Taxonomy of Complexity

Chapter 5 had the chance to briefly mention the complexity of some operations on the .NET Framework's collections and some collections of our own implementation. This section defines more accurately what Big-Oh complexity means and reviews the main complexity classes known in computability and complexity theory.

Big-Oh Notation

When we discussed the complexity of a lookup operation on a `List<T>` in Chapter 5, we said it had runtime complexity $O(n)$. What we meant to say informally is that, when you have a list of 1,000 items and are trying to find in it another item, then the *worst-case* running time of that operation is 1,000 iterations through the list—namely, if the item is not already in the list. Thus, we try to estimate the order of magnitude of the running time's growth as the inputs become larger. When specified formally, however, the Big-Oh notation might appear slightly confusing:

Suppose that the function T(A;n) returns the number of computation steps required to execute the algorithm A on an input size of n elements, and let f(n) be a monotonic function from positive integers to positive integers. Then, T(A;n) is O(f(n)), if there exists a constant c such that for all n, T(A;n) ≤ cf(n).

In a nutshell, we can say that an algorithm's runtime complexity is $O(f(n))$, if $f(n)$ is an upper bound on the actual number of steps it takes to execute the algorithm on an input of size n. The bound does not have to be tight; for example, we can also say that `List<T>` lookup has runtime complexity $O(n^4)$. However, using such a loose upper bound is not helpful, because it fails to capture the fact that it *is* realistic to search a `List<T>` for an element even if it has 1,000,000 elements in it. If `List<T>` lookup had tight runtime complexity $O(n^4)$, it would be highly inefficient to perform lookups even on lists with several thousand elements.

Additionally, the bound might be tight for some inputs but not tight for others; for example, if we search the list for an item that happens to be its first element, the number of steps is clearly constant (one!) for all list sizes—this is why we mentioned *worst-case* running time in the preceding paragraphs.

Some examples of how this notation makes it easier to reason about running time and compare different algorithms include:

- If one algorithm has runtime complexity $2n^3 + 4$ and another algorithm has runtime complexity $½n^3-n^2$, we can say that both algorithms have runtime complexity $O(n^3)$, so they are equivalent as far as Big-Oh complexity is concerned (try to find the constant c that works for each of these running times). It is easy to prove that we can omit all but the largest term when talking about Big-Oh complexity.
- If one algorithm that has runtime complexity n^2 and another algorithm has runtime complexity $100n + 5000$, we can still assert that the first algorithm is slower on large inputs, because it has Big-Oh complexity $O(n^2)$, as opposed to $O(n)$. Indeed, for $n = 1{,}000$, it is already the case that the first algorithm runs significantly slower than the second.

Similar to the definition of an upper bound on runtime complexity, there are also lower bounds (denoted by $\Omega(f(n))$) and tight bounds (denoted by $\Theta(f(n))$). They are less frequently used to discuss algorithm complexity, however, so we omit them here.

THE MASTER THEOREM

The *master theorem* is a simple result that provides a ready-made solution for analyzing the complexity of many recursive algorithms that decompose the original problem into smaller chunks. For example, consider the following code, which implements the *merge sort* algorithm:

```
public static List<T> MergeSort(List<T> list) where T : IComparable < T > {
  if (list.Count <= 1) return list;
  int middle = list.Count / 2;
  List<T> left = list.Take(middle).ToList();
  List<T> right = list.Skip(middle).ToList();
  left = MergeSort(left);
  right = MergeSort(right);
  return Merge(left, right);
}
private List < T > Merge(List < T > left, List < T > right) where T : IComparable < T > {
  List < T > result = new List < T > ();
  int i = 0, j = 0;
  while (i < left.Count || j < right.Count) {
    if (i < left.Count && j < right.Count) {
      if (left[i].CompareTo(right[j]) <= 0)
        result.Add(left[i++]);
      else
      result.Add(right[j++]);
    } else if (i < left.Count) {
        result.Add(left[i++];
    } else {
      result.Add(right++);
    }
  }
  return result;
}
```

Analyzing this algorithm's runtime complexity requires solving the *recurrence equation* for its running time, $T(n)$, which is given recursively as $T(n) = 2T(n/2) + O(n)$. The explanation is that, for every invocation of `MergeSort`, we recurse into `MergeSort` for each half of the original list and perform linear-time work merging the lists (clearly, the `Merge` helper method performs exactly n operations for an original list of size n).

One approach to solving recurrence equations is guessing the outcome, then trying to prove (usually by mathematical induction) the correctness of the result. In this case, we can expand some terms and see if a pattern emerges:

T(n) = 2T(n/2) + O(n) = 2(2T(n/4) + O(n/2)) + O(n) = 2(2(2T(n/8) + O(n/4)) + O(n/2)) + O(n) = . . .

The master theorem provides a closed-form solution for this recurrence equation and many others. According to the master theorem, T(n) = O(n logn), which is a well-known result about the complexity of merge sort (in fact, it holds for θ, as well as O). For more details about the master theorem, consult the Wikipedia article at `http://en.wikipedia.org/wiki/Master_theorem`.

Turing Machines and Complexity Classes

It is common to refer to algorithms and problems as being "in *P*" or "in *NP*" or "*NP*-complete". These refer to different complexity classes. The classification of problems into complexity classes helps computer scientists easily identify problems that have reasonable (*tractable*) solutions and reject or find simplifications for problems that do not.

A *Turing machine* (*TM*) is a theoretical computation device that models a machine operating on an infinite *tape* of *symbols*. The machine's head can read or write one symbol at a time from the tape, and internally the machine can be in one of a finite number of *states*. The device's operation is fully determined by a finite set of *rules* (an algorithm), such as "when in state Q and the symbol on the tape is 'A', write 'a' " or "when in state P and the symbol on the tape is 'a', move the head to the right and switch to state S". There are also two special states: the *start state* from which the machine begins operation and the *end state*. When the machine reaches the end state, it is common to say it loops forever or simply halts execution. Figure 9-1 shows an example of a Turing machine's definition—the circles are states, and the arrows indicate state transitions in the syntax `read;write;head_move_direction`.

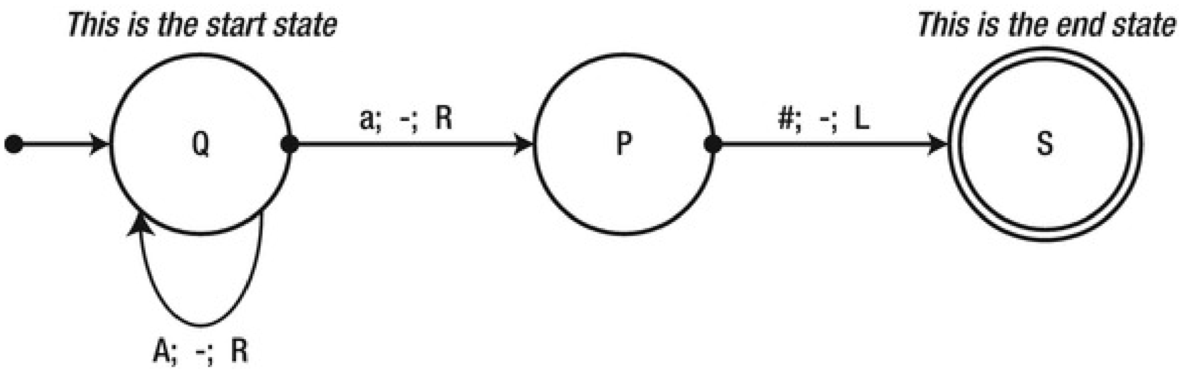

Figure 9-1 *. A state diagram of a simple Turing machine. The leftmost arrow points into the initial state, Q. The looped arrow shows that when the machine is in state Q and reads the symbol A, it moves to the right and remains in state Q*

When discussing the complexity of algorithms on a Turing machine, there is no need to reason in vague "iterations"—a computation step of a TM is a single state transition (including a transition from a state to itself). For example, when the TM in Figure 9-1 starts with the input "AAAa#" on its tape, it performs exactly four computation steps. We can generalize this to say that, for an input of *n* 'A's followed by "a#", the machine performs *O*(*n*) steps. (Indeed, for such an input of size *n,* the machine performs exactly *n* + 2 steps, so the definition of *O*(*n*) holds, for example, with the constant *c* = 3.)

Modeling real-world computation using a Turing machine is quite difficult—it makes for a good exercise in an undergraduate course on automata theory, but has no practical use. The amazing result is that every C# program (and, in fact, every algorithm that can be executed on a modern computer) can be translated—laboriously—to a Turing machine. Roughly speaking, if an algorithm written in C# has runtime complexity *O*(*f* (*n*)), then the same algorithm translated to a Turing machine has runtime complexity of *O*(f^2(*n*)). This has very useful implications on analyzing algorithm complexity: if a problem has an efficient algorithm for a Turing machine, then it has an efficient algorithm for a modern computer; if a problem does not have an efficient algorithm for a Turing machine, then it, typically, does not have an efficient algorithm for a modern computer.

Although we could call *O*(n^2) algorithms efficient and any "slower" algorithms inefficient, complexity theory takes a slightly different stance. *P* is the set of all problems a Turing machine can solve in polynomial time—in other words, if *A* is a problem in *P* (with input of size *n*), then there exists a Turing machine that produces the desired result on its tape within *polynomial time* (i.e. within *O*(n^k) steps for some natural number *k*). In many subfields of complexity theory, problems in *P* are considered easy—and algorithms that run in polynomial time are considered efficient, even though *k* and, therefore, the running time can be very large for some algorithms.

All of the algorithms considered so far in this book are efficient if we embrace this definition. Still, some are "very" efficient and others less so, indicating that this separation is not sufficiently subtle. You might even ask whether there are any problems that are not in *P*, problems that do not have efficient solutions. The answer is a resounding yes—and, in fact, from a theoretical perspective, there are *more* problems that do not have efficient solutions than problems that do.

First, we consider a problem that a Turing machine cannot solve, regardless of efficiency. Then, we see there are problems a Turing machine can solve, but not in polynomial time. Finally, we turn to problems for which *we do not know* whether there is a Turing machine that can solve them in polynomial time, but strongly suspect that there is not.

The Halting Problem

From a mathematical perspective, there are more problems than Turing machines (we say that Turing machines are "countable" and problems are not), which means there must be infinitely many problems that cannot be solved by Turing machines. This class of problems is often called *undecidable* problems.

WHAT DO YOU MEAN BY "COUNTABLE"?

In mathematics, there are many kinds of "infinite". It is easy to see that there are an infinite number of Turing machines—after all, you could add a dummy state that does nothing to any Turing machine and obtain a new, larger Turing machine. Similarly, it is easy to see that there are an infinite number problems—this requires a formal definition of a problem (as a "language") but leads to the same result. However, it is not obvious why there are "more" problems than Turing machines, especially since the number of both is infinite.

The set of Turing machines is said to be *countable*, because there is a one-to-one correspondence from natural numbers (1, 2, 3, . . .) to Turing machines. It might not be immediately obvious how to construct this correspondence, but it is possible, because Turing machines can be described as finite strings and the set of *all* finite strings is countable.

However, the set of problems (languages) is *not* countable, because a one-to-one correspondence from natural numbers to languages does not exist. A possible proof is along the following lines: consider the set of problems corresponding to all real numbers, where, for any real number *r,* the problem is to print the number or to recognize whether the number has been supplied as input. A well-known result (Cantor's Theorem) is that the real numbers are uncountable, and, hence, this set of problems is uncountable as well.

To summarize, this seems like an unfortunate conclusion. Not only are there problems that cannot be solved (decided) by a Turing machine, but there are many *more* such problems than problems that can be solved by a Turing machine. Luckily, a great many problems *can* be solved by Turing machines, as the incredible evolution of computers in the 20th century shows, theoretical results notwithstanding.

The *halting problem*, which we now introduce, is undecidable. The problem is as follows: receive as input a program *T* (or a description of a Turing machine) and an input *w* to the program; return `TRUE` if *T* halts when executed on *w* and `FALSE` if it does not halt (enters an infinite loop).

You could even translate this problem to a C# method that accepts a program's code as a string:

```
public static bool DoesHaltOnInput(string programCode, string input) { . . . }
```

. . .or even a C# method that takes a delegate and an input to feed it:

```
public static bool DoesHaltOnInput(Action< string > program, string input) { . . . }
```

Although it may appear there is a way to analyze a program and determine whether it halts or not (e.g. by inspecting its loops, its calls to other methods, and so on), it turns out that there is neither a Turing machine nor a C# program that can solve this problem. How does one reach this conclusion? Obviously, to say a Turing machine can solve a problem, we only need to demonstrate the Turing machine—but, to say that there exists no Turing machine to solve a particular problem, it appears that we have to go over all possible Turing machines, which are infinite in number.

As often is the case in mathematics, we use a proof by contradiction. Suppose someone came up with the method `DoesHaltOnInput`, which tests whether Then we could write the following C# method:

```
public static void Helper(string programCode, string input) {
  bool doesHalt = DoesHaltOnInput(programCode, input);
  if (doesHalt) {
    while (true) {} //Enter an infinite loop
  }
}
```

Now all it takes is to call `DoesHaltOnInput` on the source of the `Helper` method (the second parameter is meaningless). If `DoesHaltOnInput` returns `true`, the `Helper` method enters an infinite loop; if `DoesHaltOnInput` returns `false`, the `Helper` method does not enter an infinite loop. This contradiction illustrates that the `DoesHaltOnInput` method does not exist.

Note The halting problem is a humbling result; it demonstrates, in simple terms, that there are limits to the computational ability of our computing devices. The next time you blame the compiler for not finding an apparently trivial optimization or your favorite static analysis tool for giving you false warnings you can see will never occur, remember that statically analyzing a program and acting on the results is often undecidable. This is the case with optimization, halting analysis, determining whether a variable is used, and many other problems that might be easy for a developer given a specific program but cannot be generally solved by a machine.

There are many more undecidable problems. Another simple example stems again from the counting argument. There are a countable number of C# programs, because every C# program is a finite combination of symbols. However, there are uncountably many real numbers in the interval [0,1]. Therefore, there must exist a real number that cannot be printed out by a C# program.

NP-Complete Problems

Even within the realm of decidable problems—those that can be solved by a Turing machine—there are problems that do not have efficient solutions. Computing a perfect strategy for a game of chess on an $n \times n$ chessboard requires time *exponential* in *n*, which places the problem of solving the generalized chess game outside of *P*. (If you like checkers and dislike computer programs playing checkers better than humans, you should take some solace in the fact that generalized checkers is not in *P* either.)

There are problems, however, that are thought to be less complex, but for which we do not yet have a polynomial algorithm. Some of these problems would prove quite useful in real-life scenarios:

- The *traveling salesman* problem: Find a path of minimal cost for a salesman who has to visit *n* different cities.
- The *maximum clique* problem: Find the largest subset of nodes in a graph such that every two nodes in the subset are connected by an edge.
- The *minimum cut* problem: Find a way to divide a graph into two subsets of nodes such that there are a minimal number of edges crossing from one subset to the other.
- The *Boolean satisfiability* problem: Determine whether a Boolean formula of a certain form (such as "*A* and *B* or *C* and not *A*") can be satisfied by an assignment of truth values to its variables.
- The *cache placement* problem: Decide which data to place in cache and which data to evict from cache, given a complete history of memory accesses performed by an application.

These problems belong in another set of problems called *NP*. The problems in *NP* are characterized as follows: if *A* is a problem in *NP*, then there exists a Turing machine that can *verify* the solution of *A* for an input of size *n* in polynomial time. For example, verifying that a truth assignment to variables is legal and solves the Boolean satisfiability problem is clearly very easy, as well as linear in the number of variables. Similarly, verifying that a subset of nodes is a clique is very easy. In other words, these problems have easily verifiable solutions, but it is not known whether there is a way to efficiently come up with these solutions.

Another interesting facet of the above problems (and many others) is that if *any* has an efficient solution, then they *all* have an efficient solution. The reason is that they can be *reduced* from one to another. Moreover, if *any* of these problems has an efficient solution, which means that problem is in *P*, then the entire *NP* complexity class collapses into *P* such that *P* = *NP*. Arguably the biggest mystery in theoretical computer science today is whether *P* = *NP* (most computer scientists believe these complexity classes are not equal).

Problems that have this collapsing effect on *P* and *NP* are called *NP-complete* problems. For most computer scientists, showing that a problem is *NP*-complete is sufficient to reject any attempts to devise an efficient algorithm for it. Subsequent sections consider some examples of *NP*-complete problems that have acceptable approximate or probabilistic solutions.