The Importance of Algorithms

By **lbackstrom** — *topcoder member*
Discuss this article in the forums

Introduction

The first step towards an understanding of why the study and knowledge of algorithms are so important is to define exactly what we mean by an algorithm. According to the popular algorithms textbook Introduction to Algorithms (Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein), "an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output." In other words, algorithms are like road maps for accomplishing a given, well-defined task. So, a chunk of code that calculates the terms of the Fibonacci sequence is an implementation of a particular algorithm. Even a simple function for adding two numbers is an algorithm in a sense, albeit a simple one.

Some algorithms, like those that compute the Fibonacci sequences, are intuitive and may be innately embedded into our logical thinking and problem solving skills. However, for most of us, complex algorithms are best studied so we can use them as building blocks for more efficient logical problem solving in the future. In fact, you may be surprised to learn just how many complex algorithms people use every day when they check their e-mail or listen to music on their computers. This article will introduce some basic ideas related to the analysis of algorithms, and then put these into practice with a few examples illustrating why it is important to know about algorithms.

Runtime Analysis

One of the most important aspects of an algorithm is how fast it is. It is often easy to come up with an algorithm to solve a problem, but if the algorithm is too slow, it's back to the drawing board. Since the exact speed of an algorithm depends on where the algorithm is run, as well as the exact details of its implementation, computer scientists typically talk about the runtime relative to the size of the input. For example, if the input consists of N integers, an algorithm might have a runtime proportional to N^2, represented as $O(N^2)$. This means that if you were to run an implementation of the algorithm on your computer with an input of size N, it would take $C*N^2$ seconds, where C is some constant that doesn't change with the size of the input.

However, the execution time of many complex algorithms can vary due to factors other than the size of the input. For example, a sorting algorithm may run much faster when given a set of integers that are already sorted than it would when given the same set of integers in a random order. As a result, you often hear people talk about the worst-case runtime, or the average-case runtime. The worst-case runtime is how long it would take for the algorithm to run if it were given the most insidious of all possible inputs. The average-case runtime is the average of how long it would take the algorithm to run if it were given all possible inputs. Of the two, the worst-case is often easier to reason about, and therefore is more frequently used as a benchmark for a given algorithm. The process of

determining the worst-case and average-case runtimes for a given algorithm can be tricky, since it is usually impossible to run an algorithm on all possible inputs. There are many good online resources that can help you in estimating these values.

Approximate completion time for algorithms, N = 100

$O(Log(N))$	10^{-7} seconds
$O(N)$	10^{-6} seconds
$O(N*Log(N))$	10^{-5} seconds
$O(N^2)$	10^{-4} seconds
$O(N^6)$	3 minutes
$O(2^N)$	10^{14} years.
$O(N!)$	10^{142} years.

Sorting

Sorting provides a good example of an algorithm that is very frequently used by computer scientists. The simplest way to sort a group of items is to start by removing the smallest item from the group, and put it first. Then remove the next smallest, and put it next and so on. Unfortunately, this algorithm is O(N2), meaning that the amount of time it takes is proportional to the number of items squared. If you had to sort a billion things, this algorithm would take around 10^{18} operations. To put this in perspective, a desktop PC can do a little bit over 109 operations per second, and would take years to finish sorting a billion things this way.

Luckily, there are a number of better algorithms (quicksort, heapsort and mergesort, for example) that have been devised over the years, many of which have a runtime of O(N * Log(N)). This brings the number of operations required to sort a billion items down to a reasonable number that even a cheap desktop could perform. Instead of a billion squared operations (10^{18}) these algorithms require only about 10 billion operations (10^{10}), a factor of 100 million faster.

Shortest Path

Algorithms for finding the shortest path from one point to another have been researched for years. Applications abound, but lets keep things simple by saying we want to find the shortest path from point A to point B in a city with just a few streets and intersections. There are quite a few different algorithms that have been developed to solve such problems, all with different benefits and drawbacks. Before we delve into them though, lets consider how long a naive algorithm – one that tries every conceivable option – would take to run. If the algorithm considered every possible path from A to B (that didn't go in circles), it would not finish in our lifetimes, even if A and B were both in a small town. The runtime of this algorithm is exponential in the size of the input, meaning that it is O(CN) for some C. Even for small values of C, CN becomes astronomical when N gets even moderately large.

One of the fastest algorithms for solving this problem has a runtime of $O(E*V*Log(V))$, where E is the number of road segments, and V is the number of intersections. To put this in perspective, the algorithm would take about 2 seconds to find the shortest path in a city with 10,000 intersections, and 20,000 road segments (there are usually about 2 road segments per intersection). The algorithm, known as Djikstra's Algorithm, is fairly complex, and requires the use of a data structure known as a priority queue. In some applications, however, even this runtime is too slow (consider finding the shortest path from New York City to San Francisco – there are millions of intersections in the US), and programmers try to do better by using what are known as heuristics. A heuristic is an approximation of something that is relevant to the problem, and is often computed by an algorithm of its own. In the shortest path problem, for example, it is useful to know approximately how far a point is from the destination. Knowing this allows for the development of faster algorithms (such as A*, an algorithm that can sometimes run significantly faster than Djikstra's algorithm) and so programmers come up with heuristics to approximate this value. Doing so does not always improve the runtime of the algorithm in the worst case, but it does make the algorithm faster in most real-world applications.

Approximate algorithms

Sometimes, however, even the most advanced algorithm, with the most advanced heuristics, on the fastest computers is too slow. In this case, sacrifices must be made that relate to the correctness of the result. Rather than trying to get the shortest path, a programmer might be satisfied to find a path that is at most 10% longer than the shortest path.

In fact, there are quite a few important problems for which the best-known algorithm that produces an optimal answer is insufficiently slow for most purposes. The most famous group of these problems is called NP, which stands for non-deterministic polynomial (don't worry about what that means). When a problem is said to be NP-complete or NP-hard, it mean no one knows a good way to solve them optimally. Furthermore, if someone did figure out an efficient algorithm for one NP-complete problem, that algorithm would be applicable to all NP-complete problems.

A good example of an NP-hard problem is the famous traveling salesman problem. A salesman wants to visit N cities, and he knows how long it takes to get from each city to each other city. The question is "how fast can he visit all of the cities?" Since the fastest known algorithm for solving this problem is too slow – and many believe this will always be true – programmers look for sufficiently fast algorithms that give good, but not optimal solutions.

Random Algorithms

Yet another approach to some problems is to randomize an algorithm is some way. While doing so does not improve the algorithm in the worst case, it often makes very good algorithms in the average case. Quicksort is a good example of an algorithm where randomization is often used. In the worst case, quicksort sorts a group of items in $O(N2)$, where N is the number of items. If randomization is incorporated into the algorithm, however, the chances of the worst case actually occurring become diminishingly small, and on average, quicksort has a runtime of $O(N*Log(N))$. Other algorithms guarantee a runtime of $O(N*Log(N))$, even in the worst case, but they are slower in the average case. Even though both algorithms have a runtime proportional to $N*Log(N)$, quicksort has a smaller

constant factor – that is it requires C*N*Log(N) operations, while other algorithms require more like 2*C*N*Log(N) operations.

Another algorithm that uses random numbers finds the median of a group of numbers with an average runtime of O(N). This is a significant improvement over sorting the numbers and taking the middle one, which takes O(N*Log(N)). Furthermore, while deterministic (non-random) algorithms exist for finding the median with a runtime of O(N), the random algorithm is attractively simple, and often faster than the deterministic algorithms.

The basic idea of the median algorithm is to pick one of the numbers in the group at random, and count how many of the numbers in the group are less than it. Lets say there are N numbers, and K of them are less than or equal to the number we picked at random. If K is less than half of N, then we know that the median is the (N/2-K) th number that is greater than the random number we picked, so we discard the K numbers less than or equal to the random number. Now, we want to find the (N/2-K) th smallest number, instead of the median. The algorithm is the same though, and we simply pick another number at random, and repeat the above steps.

Compression

Another class of algorithm deals with situations such as data compression. This type of algorithm does not have an expected output (like a sorting algorithm), but instead tries to optimize some other criteria. In the case of data compression, the algorithm (LZW, for instance) tries to make the data use as few bytes as possible, in such a way that it can be decompressed to its original form. In some cases, this type of algorithm will use the same techniques as other algorithms, resulting in output that is good, but potentially sub-optimal. JPG and MP3 compression, for example, both compress data in a way that makes the final result somewhat lower quality than the original, but they create much smaller files. MP3 compression does not retain every feature of the original song file, but it attempts to maintain enough of the details to capture most of the quality, while at the same time ensuring the significantly reduced file size that we all know and love. The JPG image file format follows the same principle, but the details are significantly different since the goal is image rather than audio compression.

The Importance of Knowing Algorithms

As a computer scientist, it is important to understand all of these types of algorithms so that one can use them properly. If you are working on an important piece of software, you will likely need to be able to estimate how fast it is going to run. Such an estimate will be less accurate without an understanding of runtime analysis. Furthermore, you need to understand the details of the algorithms involved so that you'll be able to predict if there are special cases in which the software won't work quickly, or if it will produce unacceptable results.

Of course, there are often times when you'll run across a problem that has not been previously studied. In these cases, you have to come up with a new algorithm, or apply an old algorithm in a new way. The more you know about algorithms in this case, the better your chances are of finding a good way to solve the problem. In many cases, a new problem can be reduced to an old problem without too much effort, but you will need to have a fundamental understanding of the old problem in order to do this.

As an example of this, lets consider what a switch does on the Internet. A switch has N cables plugged into it, and

receives packets of data coming in from the cables. The switch has to first analyze the packets, and then send them back out on the correct cables. A switch, like a computer, is run by a clock with discrete steps – the packets are send out at discrete intervals, rather than continuously. In a fast switch, we want to send out as many packets as possible during each interval so they don't stack up and get dropped. The goal of the algorithm we want to develop is to send out as many packets as possible during each interval, and also to send them out so that the ones that arrived earlier get sent out earlier. In this case it turns out that an algorithm for a problem that is known as "stable matching" is directly applicable to our problem, though at first glance this relationship seems unlikely. Only through pre-existing algorithmic knowledge and understanding can such a relationship be discovered.

More Real-world Examples

Other examples of real-world problems with solutions requiring advanced algorithms abound. Almost everything that you do with a computer relies in some way on an algorithm that someone has worked very hard to figure out. Even the simplest application on a modern computer would not be possible without algorithms being utilized behind the scenes to manage memory and load data from the hard drive.

There are dozens of applications of complicated algorithms, but I'm going to discuss two problems that require the same skills as some past TopCoder problems. The first is known as the maximum flow problem, and the second is related to dynamic programming, a technique that often solves seemingly impossible problems in blazing speed.

Maximum Flow

The maximum flow problem has to do with determining the best way to get some sort of stuff from one place to another, through a network of some sort. In more concrete terms, the problem first arose in relation to the rail networks of the Soviet Union, during the 1950′s. The US wanted to know how quickly the Soviet Union could get supplies through its rail network to its satellite states in Eastern Europe.

In addition, the US wanted to know which rails it could destroy most easily to cut off the satellite states from the rest of the Soviet Union. It turned out that these two problems were closely related, and that solving the max flow problem also solves the min cut problem of figuring out the cheapest way to cut off the Soviet Union from its satellites.

The first efficient algorithm for finding the maximum flow was conceived by two Computer Scientists, named Ford and Fulkerson. The algorithm was subsequently named the Ford-Fulkerson algorithm, and is one of the more famous algorithms in computer science. In the last 50 years, a number of improvements have been made to the Ford-Fulkerson algorithm to make it faster, some of which are dauntingly complex.

Since the problem was first posed, many additional applications have been discovered. The algorithm has obvious relevance to the Internet, where getting as much data as possible from one point to another is important. It also comes up in many business settings, and is an important part of operations research. For example, if you have N employees and N jobs that need to be done, but not every employee can do every job, the max flow algorithm will tell you how to assign your N employees to jobs in such a way that every job gets done, provided that's possible.

Graduation, from SRM 200, is a good example of a TopCoder problem that lends itself to a solution using max flow.

Sequence comparison

Many coders go their entire careers without ever having to implement an algorithm that uses dynamic programming. However, dynamic programming pops up in a number of important algorithms. One algorithm that most programmers have probably used, even though they may not have known it, finds differences between two sequences. More specifically, it calculates the minimum number of insertions, deletions, and edits required to transform sequence A into sequence B.

For example, lets consider two sequences of letters, "AABAA" and "AAAB". To transform the first sequence into the second, the simplest thing to do is delete the B in the middle, and change the final A into a B. This algorithm has many applications, including some DNA problems and plagiarism detection. However, the form in which many programmers use it is when comparing two versions of the same source code file. If the elements of the sequence are lines in the file, then this algorithm can tell a programmer which lines of code were removed, which ones were inserted, and which ones were modified to get from one version to the next.

Without dynamic programming, we would have to consider a – you guessed it – exponential number of transformations to get from one sequence to the other. As it is, however, dynamic programming makes for an algorithm with a runtime of only O(N*M), where N and M are the numbers of elements in the two sequences.

Conclusion

The different algorithms that people study are as varied as the problems that they solve. However, chances are good that the problem you are trying to solve is similar to another problem in some respects. By developing a good understanding of a large range of algorithms, you will be able to choose the right one for a problem and apply it properly. Furthermore, solving problems like those found in TopCoder's competitions will help you to hone your skills in this respect. Many of the problems, though they may not seem realistic, require the same set of algorithmic knowledge that comes up every day in the real world.

More Resources

[Member Tutorials](#)
Read more than 40 data science tutorials written by topcoder members.

[Problem Set Analysis](#)
Read editorials explaining the problem and solution for each Single Round Match (SRM).

[Data Science Guide](#)
New to topcoder's data science track? Read this guide for an overview on how to get started in the arena and how competitions work.

[Help Center](#)
Need specifics about the process or the rules? Everything you need to know about competing at topcoder can be found in the Help Center.

[Member Forums](#)

Join your peers in our member forums and ask questions from the real experts - topcoder members!

SITEMAP

ABOUT US

CONTACT US

HELP CENTER

PRIVACY POLICY

TERMS

Topcoder is also on