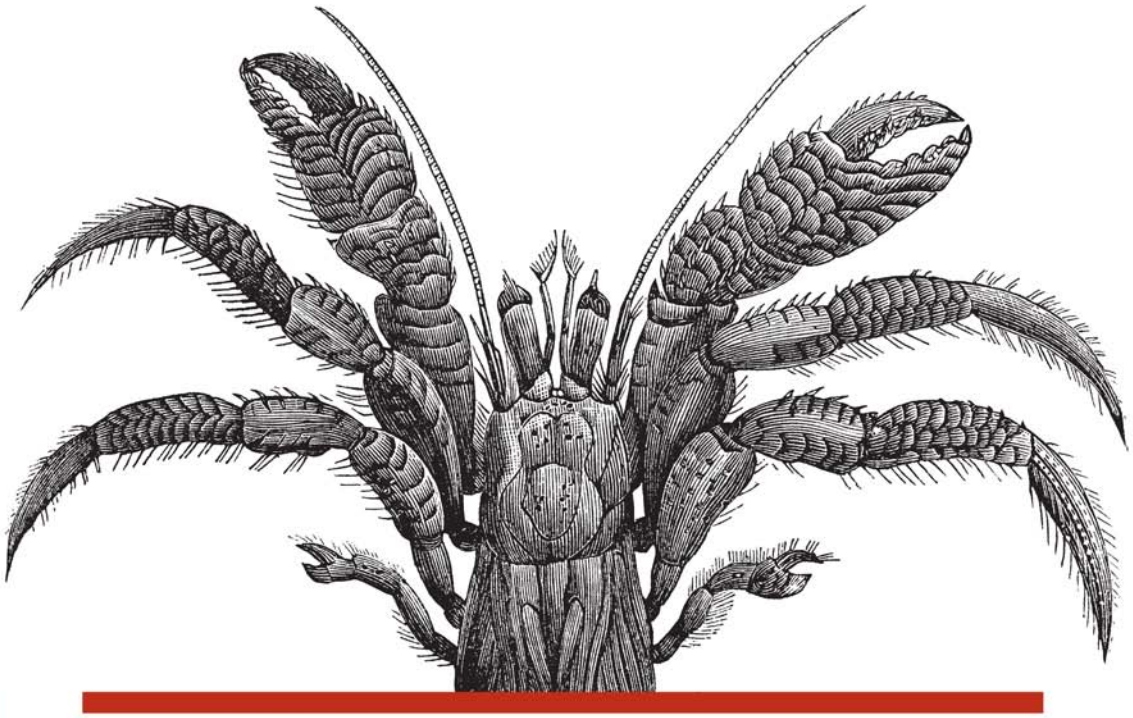

# ALGORITHMS IN A NUTSHELL

*A Desktop Quick Reference*

*George T. Heineman,*
*Gary Pollice & Stanley Selkow*

O'REILLY®

# ALGORITHMS IN A NUTSHELL

Creating robust software requires the use of efficient algorithms, but programmers seldom think about them until a problem occurs. *Algorithms in a Nutshell* describes existing algorithms for solving a variety of problems, and helps you select and implement the right algorithm for your needs—with just enough math so you can understand and analyze algorithm performance.

Focusing on application rather than theory, *Algorithms in a Nutshell* provides efficient code solutions in several programming languages that you can easily adapt to a specific project. With this book, you will:

- Solve specific coding problems or improve on the performance of existing solutions
- Quickly locate algorithms that relate to the problems you want to solve, and determine why a particular algorithm is the right one to use
- Explore algorithmic solutions in C, C++, Java, and Ruby, with implementation tips
- Learn the expected performance of an algorithm, as well as the conditions it needs to perform at its best
- Discover the impact that similar design decisions have on different algorithms
- Learn advanced data structures to improve the efficiency of algorithms

With *Algorithms in a Nutshell*, you'll learn how to improve the performance of algorithms that are key to the success of your software applications.

*"The authors have done a fantastic job of distilling reams of arcane literature into an indispensable guide that strikes the perfect balance between theory and implementation. Grokking algorithms just became a whole lot easier."*

—Matthew Russell, Director of Advanced Technology, Digital Reasoning Systems; author of *Dojo: The Definitive Guide* (O'Reilly)

**George T. Heineman**, **Gary Pollice**, and **Stanley Selkow** are professors in the Computer Science Department at Worcester Polytechnic Institute. George is co-editor of *Component-Based Software Engineering: Putting the Pieces Together* (Addison-Wesley). Gary is a coauthor of *Head First Object-Oriented Analysis and Design* (O'Reilly).

O'REILLY®
www.oreilly.com

US $49.99 CAN $49.99
ISBN: 978-0-596-51624-6

# ALGORITHMS

## IN A NUTSHELL

## Other resources from O'Reilly

**Related titles**

- Head First Object-Oriented Design and Analysis
- Head First Software Development
- Mastering Algorithms with C
- Programming Collective Intelligence

**oreilly.com**

*oreilly.com* is more than a complete catalog of O'Reilly books. You'll also find links to news, events, articles, weblogs, sample chapters, and code examples.


*oreillynet.com* is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

**Conferences**

O'Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.


Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today for free.

# ALGORITHMS

## IN A NUTSHELL

*George T. Heineman, Gary Pollice, and*
*Stanley Selkow*

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

**Algorithms in a Nutshell**
by George T. Heineman, Gary Pollice, and Stanley Selkow



Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

**Editor:** Mary Treseler
**Production Editor:** Rachel Monaghan
**Production Services:** Newgen Publishing and Data Services
**Copyeditor:** Genevieve d'Entremont
**Proofreader:** Rachel Monaghan
**Indexer:** John Bickelhaupt
**Cover Designer:** Karen Montgomery
**Interior Designer:** David Futato
**Illustrator:** Robert Romano

**Printing History:**

October 2008: First Edition.




This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-51624-6
[M]

# Table of Contents

# Preface

As Trinity states in the movie *The Matrix*:

> *It's the question that drives us, Neo. It's the question that brought you here.*
>
> *You know the question, just as I did.*

As authors of this book, we answer the question that has led you here:

> *Can I use algorithm* X *to solve my problem? If so, how do I implement it?*

You likely do not need to understand the reasons why an algorithm is correct—if you do, turn to other sources, such as the 1,180-page bible on algorithms, *Introduction to Algorithms,* Second Edition, by Thomas H. Cormen et al. (2001). There you will find lemmas, theorems, and proofs; you will find exercises and step-by-step examples showing the algorithms as they perform. Perhaps surprisingly, however, you will not find any real code, only fragments of "pseudocode," the device used by countless educational textbooks to present a high-level description of algorithms. These educational textbooks are important within the classroom, yet they fail the software practitioner because they assume it will be straightforward to develop real code from pseudocode fragments.

We intend this book to be used frequently by experienced programmers looking for appropriate solutions to their problems. Here you will find solutions to the problems you must overcome as a programmer every day. You will learn what decisions lead to an improved performance of key algorithms that are essential for the success of your software applications. You will find real code that can be adapted to your needs and solution methods that you can learn.

All algorithms are fully implemented with test suites that validate the correct implementation of the algorithms. The code is fully documented and available as a code repository addendum to this book. We rigorously followed a set of principles as we designed, implemented, and wrote this book. If these principles are meaningful to you, then you will find this book useful.

## Principle: Use Real Code, Not Pseudocode

What is a practitioner to do with Figure P-1's description of the FORD-FULKERSON algorithm for computing maximum network flow?

**Ford-Fulkerson Algorithm:**
**Input** Graph $G$ with flow capacity $c$, a source node $s$, and a sink node $t$
**Output** A flow $f$ from $s$ to $t$ which is a maximum

1. $f(u,v) \leftarrow 0$ for all edges $(u,v)$
2. **while** (there is a path $p$ from $s$ to $t$ in $G_f$ such that $c_f(u,v)>0$ for all edges $(u,v)\in p$) **do**
3. Find $c_f(p) = \min \{ c_f(u,v) \mid (u,v)\in p \}$
4. **foreach** edge $c_f(u,v)\in p$ **do**
5. $f(u,v) \leftarrow f(u,v) + c_f(p)$ // Send flow along the path
6. $f(v,u) \leftarrow f(v,u) - c_f(p)$ // The flow might be "returned" later

**end**

*Figure P-1. Example of pseudocode commonly found in textbooks*

The algorithm description in this figure comes from Wikipedia (*http://en.wikipedia.org/wiki/Ford_Fulkerson*), and it is nearly identical to the pseudocode found in (Cormen et al., 2001). It is simply unreasonable to expect a software practitioner to produce working code from the description of FORD-FULKERSON shown here! Turn to Chapter 8 to see our code listing by comparison. We use only documented, well-designed code to describe the algorithms. Use the code we provide as-is, or include its logic in your own programming language and software system.

Some algorithm textbooks do have full real-code solutions in C or Java. Often the purpose of these textbooks is to either teach the language to a beginner or to explain how to implement abstract data types. Additionally, to include code listings within the narrow confines of a textbook page, authors routinely omit documentation and error handling, or use shortcuts never used in practice. We believe programmers can learn much from documented, well-designed code, which is why we dedicated so much effort to develop actual solutions for our algorithms.

## Principle: Separate the Algorithm from the Problem Being Solved

It is hard to show the implementation for an algorithm "in the general sense" without also involving details of the specific solution. We are critical of books that show a full implementation of an algorithm yet allow the details of the specific problem to become so intertwined with the code for the generic problem that it is hard to identify the structure of the original algorithm. Even worse, many available implementations rely on sets of arrays for storing information in a way that is "simpler" to code but harder to understand. Too often, the reader will understand the concept from the supplementary text but be unable to implement it!

In our approach, we design each implementation to separate the generic algorithm from the specific problem. In Chapter 7, for example, when we describe the A*SEARCH algorithm, we use an example such as the 8-puzzle (a sliding tile puzzle with tiles numbered 1–8 in a three-by-three grid). The implementation of A*SEARCH depends only on a set of well-defined interfaces. The details of the specific 8-puzzle problem are encapsulated cleanly within classes that implement these interfaces.

We use numerous programming languages in this book and follow a strict design methodology to ensure that the code is readable and the solutions are efficient. Because of our software engineering background, it was second nature to design clear interfaces between the general algorithms and the domain-specific solutions. Coding in this way produces software that is easy to test, maintain, and expand to solve the problems at hand. One added benefit is that the modern audience can more easily read and understand the resulting descriptions of the algorithms. For select algorithms, we show how to convert the readable and efficient code that we produced into highly optimized (though less readable) code with improved performance. After all, the only time that optimization should be done is when the problem has been solved and the client demands faster code. Even then it is worth listening to C. A. R. Hoare, who stated, "Premature optimization is the root of all evil."

## Principle: Introduce Just Enough Mathematics

Many treatments of algorithms focus nearly exclusively on proving the correctness of the algorithm and explaining only at a high level its details. Our focus is always on showing how the algorithm is to be implemented in practice. To this end, we only introduce the mathematics needed to understand the data structures and the control flow of the solutions.

For example, one needs to understand the properties of sets and binary trees for many algorithms. At the same time, however, there is no need to include a proof by induction on the height of a binary tree to explain how a red-black binary tree is balanced; read Chapter 13 in (Cormen et al., 2001) if you want those details. We explain the results as needed, and refer the reader to other sources to understand how to prove these results mathematically.

In this book you will learn the key terms and analytic techniques to differentiate algorithm behavior based on the data structures used and the desired functionality.

## Principle: Support Mathematical Analysis Empirically

We mathematically analyze the performance of each algorithm in this book to help programmers understand the conditions under which each algorithm performs at its best. We provide live code examples, and in the accompanying code repository there are numerous JUnit (*http://sourceforge.net/projects/junit*) test cases to document the proper implementation of each algorithm. We generate benchmark performance data to provide empirical evidence regarding the performance of each algorithm.

We classify each algorithm into a specific performance family and provide benchmark data showing the execution performance to support the analysis. We avoid algorithms that are interesting only to the mathematical algorithmic designer trying to prove that an approach performs better at the expense of being impossible to implement. We execute our algorithms on a variety of programming platforms to demonstrate that the design of the algorithm—not the underlying platform—is the driving factor in efficiency.

The appendix contains the full details of our approach toward benchmarking, and can be used to independently validate the performance results we describe in this book. The advice we give you is common in the open source community: "Your mileage may vary." Although you won't be able to duplicate our results exactly, you will be able to verify the trends that we document, and we encourage you to use the same empirical approach when deciding upon algorithms for your own use.

## Audience

If you were trapped on a desert island and could have only one algorithms book, we recommend the complete box set of *The Art of Computer Programming*, Volumes 1–3, by Donald Knuth (1998). Knuth describes numerous data structures and algorithms and provides exquisite treatment and analysis. Complete with historical footnotes and exercises, these books could keep a programmer active and content for decades. It would certainly be challenging, however, to put directly into practice the ideas from Knuth's book.

But you are not trapped on a desert island, are you? No, you have sluggish code that must be improved by Friday and you need to understand how to do it!

We intend our book to be your primary reference when you are faced with an algorithmic question and need to either (a) solve a particular problem, or (b) improve on the performance of an existing solution. We cover a range of existing algorithms for solving a large number of problems and adhere to the following principles:

- When describing each algorithm, we use a stylized pattern to properly frame each discussion and explain the essential points of the algorithm. By using patterns, we create a readable book whose consistent presentation shows the impact that similar design decisions have on different algorithms.
- We use a variety of languages to describe the algorithms in the book (including C, C++, Java, and Ruby). In doing so, we make concrete the discussion on algorithms and speak using languages that you are already familiar with.
- We describe the expected performance of each algorithm and empirically provide evidence that supports these claims. Whether you trust in mathematics or in demonstrable execution times, you will be persuaded.

We intend this book to be most useful to software practitioners, programmers, and designers. To meet your objectives, you need access to a quality resource that explains real solutions to real algorithms that you need to solve real problems.

You already know how to program in a variety of programming languages. You know about the essential computer science data structures, such as arrays, linked lists, stacks, queues, hash tables, binary trees, and undirected and directed graphs. You don't need to implement these data structures, since they are typically provided by code libraries.

We expect that you will use this book to learn about tried and tested solutions to solve problems efficiently. You will learn some advanced data structures and some novel ways to apply standard data structures to improve the efficiency of algorithms. Your problem-solving abilities will improve when you see the key decisions for each algorithm that make for efficient solutions.

## Contents of This Book

This book is divided into three parts. Part I (Chapters 1–3) provides the mathematical introduction to algorithms necessary to properly understand the descriptions used in this book. We also describe the pattern-based style used throughout in the presentation of each algorithm. This style is carefully designed to ensure consistency, as well as to highlight the essential aspects of each algorithm. Part II contains a series of chapters (4–9), each consisting of a set of related algorithms. The individual sections of these chapters are self-contained descriptions of the algorithms.

Part III (Chapters 10 and 11) provides resources that interested readers can use to pursue these topics further. A chapter on approaches to take when "all else fails" provides helpful hints on solving problems when there is (as yet) no immediate efficient solution. We close with a discussion of important areas of study that we omitted from Part II simply because they were too advanced, too niche-oriented, or too new to have proven themselves. In Part IV, we include a benchmarking appendix that describes the approach used throughout this book to generate empirical data that supports the mathematical analysis used in each chapter. Such benchmarking is standard in the industry yet has been noticeably lacking in textbooks describing algorithms.

## Conventions Used in This Book

The following typographical conventions are used in this book:

`Code`
: `All code examples appear in this typecase.`
`This code is replicated directly from the code repository and reflects real code.`

*Italic*
: Indicates key terms used to describe algorithms and data structures. Also used when referring to variables within a pseudocode description of an example.

`Constant width`
: Indicates the name of actual software elements within an implementation, such as a Java class, the name of an array within a C implementation, and constants such as `true` or `false`.

SMALL CAPS
: Indicates the name of an algorithm.

We cite numerous books, articles, and websites throughout the book. These citations appear in text using parentheses, such as (Cormen et al., 2001), and each chapter closes with a listing of references used within that chapter. When the reference citation immediately follows the name of the author in the text, we do not duplicate the name in the reference. Thus, we refer to the *Art of Computer Programming* books by Donald Knuth (1998) by just including the year in parentheses.

All URLs used in the book were verified as of August 2008 and we tried to use only URLs that should be around for some time. We include small URLs, such as *http://www.oreilly.com*, directly within the text; otherwise, they appear in footnotes and within the references at the end of a chapter.

## Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Algorithms in a Nutshell* by George T. Heineman, Gary Pollice, and Stanley Selkow. Copyright 2009 George Heineman, Gary Pollice, and Stanley Selkow, 978-0-596-51624-6."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

## Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

*http://www.oreilly.com/catalog/9780596516246*

To comment or ask technical questions about this book, send email to:

*bookquestions@oreilly.com*

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

*http://www.oreilly.com*

## Safari® Books Online


When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

## Acknowledgments

We would like to thank the book reviewers for their attention to detail and suggestions, which improved the presentation and removed defects from earlier drafts: Alan Davidson, Scot Drysdale, Krzysztof Duleba, Gene Hughes, Murali Mani, Jeffrey Yasskin, and Daniel Yoo.

George Heineman would like to thank those who helped instill in him a passion for algorithms, including Professors Scot Drysdale (Dartmouth College) and Zvi Galil (Columbia University). As always, George thanks his wife, Jennifer, and his children, Nicholas (who always wanted to know what "notes" Daddy was working on) and Alexander (who was born as we prepared the final draft of the book).

Gary Pollice would like to thank his wife Vikki for 40 great years. He also wants to thank the WPI computer science department for a great environment and a great job.

Stanley Selkow would like to thank his wife, Deb. This book was another step on their long path together.

## References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Second Edition. McGraw-Hill, 2001.

Knuth, Donald E., *The Art of Computer Programming*, Volumes 1–3, Boxed Set Second Edition. Addison-Wesley Professional, 1998.

# 1 Algorithms Matter

Algorithms matter! Knowing which algorithm to apply under which set of circumstances can make a big difference in the software you produce. If you don't believe us, just read the following story about how Gary turned failure into success with a little analysis and choosing the right algorithm for the job.*

Once upon a time, Gary worked at a company with a lot of brilliant software developers. Like most organizations with a lot of bright people, there were many great ideas and people to implement them in the software products. One such person was Graham, who had been with the company from its inception. Graham came up with an idea on how to find out whether a program had any memory leaks—a common problem with C and C++ programs at the time. If a program ran long enough and had memory leaks, it would crash because it would run out of memory. Anyone who has programmed in a language that doesn't support automatic memory management and garbage collection knows this problem well.

Graham decided to build a small library that *wrapped* the operating system's memory allocation and deallocation routines, `malloc()` and `free()`, with his own functions. Graham's functions recorded each memory allocation and deallocation in a data structure that could be queried when the program finished. The wrapper functions recorded the information and called the real operating system functions to perform the actual memory management. It took just a few hours for Graham to implement the solution and, *voilà*, it worked! There was just one problem: the program ran so slowly when it was instrumented with Graham's libraries that no one was willing to use it. We're talking *really* slow here. You could start up a program, go have a cup of coffee—or maybe a pot of coffee—come back, and the program would still be crawling along. This was clearly unacceptable.

* The names of participants and organizations, except the authors, have been changed to protect the innocent and avoid any embarrassment—or lawsuits. :-)

Now Graham was really smart when it came to understanding operating systems and how their internals work. He was an excellent programmer who could write more working code in an hour than most programmers could write in a day. He had studied algorithms, data structures, and all of the standard topics in college, so why did the code execute so much slower with the wrappers inserted? In this case, it was a problem of knowing enough to make the program work, but not thinking through the details to make it work *quickly*. Like many creative people, Graham was already thinking about his next program and didn't want to go back to his memory leak program to find out what was wrong. So, he asked Gary to take a look at it and see whether he could fix it. Gary was more of a compiler and software engineering type of guy and seemed to be pretty good at honing code to make it release-worthy.

Gary thought he'd talk to Graham about the program before he started digging into the code. That way, he might better understand how Graham structured his solution and why he chose particular implementation options.

Before proceeding, think about what you might ask Graham. See whether you would have obtained the information that Gary did in the following section.

## Understand the Problem

A good way to solve problems is to start with the big picture: understand the problem, identify potential causes, and then dig into the details. If you decide to try to solve the problem because you *think* you know the cause, you may solve the wrong problem, or you might not explore other—possibly better—answers. The first thing Gary did was ask Graham to describe the problem and his solution.

Graham said that he wanted to determine whether a program had any memory leaks. He thought the best way to find out would be to keep a record of all memory that was allocated by the program, whether it was freed before the program ended, and a record of where the allocation was requested in the user's program. His solution required him to build a small library with three functions:

`malloc()`
: A wrapper around the operating system's memory allocation function

`free()`
: A wrapper around the operating system's memory deallocation function

`exit()`
: A wrapper around the operating system's function called when a program exits

This custom library would be linked with the program under test in such a way that the customized functions would be called instead of the operating system's functions. The custom `malloc()` and `free()` functions would keep track of each allocation and deallocation. When the program under test finished, there would be no memory leak if every allocation was subsequently deallocated. If there were any leaks, the information kept by Graham's routines would allow the programmer to find the code that caused them. When the `exit()` function was

called, the custom library routine would display its results before actually exiting. Graham sketched out what his solution looked like, as shown in Figure 1-1.

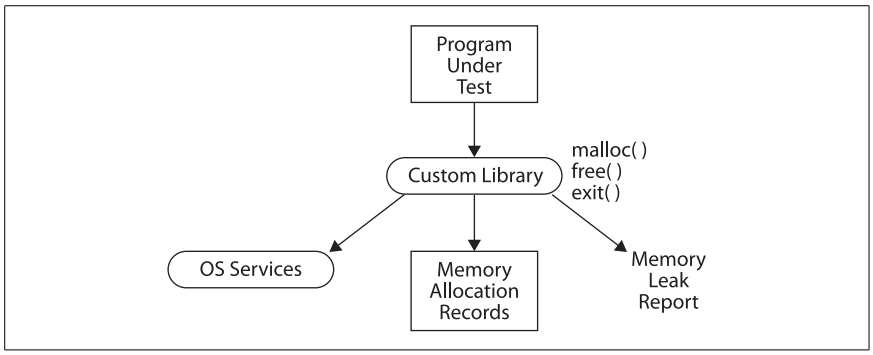


*Figure 1-1. Graham's solution*

The description seemed clear enough. Unless Graham was doing something terribly wrong in his code to wrap the operating system functions, it was hard to imagine that there was a performance problem in the wrapper code. If there were, then all programs would be proportionately slow. Gary asked whether there was a difference in the performance of the programs Graham had tested. Graham explained that the running profile seemed to be that small programs—those that did relatively little—all ran in acceptable time, regardless of whether they had memory leaks. However, programs that did a lot of processing and had memory leaks ran disproportionately slow.

## Experiment if Necessary

Before going any further, Gary wanted to get a better understanding of the running profile of programs. He and Graham sat down and wrote some short programs to see how they ran with Graham's custom library linked in. Perhaps they could get a better understanding of the conditions that caused the problem to arise.

What type of experiments would you run? What would your program(s) look like?

The first test program Gary and Graham wrote (ProgramA) is shown in Example 1-1.

*Example 1-1. ProgramA code*

```
int main(int argc, char **argv) {
  int i = 0;
  for (i = 0; i < 1000000; i++) {
    malloc(32);
  }
  exit (0);
}
```

They ran the program and waited for the results. It took several minutes to finish. Although computers were slower back then, this was clearly unacceptable. When this program finished, there were 32 MB of memory leaks. How would the program run if all of the memory allocations were deallocated? They made a simple modification to create ProgramB, shown in Example 1-2.

*Example 1-2. ProgramB code*

```
int main(int argc, char **argv) {
  int i = 0;
  for (i = 0; i < 1000000; i++) {
    void *x = malloc(32);
    free(x);
  }
  exit (0);
}
```

When they compiled and ran ProgramB, it completed in a few seconds. Graham was convinced that the problem was related to the number of memory allocations open when the program ended, but couldn't figure out where the problem occurred. He had searched through his code for several hours and was unable to find any problems. Gary wasn't as convinced as Graham that the problem was the number of memory leaks. He suggested one more experiment and made another modification to the program, shown as ProgramC in Example 1-3, in which the deallocations were grouped together at the end of the program.

*Example 1-3. ProgramC code*

```
int main(int argc, char **argv) {
  int i = 0;
  void *addrs[1000000];
  for (i = 0; i < 1000000; i++) {
    addrs[i] = malloc(32);
  }
  for (i = 0; i < 1000000; i++) {
    free(addrs[i]);
  }
  exit (0);
}
```

This program crawled along even slower than the first program! This example invalidated the theory that the number of memory leaks affected the performance of Graham's program. However, the example gave Gary an insight that led to the real problem.

It wasn't the number of memory allocations open at the end of the program that affected performance; it was the maximum number of them that were open at any single time. If memory leaks were not the only factor affecting performance, then there had to be something about the way Graham maintained the information used to determine whether there were leaks. In ProgramB, there was never more than one 32-byte chunk of memory allocated at any point during the program's execution. The first and third programs had one million open allocations.