Appendix A

Algorithm Walkthrough

Learning how to design good algorithms can be assisted greatly by using a structured approach to tracing its behaviour. In most cases tracing an algorithm only requires a single table. In most cases tracing is not enough, you will also want to use a diagram of the data structure your algorithm operates on. This diagram will be used to visualise the problem more effectively. Seeing things visually can help you understand the problem quicker, and better.

The trace table will store information about the variables used in your algorithm. The values within this table are constantly updated when the algorithm mutates them. Such an approach allows you to attain a history of the various values each variable has held. You may also be able to infer patterns from the values each variable has contained so that you can make your algorithm more efficient.

We have found this approach both simple, and powerful. By combining a visual representation of the problem as well as having a history of past values generated by the algorithm it can make understanding, and solving problems much easier.

In this chapter we will show you how to work through both iterative, and recursive algorithms using the technique outlined.

A.1 Iterative algorithms

We will trace the *IsPalindrome* algorithm (defined in §11.2) as our example iterative walkthrough. Before we even look at the variables the algorithm uses, first we will look at the actual data structure the algorithm operates on. It should be pretty obvious that we are operating on a string, but how is this represented? A string is essentially a block of contiguous memory that consists of some char data types, one after the other. Each character in the string can be accessed via an index much like you would do when accessing items within an array. The picture should be presenting itself - a string can be thought of as an array of characters.

For our example we will use *IsPalindrome* to operate on the string "Never odd or even" Now we know how the string data structure is represented, and the value of the string we will operate on let's go ahead and draw it as shown in Figure A.1.

N	e	v	e	r		o	d	d		o	r		e	v	e	n
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure A.1: Visualising the data structure we are operating on

value	*word*	*left*	*right*

Table A.1: A column for each variable we wish to track

The *IsPalindrome* algorithm uses the following list of variables in some form throughout its execution:

1. *value*
2. *word*
3. *left*
4. *right*

Having identified the values of the variables we need to keep track of we simply create a column for each in a table as shown in Table A.1.

Now, using the *IsPalindrome* algorithm execute each statement updating the variable values in the table appropriately. Table A.2 shows the final table values for each variable used in *IsPalindrome* respectively.

While this approach may look a little bloated in print, on paper it is much more compact. Where we have the strings in the table you should annotate these strings with array indexes to aid the algorithm walkthrough.

There is one other point that we should clarify at this time - whether to include variables that change only a few times, or not at all in the trace table. In Table A.2 we have included both the *value*, and *word* variables because it was convenient to do so. You may find that you want to promote these values to a larger diagram (like that in Figure A.1) and only use the trace table for variables whose values change during the algorithm. We recommend that you promote the core data structure being operated on to a larger diagram outside of the table so that you can interrogate it more easily.

value	*word*	*left*	*right*
"Never odd or even"	"NEVERODDOREVEN"	0	13
		1	12
		2	11
		3	10
		4	9
		5	8
		6	7
		7	6

Table A.2: Algorithm trace for *IsPalindrome*

We cannot stress enough how important such traces are when designing your algorithm. You can use these trace tables to verify algorithm correctness. At the cost of a simple table, and quick sketch of the data structure you are operating on you can devise correct algorithms quicker. Visualising the problem domain and keeping track of changing data makes problems a lot easier to solve. Moreover you always have a point of reference which you can look back on.

A.2 Recursive Algorithms

For the most part working through recursive algorithms is as simple as walking through an iterative algorithm. One of the things that we need to keep track of though is which method call returns to who. Most recursive algorithms are much simple to follow when you draw out the recursive calls rather than using a table based approach. In this section we will use a recursive implementation of an algorithm that computes a number from the Fiboncacci sequence.

```
1) algorithm Fibonacci($n$)
2)    Pre: $n$ is the number in the fibonacci sequence to compute
3)    Post: the fibonacci sequence number $n$ has been computed
4)    if $n < 1$
5)       return 0
6)    else if $n < 2$
7)       return 1
8)    end if
9)    return Fibonacci($n - 1$) + Fibonacci($n - 2$)
10) end Fibonacci
```

Before we jump into showing you a diagrammtic representation of the algorithm calls for the *Fibonacci* algorithm we will briefly talk about the cases of the algorithm. The algorithm has three cases in total:

1. $n < 1$
2. $n < 2$
3. $n \geq 2$

The first two items in the preceeding list are the base cases of the algorithm. Until we hit one of our base cases in our recursive method call tree we won't return anything. The third item from the list is our recursive case.

With each call to the recursive case we etch ever closer to one of our base cases. Figure A.2 shows a diagrammtic representation of the recursive call chain. In Figure A.2 the order in which the methods are called are labelled. Figure A.3 shows the call chain annotated with the return values of each method call as well as the order in which methods return to their callers. In Figure A.3 the return values are represented as annotations to the red arrows.

It is important to note that each recursive call only ever returns to its caller upon hitting one of the two base cases. When you do eventually hit a base case that branch of recursive calls ceases. Upon hitting a base case you go back to

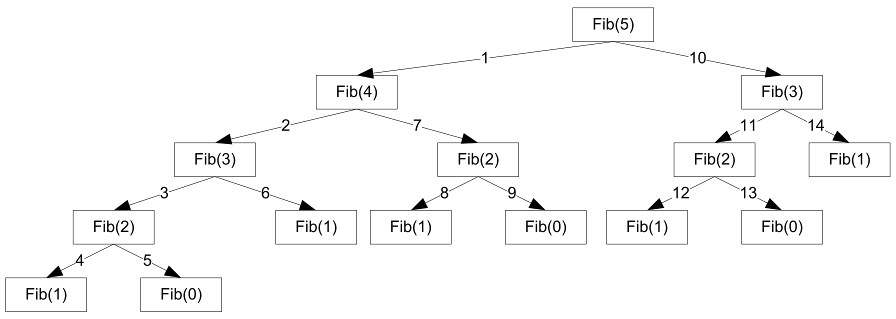

Figure A.2: Call chain for *Fibonacci* algorithm

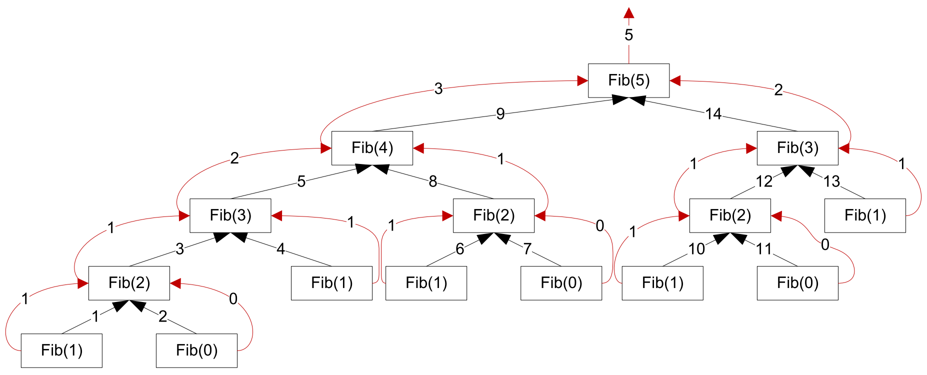

Figure A.3: Return chain for *Fibonacci* algorithm

the caller and continue execution of that method. Execution in the caller is contiued at the next statement, or expression after the recursive call was made.

In the *Fibonacci* algorithms' recursive case we make two recursive calls. When the first recursive call (Fibonacci($n - 1$)) returns to the caller we then execute the the second recursive call (Fibonacci($n - 2$)). After both recursive calls have returned to their caller, the caller can then subesequently return to its caller and so on.

Recursive algorithms are much easier to demonstrate diagrammatically as Figure A.2 demonstrates. When you come across a recursive algorithm draw method call diagrams to understand how the algorithm works at a high level.

A.3 Summary

Understanding algorithms can be hard at times, particularly from an implementation perspective. In order to understand an algorithm try and work through it using trace tables. In cases where the algorithm is also recursive sketch the recursive calls out so you can visualise the call/return chain.

In the vast majority of cases implementing an algorithm is simple provided that you know how the algorithm works. Mastering how an algorithm works from a high level is key for devising a well designed solution to the problem in hand.

Appendix B

Translation Walkthrough

The conversion from pseudo to an actual imperative language is usually very straight forward, to clarify an example is provided. In this example we will convert the algorithm in §9.1 to the C# language.

```
public static bool IsPrime(int number)
{
    if (number < 2)
    {
        return false;
    }
    int innerLoopBound = (int)Math.Floor(Math.Sqrt(number));
    for (int i = 1; i < number; i++)
    {
        for(int j = 1; j <= innerLoopBound; j++)
        {
            if (i * j == number)
            {
                return false;
            }
        }
    }
    return true;
}
```

For the most part the conversion is a straight forward process, however you may have to inject various calls to other utility algorithms to ascertain the correct result.

A consideration to take note of is that many algorithms have fairly strict preconditions, of which there may be several - in these scenarios you will need to inject the correct code to handle such situations to preserve the correctness of the algorithm. Most of the preconditions can be suitably handled by throwing the correct exception.

B.1 Summary

As you can see from the example used in this chapter we have tried to make the translation of our pseudo code algorithms to mainstream imperative languages as simple as possible.

Whenever you encounter a keyword within our pseudo code examples that you are unfamiliar with just browse to Appendix E which descirbes each keyword.

Appendix C

Recursive Vs. Iterative Solutions

One of the most succinct properties of modern programming languages like C++, C#, and Java (as well as many others) is that these languages allow you to define methods that reference themselves, such methods are said to be recursive. One of the biggest advantages recursive methods bring to the table is that they usually result in more readable, and compact solutions to problems.

A recursive method then is one that is defined in terms of itself. Generally a recursive algorithms has two main properties:

1. One or more base cases; and
2. A recursive case

For now we will briefly cover these two aspects of recursive algorithms. With each recursive call we should be making progress to our base case otherwise we are going to run into trouble. The trouble we speak of manifests itself typically as a stack overflow, we will describe why later.

Now that we have briefly described what a recursive algorithm is and why you might want to use such an approach for your algorithms we will now talk about iterative solutions. An iterative solution uses no recursion whatsoever. An iterative solution relies only on the use of loops (e.g. for, while, do-while, etc). The down side to iterative algorithms is that they tend not to be as clear as to their recursive counterparts with respect to their operation. The major advantage of iterative solutions is speed. Most production software you will find uses little or no recursive algorithms whatsoever. The latter property can sometimes be a companies prerequisite to checking in code, e.g. upon checking in a static analysis tool may verify that the code the developer is checking in contains no recursive algorithms. Normally it is systems level code that has this zero tolerance policy for recursive algorithms.

Using recursion should always be reserved for fast algorithms, you should avoid it for the following algorithm run time deficiencies:

1. $O(n^2)$
2. $O(n^3)$

3. $O(2^n)$

If you use recursion for algorithms with any of the above run time efficiency's you are inviting trouble. The growth rate of these algorithms is high and in most cases such algorithms will lean very heavily on techniques like divide and conquer. While constantly splitting problems into smaller problems is good practice, in these cases you are going to be spawning a lot of method calls. All this overhead (method calls don't come *that* cheap) will soon pile up and either cause your algorithm to run a lot slower than expected, or worse, you will run out of stack space. When you exceed the allotted stack space for a thread the process will be shutdown by the operating system. This is the case irrespective of the platform you use, e.g. .NET, or native C++ etc. You can ask for a bigger stack size, but you typically only want to do this if you have a very good reason to do so.

C.1 Activation Records

An activation record is created every time you invoke a method. Put simply an activation record is something that is put on the stack to support method invocation. Activation records take a small amount of time to create, and are pretty lightweight.

Normally an activation record for a method call is as follows (this is very general):

- The actual parameters of the method are pushed onto the stack
- The return address is pushed onto the stack
- The top-of-stack index is incremented by the total amount of memory required by the local variables within the method
- A jump is made to the method

In many recursive algorithms operating on large data structures, or algorithms that are inefficient you will run out of stack space quickly. Consider an algorithm that when invoked given a specific value it creates many recursive calls. In such a case a big chunk of the stack will be consumed. We will have to wait until the activation records start to be unwound after the nested methods in the call chain exit and return to their respective caller. When a method exits it's activation record is unwound. Unwinding an activation record results in several steps:

1. The top-of-stack index is decremented by the total amount of memory consumed by the method
2. The return address is popped off the stack
3. The top-of-stack index is decremented by the total amount of memory consumed by the actual parameters

While activation records are an efficient way to support method calls they can build up very quickly. Recursive algorithms can exhaust the stack size allocated to the thread fairly fast given the chance.

Just about now we should be dusting the cobwebs off the age old example of an iterative vs. recursive solution in the form of the Fibonacci algorithm. This is a famous example as it highlights both the beauty and pitfalls of a recursive algorithm. The iterative solution is not as pretty, nor self documenting but it does the job a lot quicker. If we were to give the Fibonacci algorithm an input of say 60 then we would have to wait a while to get the value back because it has an $O(g^n)$ run time. The iterative version on the other hand has a $O(n)$ run time. Don't let this put you off recursion. This example is mainly used to shock programmers into thinking about the ramifications of recursion rather than warning them off.

C.2 Some problems are recursive in nature

Something that you may come across is that some data structures and algorithms are actually recursive in nature. A perfect example of this is a tree data structure. A common tree node usually contains a value, along with two pointers to two other nodes of the same node type. As you can see tree is recursive in its makeup wit each node possibly pointing to two other nodes.

When using recursive algorithms on tree's it makes sense as you are simply adhering to the inherent design of the data structure you are operating on. Of course it is not all good news, after all we are still bound by the limitations we have mentioned previously in this chapter.

We can also look at sorting algorithms like merge sort, and quick sort. Both of these algorithms are recursive in their design and so it makes sense to model them recursively.

C.3 Summary

Recursion is a powerful tool, and one that all programmers should know of. Often software projects will take a trade between readability, and efficiency in which case recursion is great provided you don't go and use it to implement an algorithm with a quadratic run time or higher. Of course this is not a rule of thumb, this is just us throwing caution to the wind. Defensive coding will always prevail.

Many times recursion has a natural home in recursive data structures and algorithms which are recursive in nature. Using recursion in such scenarios is perfectly acceptable. Using recursion for something like linked list traversal is a little overkill. Its iterative counterpart is probably less lines of code than its recursive counterpart.

Because we can only talk about the implications of using recursion from an abstract point of view you should consult your compiler and run time environment for more details. It may be the case that your compiler recognises things like tail recursion and can optimise them. This isn't unheard of, in fact most commercial compilers will do this. The amount of optimisation compilers can do though is somewhat limited by the fact that you are still using recursion. You, as the developer have to accept certain accountability's for performance.

Appendix D

Testing

Testing is an essential part of software development. Testing has often been discarded by many developers in the belief that the burden of proof of their software is on those within the company who hold test centric roles. This couldn't be further from the truth. As a developer you should at least provide a suite of unit tests that verify certain boundary conditions of your software.

A great thing about testing is that you build up progressively a safety net. If you add or tweak algorithms and then run your suite of tests you will be quickly alerted to any cases that you have broken with your recent changes. Such a suite of tests in any sizeable project is absolutely essential to maintaining a fairly high bar when it comes to quality. Of course in order to attain such a standard you need to think carefully about the tests that you construct.

Unit testing which will be the subject of the vast majority of this chapter are widely available on most platforms. Most modern languages like C++, C#, and Java offer an impressive catalogue of testing frameworks that you can use for unit testing.

The following list identifies testing frameworks which are popular:

JUnit: Targeted at Jav., `http://www.junit.org/`

NUnit: Can be used with languages that target Microsoft's Common Language Runtime. `http://www.nunit.org/index.php`

Boost Test Library: Targeted at C++. The test library that ships with the incredibly popular Boost libraries. `http://www.boost.org`. A direct link to the libraries documentation `http://www.boost.org/doc/libs/1_36_0/libs/test/doc/html/index.html`

CppUnit: Targeted at C++. `http://cppunit.sourceforge.net/`

Don't worry if you think that the list is very sparse, there are far more on offer than those that we have listed. The ones listed are the testing frameworks that we believe are the most popular for C++, C#, and Java.

D.1 What constitutes a unit test?

A unit test should focus on a single atomic property of the subject being tested. Do not try and test many things at once, this will result in a suite of somewhat unstructured tests. As an example if you were wanting to write a test that verified that a particular value V is returned from a specific input I then your test should do the smallest amount of work possible to verify that V is correct given I. A unit test should be simple and self describing.

As well as a unit test being relatively atomic you should also make sure that your unit tests execute quickly. If you can imagine in the future when you may have a test suite consisting of thousands of tests you want those tests to execute as quickly as possible. Failure to attain such a goal will most likely result in the suite of tests not being ran that often by the developers on your team. This can occur for a number of reasons but the main one would be that it becomes incredibly tedious waiting several minutes to run tests on a developers local machine.

Building up a test suite can help greatly in a team scenario, particularly when using a continuous build server. In such a scenario you can have the suite of tests devised by the developers and testers ran as part of the build process.

Employing such strategies can help you catch niggling little error cases early rather than via your customer base. There is nothing more embarrassing for a developer than to have a very trivial bug in their code reported to them from a customer.

D.2 When should I write my tests?

A source of great debate would be an understatement to personify such a question as this. In recent years a test driven approach to development has become very popular. Such an approach is known as test driven development, or more commonly the acronym TDD.

One of the founding principles of TDD is to write the unit test first, watch it fail and then make it pass. The premise being that you only ever write enough code at any one time to satisfy the state based assertions made in a unit test. We have found this approach to provide a more structured intent to the implementation of algorithms. At any one stage you only have a single goal, to make the failing test pass. Because TDD makes you write the tests up front you never find yourself in a situation where you forget, or can't be bothered to write tests for your code. This is often the case when you write your tests after you have coded up your implementation. We, as the authors of this book ourselves use TDD as our preferred method.

As we have already mentioned that TDD is our favoured approach to testing it would be somewhat of an injustice to not list, and describe the mantra that is often associate with it:

Red: Signifies that the test has failed.

Green: The failing test now passes.

Refactor: Can we restructure our program so it makes more sense, and easier to maintain?

The first point of the above list always occurs at least once (more if you count the build error) in TDD initially. Your task at this stage is solely to make the test pass, that is to make the respective test green. The last item is based around the restructuring of your program to make it as readable and maintainable as possible. The last point is very important as TDD is a progressive methodology to building a solution. If you adhere to progressive revisions of your algorithm restructuring when appropriate you will find that using TDD you can implement very cleanly structured types and so on.

D.3 How seriously should I view my test suite?

Your tests are a major part of your project ecosystem and so they should be treated with the same amount of respect as your production code. This ranges from correct, and clean code formatting, to the testing code being stored within a source control repository.

Employing a methodology like TDD, or testing after implementing you will find that you spend a great amount of time writing tests and thus they should be treated no differently to your production code. All tests should be clearly named, and fully documented as to their intent.

D.4 The three A's

Now that you have a sense of the importance of your test suite you will inevitably want to know how to actually structure each block of imperatives within a single unit test. A popular approach - the three A's is described in the following list:

- Assemble: Create the objects you require in order to perform the state based assertions.
- Act: Invoke the respective operations on the objects you have assembled to mutate the state to that desired for your assertions.
- Assert: Specify what you expect to hold after the previous two steps.

The following example shows a simple test method that employs the three A's:

```
public void MyTest()
{
    // assemble
    Type t = new Type();
    // act
    t.MethodA();
    // assert
    Assert.IsTrue(t.BoolExpr)
}
```

D.5 The structuring of tests

Structuring tests can be viewed upon as being the same as structuring production code, e.g. all unit tests for a *Person* type may be contained within a *PersonTest* type. Typically all tests are abstracted from production code. That is that the tests are disjoint from the production code, you may have two dynamic link libraries (dll); the first containing the production code, the second containing your test code.

We can also use things like inheritance etc when defining classes of tests. The point being that the test code is very much like your production code and you should apply the same amount of thought to its structure as you would do the production code.

D.6 Code Coverage

Something that you can get as a product of unit testing are code coverage statistics. Code coverage is merely an indicator as to the portions of production code that your units tests cover. Using TDD it is likely that your code coverage will be very high, although it will vary depending on how easy it is to use TDD within your project.

D.7 Summary

Testing is key to the creation of a moderately stable product. Moreover unit testing can be used to create a safety blanket when adding and removing features providing an early warning for breaking changes within your production code.

Appendix E

Symbol Definitions

Throughout the pseudocode listings you will find several symbols used, describes the meaning of each of those symbols.

Symbol	**Description**
$\leftarrow$	Assignment.
$=$	Equality.
$\leq$	Less than or equal to.
$<$	Less than.*
$\geq$	Greater than or equal to.
$>$	Greater than.*
$\neq$	Inequality.
$\emptyset$	Null.
and	Logical and.
or	Logical or.
whitespace	Single occurrence of whitespace.
yield	Like **return** but builds a sequence.

Table E.1: Pseudo symbol definitions

* This symbol has a direct translation with the vast majority of imperative counterparts.