Introductory Programming Tips

Paul Gettings Derrick Hasterok Marshall Bartlett

January 26, 2005

1 Introduction

This is a collection of hard-won, probably useful, freely-given tips on how to program for scientific problem-solving. These tips and ideas may (or may not) scale to applications, databases, or other aspects of software engineering. Software engineering, more commonly called programming, is an art more than a science. There is no well-defined, rigorously-proved, method for solving any programming problem. Rather, programming is a creative application of a toolset to a given problem.

With experience, a programmer's toolset increases in scope and complexity, allowing more difficult problems to be solved with faster algorithms in less time. The initial toolset is generally acquired in an introductory programming class, which typically limits itself to the nuts-and-bolts syntax of a given language. A small hint of the underlying art is often introduced, although not always clearly, in such a class.

This document does not seek to address any of the nuts-and-bolts aspects of programming. These change with language and programming style, and are very specific and time-sensitive; the language of today is forgotten tomorrow. Instead, the tips presented here are a higher-level abstraction of what and why, rather than how, to program.

1.1 Target Audience

These tips are directed at new programmers, not experienced ones. Experienced programmers will have already learned all these tips (probably the hard way, as the authors did). The hope is to allow new programmers to gain the benefit of experience, without the otherwise-necessary time investment of making all the mistakes that would lead to these tips.

Like any such set of tips, these ideas are not hard rules that must be obeyed; discard, modify, and refine as necessary. The goal is a working, maintainable, program that solves the problem posed. Any path to that goal is acceptable, but some may be faster than others. Here are a few tips to speed the trip.

2 How to build algorithms

Before beginning to write a program to solve a given problem, the first step is to determine what sort of algorithm will be used to solve the problem. This, of course, should depend on the type of problem being solved (computation, sorting, data formatting, etc.), the available computing power (disk space, memory, CPU, etc.), and the programming environments/languages/toolkits available.

The tips in this section are designed to help guide the choice of algorithm before actual coding starts. Pick a better algorithm at the beginning, and the rest is much easier.

2.1 Problems are rarely unique

Most scientific problems fall into a fairly small class of very general problems that have been identified and solved for many years. Therefore, there is (very) rarely the need to design an algorithm from scratch. This is a good thing, since designing an efficient, implementable, algorithm is a lot of work.

Since the general problems have been solved many times before, there is always a selection of algorithms and implementations available to solve a particular problem. The first trick is to extract the general problem out of the specific one. Once the general problem class is determined, then the literature will provide various algorithms; choose the fastest/best/cheapest alternative that will solve the special case that is a particular problem.

Finding the general problem from the specific is not trivial, and may require some very lateral thinking. For example, an algorithm to fill all the entries of a geophysical grid inside an arbitrary polygon appears hard, until one realizes this is identical to the flood-fill problem in graphics, which is well solved. This type of solution comes from abstracting the particular problem space (geophysical data grid) to a more general level (grid of values), and then comparing to a construct in computer science (video display of pixels). A list of possible abstraction techniques is infinite, and not addressed here; if stuck, explain the problem to someone else unfamiliar with it, and see what they can find.

Just as there is skill required to read and understand geoscience papers, there is a base level of knowledge required to understand computer science papers. This level of knowledge is not hard to acquire, and very valuable when looking for solutions to general programming problems. Web resources, friends in computer science, and other geoscientists with computer expertise are places to start acquiring the necessary CS background.

Never forget that computer science works on all computable problems, and all scientific problems fall into some realm of computer science. Some are in very strange places, far from the study of numerical algorithms.

2.2 Computers count and add well

Computers are powerful adding machines equipped with some memory. However, they are terrible at symbolic manipulation and pattern recognition. Hence, algorithms should emphasize simple arithmetic rather than abstract mathematics. This is the opposite of an algorithm designed for theoretical mathematics. Algorithms for programming should use the memory and fast computation speed to replace symbolic manipulation.

2.3 Bookkeeping is hard

Most scientific problems solved on computers are more bookkeeping than computation; typically, a scientific problem boils down to performing a relatively simple computation on huge sets of numbers. This is a direct result of the previous point; computers add and remember well, so efficient algorithms use these strengths to replace symbolic math.

Humans are poor at bookkeeping, which makes designing large systems difficult. Data abstraction, structures, and other programming constructs exist to ease bookkeeping; learn and use them in whatever environment is available. Spend time on the bookkeeping system, making it clear, concise, and efficient, and the rest of the problem will be much easier to code (e.g. I/O and computations).

2.4 Memory is fast, disk is slow

All computers have both memory and disk, typically in a ratio of 1 to 100. The penalty of disk size is speed. Disk is typically 100 (or more) times *slower* than memory. The performance gap is widening faster than the capacity gap, and has been since the introduction of the solid-state computer.

Hence, minimize disk access in any program. Keep results in memory if possible, recompute at need if necessary, and store on disk only if unavoidable. The desire to keep all necessary parts of a problem in memory at once often requires some careful management of data structures to pack the most data into the smallest space. This concern should not entirely trump maintainability, but can only be ignored on small problems.

In addition to the main memory of the computer, there is also a very fast, but small, memory on the CPU itself, called the cache. This is typically quite small (100-300 kB), but can be used to very quickly retrieve constants and intermediate results. Compilers (and many interpreted languages) can now automatically make good use of the cache to speed computations while minimizing memory access. The very largest problems will need to optimize cache storage as well as memory storage, but such advanced techniques are beyond the scope of this article.

3 How to implement algorithms

Implementation of algorithms is just as hard as choosing or designing a new algorithm, and has its' own set of tips and tricks.

3.1 Be lazy

Most algorithms are already implemented in a nice, fast function. Rather than writing a new implementation, which will have to be verified, then tuned and re-verified, adapt an existing implementation. Particularly for common operations (e.g. SVD of a matrix), fast, well-tested implementations exist for virtually every language (that doesn't have it built in). Use these implementations and spend more time on science.

Good implementations are as general as possible, to allow maximum reuse. However, more general solutions are often not as fast, so some problems benefit from very specialized versions that can exploit symmetries or limitations of a problem (e.g. sparse matrices for large systems). Be aware of the assumptions and specializations built into any implementation, and make sure that these do not compromise the results in a particular problem.

3.2 Start with loops

Since most problems are simple computations applied to many numbers, they lend themselves to coding as loops. Loops are often not the fastest possible solution. Therefore, start with a prototype code with loops, to make sure the algorithm works and will solve the problem. Then, start replacing simple loops with more efficient/faster options (e.g. vector ops in Matlab). This is best done in pieces, with each loop being replaced by a function/statement that implements the efficient solution.

As each loop/iteration is replaced, retest the code on a known input/output set to verify the new version. This bootstrapping can produce fast code that is also maintainable, as the loop replacement can stop at exactly the point where the code is "fast enough", leaving maximum transparency.

3.3 Maintainability over efficiency

Maintainable code is code that can be changed, modified, and reworked well after the original coding effort. Efficient code is small, clean, and fast to run. Writing maintainable, efficient code is nearly impossible for most problems; the techniques for maintainable code often run counter to the techniques for efficient code.

Therefore, do not expect to write maintainable, efficient code for real problems. Instead, strive to write maintainable code that is "efficient enough". Compute cycles are very cheap, and only get cheaper. This tips the balance in favor of maintainable code, as more compute speed can always be borrowed or purchased.

There is a final point to consider when considering maintainability or efficiency. Virtually all programs written for scientific work will last vastly longer than the original programmer intends. Programs represent the solution of a problem, and most problems recurr in science. A near-to-hand, proven solution is increasingly valuable the second, third, and hundredth time a problem recurrs.

Maintainable code will lend itself to small tweaks and ports, so a scientist can continuously tweak proven code for new problems, rather than having to start from scratch each time.

3.4 What is maintainable code?

Few programming courses teach examples of maintainable code. The goal of maintainable code is not only to run correctly, but also to be readable and understandable by someone who has knowledge of the problem, but not the program. Hence, maintainable ("clean") code is readable like a (boring) book, with no surprises or hidden magic.

There are many ways to write maintainable code, and it is an art like the design of algorithms. There are some common tips for writing code that makes all code more maintainable for minimal effort.

3.4.1 Comments, and the evils of hyper-commenting

Clean code should have comments, but only as many as necessary. Documenting the purpose, general algorithm, and strange variables (and units) is useful for all functions. Documenting the magical bits (clever, but unclear operations or operations with odd assumptions) is required.

Do not document every line of code, as most are perfectly clear on their own (in context). Additional useless comments add fluff and clutter to the code, and detract from readability.

Comments are precious; use them wisely.

3.4.2 Functions are your friends; use them well

Any chunk of code done more than once should be a function. Function calls are cheap, and good compilers can inline many functions anyway. Functions make algorithms cleaner, as they provide an easy abstraction from some details to a more general view.

Functions can also segregate complicated code into a single location for testing and maintainance. This also applies to debugging; if a repeated block is found to have an error, a function allows a single fix to propogate throughout a program automatically.

In general, any chunk of code done more than once should be a function; let the compiler inline the function for speed, and keep the readability of clean code.

3.4.3 Small is beautiful

Small functions are better than large, simply because anything that is off a page is hard to remember. Hence, it is better to break a large chunk of code into a number of small functions, even if they are called only once. In this case, the function call overhead is negligable, and the readability is greatly enhanced.

The use of small functions also forces a more fine-grained structure to the program, which eases abstraction and algorithm understanding.

3.4.4 The power of names

Avoid the use of very short, very cryptic names for important variables. Loop counters, temporary storage, or universal variables can have short names ("i" or "temp"), but these variables should never be used outside a very local scope (∼20 line blocks).

Use case and underscores (allowed in all modern languages) to make meaningful variable and function names. Someone with no knowledge of the program structure should be able to take any function, and deduce from the names and comments alone the purpose, scope, and type of all variables and function calls.

3.4.5 Reject globalization

Do not use global variables unless absolutely necessary. All modern languages have data structures of some sort, which can be used to transfer large numbers of parameters in a convenient way. This removes the major use of global variables.

Global variables can form very difficult-to-track bugs if they are corrupted in very different parts of a program. Use of global variables in functions can lead to terrible side-effects of a function call that can take days to find. All of these possible problems are avoided by simply using local variables in all functions and programs.

Note that Fortran, in particular, makes most variables global by default. This is a significant problem, and is a significant reason to avoid old dialects of Fortran (pre-F90). Modern languages (F90 or later, C, Java, Matlab, etc.) have local variables in functions if not everywhere by default. Be very careful in the use of global variables in any language.

3.4.6 Programs should not be Choose-Your-Own Adventures

Early programming languages made extensive use of the `GOTO` statement to change program flow, as early languages did not have the powerful function-definition, subroutine, and conditional loop constructs of modern languages. This is particularly true of Fortran, which still has `GOTO` statements pervading new code written today, to the great pain and agony of maintenance programmers everywhere (and *everyone* is a maintenance programmer). While `GOTO` was a necessary evil in the early days of programming, it is no longer a statement to be used in new code (there is a handful of advanced, unique exceptions, which will be clear when encountered).

Modern languages, such as C/C++, still include a `GOTO` statement in the language, but its' use is highly deprecated (frowned upon). The evils of using `GOTO` are apparent to anyone who has ever looked at complicated Fortran 77 codes; there are hundreds of `GOTO` statements that jump throughout the code, making it extremely difficult to determine the actual execution path of the program. This makes it very difficult to determine the state of the program at a given point, and hence makes debugging and maintenance orders of magnitude harder. This is the very definition of opaque code; code which is nearly impossible to understand and reverse-engineer, and hence nearly impossible to maintain.

Therefore, while `GOTO` may exist in a given language, it should never be used. Constructs which were done with `GOTO` are more clearly done with functions, subroutines, and loops. Anyone writing something other than an operating system kernel should never have a need for `GOTO`.

4 Summary

For ease of reference, a summary of the tips discused above:

- Algorithm Design Tips
  - Problems are rarely unique - Find the general problem from your specific one, and then adapt a general solution to the specific problem
  - Computers count and add well - Choose solutions that trade arithmetic for complex math
  - Bookkeeping is hard - Use data structures to ease the monumental bookkeeping required in most problems
  - Memory is fast, disk is slow - Use memory instead of disk if possible; keep data as small as possible to minimize disk access
- Algorithm implementation tips
  - Be lazy - Adapt existing codes rather than start from scratch; codes built from scratch must be verified and optimized
  - Start with loops - First make an algorithm work with simple-to-understand loops, then optimize to more efficient versions (e.g. vectorized computations); stop optimizing when the code is "fast enough" for the problem
  - Maintainability over efficiency - Compute cycles are cheap, programming time isn't
  - Tips for Maintainable Code
    - Comments, and the evils of hyper-commenting - Comments are precious, don't waste them on obvious code; wrong comments are worse than none at all
    - Functions are your friends; use them well - Anything done more than once is a function
    - Small is beautiful - Turn big functions and code blocks in many small ones
    - The power of names - Name variables, functions, etc. with meaningful identifiers; case and underscores can break apart words in a name

* Reject globalization - Use global variables only when unavoidable; data structures and function parameters make global variables almost useless
* No Choose-Your-Own Adventures - Do **NOT** use `GOTO`, ever.

The tips and techniques presented here are by no means an exhaustive list of good programming practice, but they will help produce code than is fast enough to solve a problem, yet understandable years from now. This is necessary, since code written today will likely be used decades from now, and will need to be adapted, ported, and fixed on machines undreamt today.