Chapter 1

Introduction

1.1 What this book is, and what it isn't

This book provides implementations of common and uncommon algorithms in pseudocode which is language independent and provides for easy porting to most imperative programming languages. It is not a definitive book on the theory of data structures and algorithms.

For the most part this book presents implementations devised by the authors themselves based on the concepts by which the respective algorithms are based upon so it is more than possible that our implementations differ from those considered the norm.

You should use this book alongside another on the same subject, but one that contains formal proofs of the algorithms in question. In this book we use the abstract big Oh notation to depict the run time complexity of algorithms so that the book appeals to a larger audience.

1.2 Assumed knowledge

We have written this book with few assumptions of the reader, but some have been necessary in order to keep the book as concise and approachable as possible. We assume that the reader is familiar with the following:

1. Big Oh notation
2. An imperative programming language
3. Object oriented concepts

1.2.1 Big Oh notation

For run time complexity analysis we use big Oh notation extensively so it is vital that you are familiar with the general concepts to determine which is the best algorithm for you in certain scenarios. We have chosen to use big Oh notation for a few reasons, the most important of which is that it provides an abstract measurement by which we can judge the performance of algorithms without using mathematical proofs.

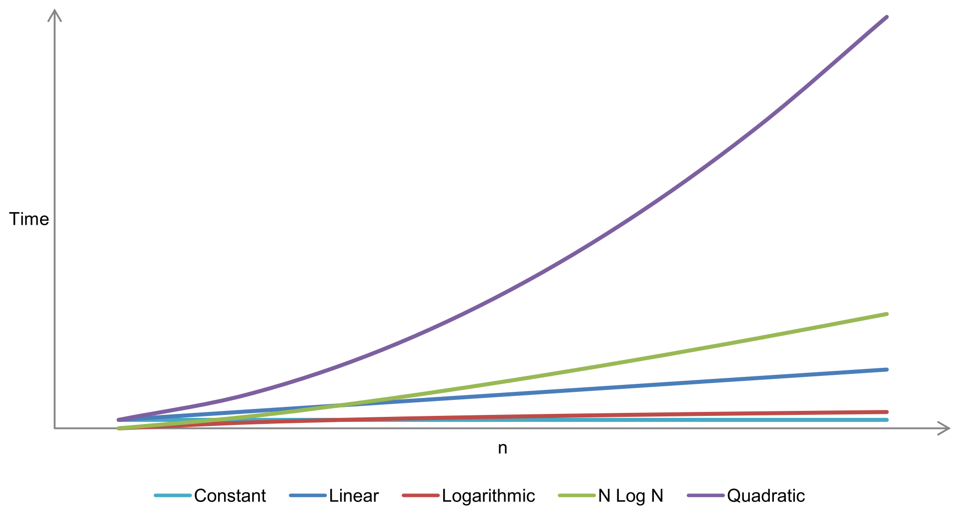

Figure 1.1: Algorithmic run time expansion

Figure 1.1 shows some of the run times to demonstrate how important it is to choose an efficient algorithm. For the sanity of our graph we have omitted cubic $O(n^3)$, and exponential $O(2^n)$ run times. Cubic and exponential algorithms should only ever be used for very small problems (if ever!); avoid them if feasibly possible.

The following list explains some of the most common big Oh notations:

$O(1)$ constant: the operation doesn't depend on the size of its input, e.g. adding a node to the tail of a linked list where we always maintain a pointer to the tail node.

$O(n)$ linear: the run time complexity is proportionate to the size of n.

$O(log\ n)$ logarithmic: normally associated with algorithms that break the problem into smaller chunks per each invocation, e.g. searching a binary search tree.

$O(n\ log\ n)$ just $n\ log\ n$: usually associated with an algorithm that breaks the problem into smaller chunks per each invocation, and then takes the results of these smaller chunks and stitches them back together, e.g. quick sort.

$O(n^2)$ quadratic: e.g. bubble sort.

$O(n^3)$ cubic: very rare.

$O(2^n)$ exponential: incredibly rare.

If you encounter either of the latter two items (cubic and exponential) this is really a signal for you to review the design of your algorithm. While prototyping algorithm designs you may just have the intention of solving the problem irrespective of how fast it works. We would strongly advise that you always review your algorithm design and optimise where possible—particularly loops and recursive calls—so that you can get the most efficient run times for your algorithms.

The biggest asset that big Oh notation gives us is that it allows us to essentially discard things like hardware. If you have two sorting algorithms, one with a quadratic run time, and the other with a logarithmic run time then the logarithmic algorithm will always be faster than the quadratic one when the data set becomes suitably large. This applies even if the former is ran on a machine that is far faster than the latter. Why? Because big Oh notation isolates a key factor in algorithm analysis: growth. An algorithm with a quadratic run time grows faster than one with a logarithmic run time. It is generally said at some point as $n \rightarrow \infty$ the logarithmic algorithm will become faster than the quadratic algorithm.

Big Oh notation also acts as a communication tool. Picture the scene: you are having a meeting with some fellow developers within your product group. You are discussing prototype algorithms for node discovery in massive networks. Several minutes elapse after you and two others have discussed your respective algorithms and how they work. Does this give you a good idea of how fast each respective algorithm is? No. The result of such a discussion will tell you more about the high level algorithm design rather than its efficiency. Replay the scene back in your head, but this time as well as talking about algorithm design each respective developer states the asymptotic run time of their algorithm. Using the latter approach you not only get a good general idea about the algorithm design, but also key efficiency data which allows you to make better choices when it comes to selecting an algorithm fit for purpose.

Some readers may actually work in a product group where they are given budgets per feature. Each feature holds with it a budget that represents its uppermost time bound. If you save some time in one feature it doesn't necessarily give you a buffer for the remaining features. Imagine you are working on an application, and you are in the team that is developing the routines that will essentially spin up everything that is required when the application is started. Everything is great until your boss comes in and tells you that the start up time should not exceed n ms. The efficiency of every algorithm that is invoked during start up in this example is absolutely key to a successful product. Even if you don't have these budgets you should still strive for optimal solutions.

Taking a quantitative approach for many software development properties will make you a far superior programmer - measuring one's work is critical to success.

1.2.2 Imperative programming language

All examples are given in a pseudo-imperative coding format and so the reader must know the basics of some imperative mainstream programming language to port the examples effectively, we have written this book with the following target languages in mind:

1. C++
2. C#
3. Java

The reason that we are explicit in this requirement is simple—all our implementations are based on an imperative thinking style. If you are a functional programmer you will need to apply various aspects from the functional paradigm to produce efficient solutions with respect to your functional language whether it be Haskell, F#, OCaml, etc.

Two of the languages that we have listed (C# and Java) target virtual machines which provide various things like security sand boxing, and memory management via garbage collection algorithms. It is trivial to port our implementations to these languages. When porting to C++ you must remember to use pointers for certain things. For example, when we describe a linked list node as having a reference to the next node, this description is in the context of a managed environment. In C++ you should interpret the reference as a pointer to the next node and so on. For programmers who have a fair amount of experience with their respective language these subtleties will present no issue, which is why we really do emphasise that the reader must be comfortable with at least one imperative language in order to successfully port the pseudo-implementations in this book.

It is essential that the user is familiar with primitive imperative language constructs before reading this book otherwise you will just get lost. Some algorithms presented in this book can be confusing to follow even for experienced programmers!

1.2.3 Object oriented concepts

For the most part this book does not use features that are specific to any one language. In particular, we never provide data structures or algorithms that work on generic types—this is in order to make the samples as easy to follow as possible. However, to appreciate the designs of our data structures you will need to be familiar with the following object oriented (OO) concepts:

1. Inheritance
2. Encapsulation
3. Polymorphism

This is especially important if you are planning on looking at the C# target that we have implemented (more on that in §1.7) which makes extensive use of the OO concepts listed above. As a final note it is also desirable that the reader is familiar with interfaces as the C# target uses interfaces throughout the sorting algorithms.

1.3 Pseudocode

Throughout this book we use pseudocode to describe our solutions. For the most part interpreting the pseudocode is trivial as it looks very much like a more abstract C++, or C#, but there are a few things to point out:

1. Pre-conditions should always be enforced
2. Post-conditions represent the result of applying algorithm a to data structure d

3. The type of parameters is inferred
4. All primitive language constructs are explicitly begun and ended

If an algorithm has a return type it will often be presented in the post-condition, but where the return type is sufficiently obvious it may be omitted for the sake of brevity.

Most algorithms in this book require parameters, and because we assign no explicit type to those parameters the type is inferred from the contexts in which it is used, and the operations performed upon it. Additionally, the name of the parameter usually acts as the biggest clue to its type. For instance n is a pseudo-name for a number and so you can assume unless otherwise stated that n translates to an integer that has the same number of bits as a WORD on a 32 bit machine, similarly l is a pseudo-name for a list where a list is a resizeable array (e.g. a vector).

The last major point of reference is that we always explicitly end a language construct. For instance if we wish to close the scope of a **for** loop we will explicitly state **end for** rather than leaving the interpretation of when scopes are closed to the reader. While implicit scope closure works well in simple code, in complex cases it can lead to ambiguity.

The pseudocode style that we use within this book is rather straightforward. All algorithms start with a simple algorithm signature, e.g.

```
1) algorithm AlgorithmName($arg1$, $arg2$, ..., $argN$)
2) ...
n) end AlgorithmName
```

Immediately after the algorithm signature we list any **Pre** or **Post** conditions.

```
1) algorithm AlgorithmName($n$)
2)     Pre: $n$ is the value to compute the factorial of
3)          $n \geq 0$
4)     Post: the factorial of $n$ has been computed
5)     // ...
n) end AlgorithmName
```

The example above describes an algorithm by the name of *AlgorithmName*, which takes a single numeric parameter n. The pre and post conditions follow the algorithm signature; you should always enforce the pre-conditions of an algorithm when porting them to your language of choice.

Normally what is listed as a pre-conidition is critical to the algorithms operation. This may cover things like the actual parameter not being null, or that the collection passed in must contain at least n items. The post-condition mainly describes the effect of the algorithms operation. An example of a post-condition might be “The list has been sorted in ascending order”

Because everything we describe is language independent you will need to make your own mind up on how to best handle pre-conditions. For example, in the C# target we have implemented, we consider non-conformance to pre-conditions to be exceptional cases. We provide a message in the exception to tell the caller why the algorithm has failed to execute normally.

1.4 Tips for working through the examples

As with most books you get out what you put in and so we recommend that in order to get the most out of this book you work through each algorithm with a pen and paper to track things like variable names, recursive calls etc.

The best way to work through algorithms is to set up a table, and in that table give each variable its own column and continuously update these columns. This will help you keep track of and visualise the mutations that are occurring throughout the algorithm. Often while working through algorithms in such a way you can intuitively map relationships between data structures rather than trying to work out a few values on paper and the rest in your head. We suggest you put everything on paper irrespective of how trivial some variables and calculations may be so that you always have a point of reference.

When dealing with recursive algorithm traces we recommend you do the same as the above, but also have a table that records function calls and who they return to. This approach is a far cleaner way than drawing out an elaborate map of function calls with arrows to one another, which gets large quickly and simply makes things more complex to follow. Track everything in a simple and systematic way to make your time studying the implementations far easier.

1.5 Book outline

We have split this book into two parts:

Part 1: Provides discussion and pseudo-implementations of common and uncommon data structures; and

Part 2: Provides algorithms of varying purposes from sorting to string operations.

The reader doesn't have to read the book sequentially from beginning to end: chapters can be read independently from one another. We suggest that in part 1 you read each chapter in its entirety, but in part 2 you can get away with just reading the section of a chapter that describes the algorithm you are interested in.

Each of the chapters on data structures present initially the algorithms concerned with:

1. Insertion
2. Deletion
3. Searching

The previous list represents what we believe in the vast majority of cases to be the most important for each respective data structure.

For all readers we recommend that before looking at any algorithm you quickly look at Appendix E which contains a table listing the various symbols used within our algorithms and their meaning. One keyword that we would like to point out here is **yield**. You can think of **yield** in the same light as **return**. The **return** keyword causes the method to exit and returns control to the caller, whereas **yield** returns each value to the caller. With **yield** control only returns to the caller when all values to return to the caller have been exhausted.

1.6 Testing

All the data structures and algorithms have been tested using a minimised test driven development style on paper to flesh out the pseudocode algorithm. We then transcribe these tests into unit tests satisfying them one by one. When all the test cases have been progressively satisfied we consider that algorithm suitably tested.

For the most part algorithms have fairly obvious cases which need to be satisfied. Some however have many areas which can prove to be more complex to satisfy. With such algorithms we will point out the test cases which are tricky and the corresponding portions of pseudocode within the algorithm that satisfy that respective case.

As you become more familiar with the actual problem you will be able to intuitively identify areas which may cause problems for your algorithms implementation. This in some cases will yield an overwhelming list of concerns which will hinder your ability to design an algorithm greatly. When you are bombarded with such a vast amount of concerns look at the overall problem again and sub-divide the problem into smaller problems. Solving the smaller problems and then composing them is a far easier task than clouding your mind with too many little details.

The only type of testing that we use in the implementation of all that is provided in this book are unit tests. Because unit tests contribute such a core piece of creating somewhat more stable software we invite the reader to view Appendix D which describes testing in more depth.

1.7 Where can I get the code?

This book doesn't provide any code specifically aligned with it, however we do actively maintain an open source project[1] that houses a C# implementation of all the pseudocode listed. The project is named *Data Structures and Algorithms* (DSA) and can be found at `http://codeplex.com/dsa`.

1.8 Final messages

We have just a few final messages to the reader that we hope you digest before you embark on reading this book:

1. Understand how the algorithm works first in an abstract sense; and
2. Always work through the algorithms on paper to understand how they achieve their outcome

If you always follow these key points, you will get the most out of this book.

[1]All readers are encouraged to provide suggestions, feature requests, and bugs so we can further improve our implementations.