14

Analysis Techniques

This book contains many examples of asymptotic analysis of the time requirements for algorithms and the space requirements for data structures. Often it is a simple matter to invent an equation to model the behavior of the algorithm or data structure in question, and then a simple matter to derive a closed-form solution for the equation should it contain a recurrence or summation.

Sometimes an analysis proves more difficult. It may take a clever insight to derive the right model, such as the snowplow argument for analyzing the average run length resulting from Replacement Selection (Section 8.7). The equations resulting from the snowplow argument are quite simple. In other cases, developing the model is straightforward, but analyzing the resulting equations is not. An example is the average-case analysis for Quicksort. The equation given in Section 7.4 simply enumerates all possible cases for the pivot position, summing corresponding costs for the recursive calls to Quicksort. However, deriving a closed-form solution for the resulting recurrence relation is not as easy.

Many iterative algorithms require that we compute a summation to determine the cost of a loop. Techniques for finding closed-form solutions to summations are presented in Section 14.1. Time requirements for many algorithms based on recursion are best modeled by recurrence relations. A brief discussion of techniques for solving recurrences is provided in Section 14.2. These sections extend the introduction to summations and recurrences provided in Section 2.5; the reader should already be familiar with that material.

Section 14.3 provides an introduction to the topic of **amortized analysis**. Amortized analysis deals with the cost of a series of operations. Perhaps a single operation in the series has high cost, but as a result the cost of the remaining operations is limited in such a way that the entire series can be done efficiently. Amortized analysis has been used successfully to analyze several of the algorithms presented in this book, including the cost of a series of UNION/FIND operations (Section 6.2), the cost of a series of splay tree operations (Section 13.2), and the cost of a series of operations on self-organizing lists (Section 9.2). Section 14.3 discusses the topic in more detail.

14.1 Summation Techniques

This section presents some basic techniques for deriving closed-form solutions for summations (also referred to as "solving" the summation). Our first approach is the "guess and test" method, appropriate for summations whose closed-form solution is a simple polynomial.

Example 14.1 Consider the familiar summation $\sum_{i=0}^{n} i$. Clearly, this is less than $\sum_{i=0}^{n} n$, which is simply $n^2 + n$. Thus, it is reasonable to guess that the closed-form solution for this summation is a polynomial of the form $c_1n^2 + c_2n + c_3$ for some constants c_1, c_2, and c_3. If this is the case, we can plug in the answers to small cases of the summation to solve for the coefficients. For this example, substituting 0, 1, and 2 for n leads to three simultaneous equations. Since the summation when $n = 0$ is just 0, c_3 must be 0. For $n = 1$ and $n = 2$ we get the two equations

$$\begin{aligned} c_1 + c_2 &= 1 \\ 4c_1 + 2c_2 &= 3, \end{aligned}$$

which in turn yield $c_1 = 1/2$ and $c_2 = 1/2$. If the closed-form solution for the summation is a polynomial, it can only be

$$1/2n^2 + 1/2n + 0$$

which is more commonly written

$$\frac{n(n+1)}{2}.$$

At this point, we should use an induction proof to verify that our candidate closed-form solution is indeed correct. In this case it is correct, as shown by Example 2.10. The induction proof is necessary because our initial assumption that the solution is a simple polynomial could be wrong. For example, it is possible that the true solution includes a logarithmic term. The process shown here is essentially fitting a curve to a fixed number of points, and there is always an n-degree polynomial that fits $n + 1$ points, so we cannot be sure that we have checked enough points to know the true equation.

The approach of "guess and test" is useful whenever the solution is a polynomial expression. In particular, similar reasoning can be used to solve for $\sum_{i=1}^{n} i^2$, or more generally $\sum_{i=1}^{n} i^c$ for c any positive integer.

The **shifting method** is a more general approach to solving summations. The shifting method subtracts the summation from a variation on the summation. The variation selected for the subtraction should be one that makes most of the terms cancel out.

Example 14.2 Our first example of shifting sums solves the summation

$$F(n) = \sum_{i=0}^{n} ar^i = a + ar + ar^2 + \cdots + ar^n.$$

This is called a geometric series. Our goal is to find some variation for $F(n)$ such that subtracting one from the other leaves us with an easily manipulated equation. Since the difference between consecutive terms of the summation is a factor of r, we can shift terms if we multiply the entire expression by r:

$$rF(n) = r\sum_{i=0}^{n} ar^i = ar + ar^2 + ar^3 + \cdots + ar^{n+1}.$$

We can now subtract the one equation from the other, as follows:

$$\begin{aligned} F(n) - rF(n) = a \quad &+ \quad ar + ar^2 + ar^3 + \cdots + ar^n \\ &- \quad (ar + ar^2 + ar^3 + \cdots + ar^n) - ar^{n+1}. \end{aligned}$$

The result leaves only the end terms:

$$\begin{aligned} F(n) - rF(n) &= \sum_{i=0}^{n} ar^i - r\sum_{i=0}^{n} ar^i. \\ (1-r)F(n) &= a - ar^{n+1}. \end{aligned}$$

Thus, we get the result

$$F(n) = \frac{a - ar^{n+1}}{1 - r}$$

where $r \neq 1$.

Example 14.3 For our second example of the shifting method, we solve

$$F(n) = \sum_{i=1}^{n} i2^i = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + n \cdot 2^n.$$

We can achieve our goal if we multiply by two:

$$2F(n) = 2\sum_{i=1}^{n} i2^i = 1 \cdot 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + \cdots + (n-1) \cdot 2^n + n \cdot 2^{n+1}.$$

The ith term of $2F(n)$ is $i \cdot 2^{i+1}$, while the $(i+1)$th term of $F(n)$ is $(i+1) \cdot 2^{i+1}$. Subtracting one expression from the other yields the summation of 2^i and a few non-canceled terms:

$$\begin{aligned} 2F(n) - F(n) &= 2\sum_{i=1}^{n} i2^i - \sum_{i=1}^{n} i2^i \\ &= \sum_{i=1}^{n} i2^{i+1} - \sum_{i=1}^{n} i2^i. \end{aligned}$$

Shift i's value in the second summation, substituting $(i+1)$ for i:

$$= n2^{n+1} + \sum_{i=0}^{n-1} i2^{i+1} - \sum_{i=0}^{n-1} (i+1)2^{i+1}.$$

Break the second summation into two parts:

$$= n2^{n+1} + \sum_{i=0}^{n-1} i2^{i+1} - \sum_{i=0}^{n-1} i2^{i+1} - \sum_{i=0}^{n-1} 2^{i+1}.$$

Cancel like terms:

$$= n2^{n+1} - \sum_{i=0}^{n-1} 2^{i+1}.$$

Again shift i's value in the summation, substituting i for $(i+1)$:

$$= n2^{n+1} - \sum_{i=1}^{n} 2^i.$$

Replace the new summation with a solution that we already know:

$$= n2^{n+1} - \left(2^{n+1} - 2\right).$$

Finally, reorganize the equation:

$$= (n-1)2^{n+1} + 2.$$

14.2 Recurrence Relations

Recurrence relations are often used to model the cost of recursive functions. For example, the standard Mergesort (Section 7.5) takes a list of size n, splits it in half, performs Mergesort on each half, and finally merges the two sublists in n steps. The cost for this can be modeled as

$$\mathbf{T}(n) = 2\mathbf{T}(n/2) + n.$$

In other words, the cost of the algorithm on input of size n is two times the cost for input of size $n/2$ (due to the two recursive calls to Mergesort) plus n (the time to merge the sublists together again).

There are many approaches to solving recurrence relations, and we briefly consider three here. The first is an estimation technique: Guess the upper and lower bounds for the recurrence, use induction to prove the bounds, and tighten as required. The second approach is to expand the recurrence to convert it to a summation and then use summation techniques. The third approach is to take advantage of already proven theorems when the recurrence is of a suitable form. In particular, typical divide and conquer algorithms such as Mergesort yield recurrences of a form that fits a pattern for which we have a ready solution.

14.2.1 Estimating Upper and Lower Bounds

The first approach to solving recurrences is to guess the answer and then attempt to prove it correct. If a correct upper or lower bound estimate is given, an easy induction proof will verify this fact. If the proof is successful, then try to tighten the bound. If the induction proof fails, then loosen the bound and try again. Once the upper and lower bounds match, you are finished. This is a useful technique when you are only looking for asymptotic complexities. When seeking a precise closed-form solution (i.e., you seek the constants for the expression), this method will not be appropriate.

Example 14.4 Use the guessing technique to find the asymptotic bounds for Mergesort, whose running time is described by the equation

$$\mathbf{T}(n) = 2\mathbf{T}(n/2) + n; \quad \mathbf{T}(2) = 1.$$

We begin by guessing that this recurrence has an upper bound in $O(n^2)$. To be more precise, assume that

$$\mathbf{T}(n) \leq n^2.$$

We prove this guess is correct by induction. In this proof, we assume that n is a power of two, to make the calculations easy. For the base case, $\mathbf{T}(2) = 1 \leq 2^2$. For the induction step, we need to show that $\mathbf{T}(n) \leq n^2$ implies that $\mathbf{T}(2n) \leq (2n)^2$ for $n = 2^N, N \geq 1$. The induction hypothesis is

$$\mathbf{T}(i) \leq i^2, \text{for all } i \leq n.$$

It follows that

$$\mathbf{T}(2n) = 2\mathbf{T}(n) + 2n \leq 2n^2 + 2n \leq 4n^2 \leq (2n)^2$$

which is what we wanted to prove. Thus, $\mathbf{T}(n)$ is in $\mathrm{O}(n^2)$.

Is $\mathrm{O}(n^2)$ a good estimate? In the next-to-last step we went from n^2+2n to the much larger $4n^2$. This suggests that $\mathrm{O}(n^2)$ is a high estimate. If we guess something smaller, such as $\mathbf{T}(n) \leq cn$ for some constant c, it should be clear that this cannot work since $c2n = 2cn$ and there is no room for the extra n cost to join the two pieces together. Thus, the true cost must be somewhere between cn and n^2.

Let us now try $\mathbf{T}(n) \leq n \log n$. For the base case, the definition of the recurrence sets $\mathbf{T}(2) = 1 \leq (2 \cdot \log 2) = 2$. Assume (induction hypothesis) that $\mathbf{T}(n) \leq n \log n$. Then,

$$\mathbf{T}(2n) = 2\mathbf{T}(n) + 2n \leq 2n \log n + 2n \leq 2n(\log n + 1) \leq 2n \log 2n$$

which is what we seek to prove. In similar fashion, we can prove that $\mathbf{T}(n)$ is in $\Omega(n \log n)$. Thus, $\mathbf{T}(n)$ is also $\Theta(n \log n)$.

14.2.2 Expanding Recurrences

Estimating bounds is effective if you only need an approximation to the answer. More precise techniques are required to find an exact solution. One such technique is called **expanding** the recurrence. In this method, the smaller terms on the right side of the equation are in turn replaced by their definition. This is the expanding step. These terms are again expanded, and so on, until a full series with no recurrence results. This yields a summation, and techniques for solving summations can then be used. A couple of simple expansions were shown in Section 2.5; a more complex example is given below.

Example 14.5 Find the solution for

$$\mathbf{T}(n) = 2\mathbf{T}(n/2) + 5n^2; \quad \mathbf{T}(1) = 7.$$

For simplicity we assume that n is a power of two, so we will rewrite it as $n = 2^k$. This recurrence can be expanded as follows:

$$\begin{aligned}
\mathbf{T}(n) &= 2\mathbf{T}(n/2) + 5n^2 \\
&= 2(2\mathbf{T}(n/4) + 5(n/2)^2) + 5n^2 \\
&= 2(2(2\mathbf{T}(n/8) + 5(n/4)^2) + 5(n/2)^2) + 5n^2 \\
&= 2^k\mathbf{T}(1) + 2^{k-1} \cdot 5\left(\frac{n}{2^{k-1}}\right)^2 + \cdots + 2 \cdot 5\left(\frac{n}{2}\right)^2 + 5n^2.
\end{aligned}$$

This last expression can best be represented by a summation as follows:

$$\begin{aligned}
& 7n + 5\sum_{i=0}^{k-1} n^2/2^i \\
= \quad & 7n + 5n^2 \sum_{i=0}^{k-1} 1/2^i.
\end{aligned}$$

From Equation 2.7, we have:

$$\begin{aligned}
&= 7n + 5n^2\left(2 - 1/2^{k-1}\right) \\
&= 7n + 5n^2(2 - 2/n) \\
&= 7n + 10n^2 - 10n \\
&= 10n^2 - 3n.
\end{aligned}$$

This is the *exact* solution to the recurrence for n a power of two. At this point, we should use a simple induction proof to verify that our solution is indeed correct.

14.2.3 Divide and Conquer Recurrences

The third approach to solving recurrences is to take advantage of known theorems that describe the solution for classes of recurrences. One useful example is a theorem that gives the answer for a class known as **divide and conquer** recurrences. These have the form

$$\mathbf{T}(n) = a\mathbf{T}(n/b) + cn^k; \quad \mathbf{T}(1) = c$$

where a, b, c, and k are constants. In general, this recurrence describes a problem of size n divided into a subproblems of size n/b, while cn^k is the amount of work necessary to combine the partial solutions. Mergesort is an example of a divide and conquer algorithm, and its recurrence fits this form. So does binary search. We use the method of expanding recurrences to derive the general solution for any divide and conquer recurrence, assuming that $n = b^m$.

$$\begin{aligned}\mathbf{T}(n) &= a(a\mathbf{T}(n/b^2) + c(n/b)^k) + cn^k \\ &= a^m\mathbf{T}(1) + a^{m-1}c(n/b^{m-1})^k + \cdots + ac(n/b)^k + cn^k \\ &= c\sum_{i=0}^{m} a^{m-i}b^{ik} \\ &= ca^m\sum_{i=0}^{m}(b^k/a)^i.\end{aligned}$$

Note that

$$a^m = a^{\log_b n} = n^{\log_b a}. \tag{14.1}$$

The summation is a geometric series whose sum depends on the ratio $r = b^k/a$. There are three cases.

1. $r < 1$. From Equation 2.4,

$$\sum_{i=0}^{m} r^i < 1/(1-r), \text{a constant}.$$

Thus,

$$\mathbf{T}(n) = \Theta(a^m) = \Theta(n^{log_b a}).$$

2. $r = 1$. Since $r = b^k/a$, we know that $a = b^k$. From the definition of logarithms it follows immediately that $k = \log_b a$. We also note from Equation 14.1 that $m = \log_b n$. Thus,

$$\sum_{i=0}^{m} r = m + 1 = \log_b n + 1.$$

Since $a^m = n\log_b a = n^k$, we have

$$\mathbf{T}(n) = \Theta(n^{\log_b a}\log n) = \Theta(n^k \log n).$$

3. $r > 1$. From Equation 2.6,

$$\sum_{i=0}^{m} r = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m).$$

Thus,

$$\mathbf{T}(n) = \Theta(a^m r^m) = \Theta(a^m (b^k/a)^m) = \Theta(b^{km}) = \Theta(n^k).$$

We can summarize the above derivation as the following theorem.

Theorem 14.1

$$\mathbf{T}(n) = \begin{cases} \Theta(n^{\log_b a}) & \textit{if } a > b^k \\ \Theta(n^k \log n) & \textit{if } a = b^k \\ \Theta(n^k) & \textit{if } a < b^k. \end{cases}$$

This theorem may be applied whenever appropriate, rather than rederiving the solution for the recurrence. For example, apply the theorem to solve

$$\mathbf{T}(n) = 3\mathbf{T}(n/5) + 8n^2.$$

Since $a = 3$, $b = 5$, $c = 8$, and $k = 2$, we find that $3 < 5^2$. Applying case (3) of the theorem, $\mathbf{T}(n) = \Theta(n^2)$.

As another example, use the theorem to solve the recurrence relation for Mergesort:

$$\mathbf{T}(n) = 2\mathbf{T}(n/2) + n; \quad \mathbf{T}(1) = 1.$$

Since $a = 2$, $b = 2$, $c = 1$, and $k = 1$, we find that $2 = 2^1$. Applying case (2) of the theorem, $\mathbf{T}(n) = \Theta(n \log n)$.

14.2.4 Average-Case Analysis of Quicksort

In Section 7.4, we determined that the average-case analysis of Quicksort had the following recurrence:

$$\mathbf{T}(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}[\mathbf{T}(k) + \mathbf{T}(n-1-k)], \qquad \mathbf{T}(0) = \mathbf{T}(1) = c.$$

The cn term is an upper bound on the `findpivot` and `partition` steps. This equation comes from observing that each element k is equally likely to be the partitioning element. It can be simplified by observing that the two recurrence terms

$\mathbf{T}(k)$ and $\mathbf{T}(n-1-k)$ are equivalent, since one simply counts up from $T(0)$ to $T(n-1)$ while the other counts down from $T(n-1)$ to $T(0)$. This yields

$$\mathbf{T}(n) = cn + \frac{2}{n}\sum_{k=0}^{n-1}\mathbf{T}(k).$$

This form is known as a recurrence with **full history**. The key to solving such a recurrence is to cancel out the summation terms. The shifting method for summations provides a way to do this. Multiply both sides by n and subtract the result from the formula for $n\mathbf{T}(n+1)$:

$$\begin{aligned} n\mathbf{T}(n) &= cn^2 + 2\sum_{k=1}^{n-1}\mathbf{T}(k) \\ (n+1)\mathbf{T}(n+1) &= c(n+1)^2 + 2\sum_{k=1}^{n}\mathbf{T}(k). \end{aligned}$$

Subtracting $n\mathbf{T}(n)$ from both sides yields:

$$\begin{aligned} (n+1)\mathbf{T}(n+1) - n\mathbf{T}(n) &= c(n+1)^2 - cn^2 + 2\mathbf{T}(n) \\ (n+1)\mathbf{T}(n+1) - n\mathbf{T}(n) &= c(2n+1) + 2\mathbf{T}(n) \\ (n+1)\mathbf{T}(n+1) &= c(2n+1) + (n+2)\mathbf{T}(n) \\ \mathbf{T}(n+1) &= \frac{c(2n+1)}{n+1} + \frac{n+2}{n+1}\mathbf{T}(n). \end{aligned}$$

At this point, we have eliminated the summation and can now use our normal methods for solving recurrences to get a closed-form solution. Note that $\frac{c(2n+1)}{n+1} < 2c$, so we can simplify the result. Expanding the recurrence, we get

$$\begin{aligned} \mathbf{T}(n+1) &\leq 2c + \frac{n+2}{n+1}\mathbf{T}(n) \\ &= 2c + \frac{n+2}{n+1}\left(2c + \frac{n+1}{n}\mathbf{T}(n-1)\right) \\ &= 2c + \frac{n+2}{n+1}\left(2c + \frac{n+1}{n}\left(2c + \frac{n}{n-1}\mathbf{T}(n-2)\right)\right) \\ &= 2c + \frac{n+2}{n+1}\left(2c + \cdots + \frac{4}{3}(2c + \frac{3}{2}\mathbf{T}(1))\right) \\ &= 2c\left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1}\frac{n+1}{n} + \cdots + \frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right) \end{aligned}$$

$$
\begin{aligned}
&= 2c\left(1 + (n+2)\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2}\right)\right) \\
&= 2c + 2c(n+2)\left(\mathcal{H}_{n+1} - 1\right)
\end{aligned}
$$

for $\mathcal{H}_{n+1}$, the Harmonic Series. From Equation 2.10, $\mathcal{H}_{n+1} = \Theta(\log n)$, so the final solution is $\Theta(n \log n)$.

14.3 Amortized Analysis

This section presents the concept of **amortized analysis**, which is the analysis for a series of operations. In particular, amortized analysis allows us to deal with the situation where the worst-case cost for n operations is less than n times the worst-case cost of any one operation. Rather than focusing on the individual cost of each operation independently and summing them, amortized analysis looks at the cost of the entire series and "charges" each individual operation with a share of the total cost.

We can apply the technique of amortized analysis in the case of a series of sequential searches in an unsorted array. For n random searches, the average-case cost for each search is $n/2$, and so the *expected* total cost for the series is $n^2/2$. Unfortunately, in the worst case all of the searches would be to the last item in the array. In this case, each search costs n for a total worst-case cost of n^2. Compare this to the cost for a series of n searches such that each item in the array is searched for precisely once. In this situation, some of the searches *must* be expensive, but also some searches *must* be cheap. The total number of searches, in the best, average, and worst case, for this problem must be $\sum_{i=i}^{n} i \approx n^2/2$. This is a factor of two better than the more pessimistic analysis that charges each operation in the series with its worst-case cost.

As another example of amortized analysis, consider the process of incrementing a binary counter. The algorithm is to move from the lower-order (rightmost) bit toward the high-order (leftmost) bit, changing 1s to 0s until the first 0 is encountered. This 0 is changed to a 1, and the increment operation is done. Below is **C**++ code to implement the increment operation, assuming that a binary number of length n is stored in array A of length n.

```
for (i=0; ((i<n) && (A[i] == 1)); i++)
  A[i] = 0;
if (i < n)
  A[i] = 1;
```

If we count from 0 through $2^n - 1$, (requiring a counter with at least n bits), what is the average cost for an increment operation in terms of the number of bits processed? Naive worst-case analysis says that if all n bits are 1 (except for the high-order bit), then n bits need to be processed. Thus, if there are 2^n increments, then the cost is $n2^n$. However, this is much too high, since it is rare for so many bits to be processed. In fact, half of the time the low-order bit is 0, and so only that bit is processed. One quarter of the time, the low-order two bits are 01, and so only the low-order two bits are processed. Another way to view this is that the low-order bit is always flipped, the bit to its left is flipped half the time, the next bit one quarter of the time, and so on. We can capture this with the summation (charging costs to bits going from right to left)

$$\sum_{i=0}^{n-1} \frac{1}{2^i} < 2.$$

In other words, the average number of bits flipped on each increment is 2, leading to a total cost of only $2 \cdot 2^n$ for a series of 2^n increments.

A useful concept for amortized analysis is illustrated by a simple variation on the stack data structure, where the `pop` function is slightly modified to take a second parameter k indicating that k pop operations are to be performed. This revised pop function, called `multipop`, might look as follows:

```
// pop k elements from stack
void Stack::multipop(int k);
```

The "local" worst-case analysis for `multipop` is $\Theta(n)$ for n elements in the stack. Thus, if there are m_1 calls to `push` and m_2 calls to `multipop`, then the naive worst-case cost for the series of operation is $m_1 + m_2 \cdot n = m_1 + m_2 \cdot m_1$. This analysis is unreasonably pessimistic. Clearly it is not really possible to pop m_1 elements each time `multipop` is called. Analysis that focuses on single operations cannot deal with this global limit, and so we turn to amortized analysis to model the entire series of operations.

The key to an amortized analysis of this problem lies in the concept of **potential**. At any given time, a certain number of items may be on the stack. The cost for `multipop` can be no more than this number of items. Each call to `push` places another item on the stack, which can be removed by only a single `multipop` operation. Thus, each call to `push` raises the potential of the stack by one item. The sum of costs for all calls to `multipop` can never be more than the total potential of the stack (aside from a constant time cost associated with each call to `multipop` itself).

The amortized cost for any series of `push` and `multipop` operations is the sum of three costs. First, each of the `push` operations takes constant time. Second, each `multipop` operation takes a constant time in overhead, regardless of the number of items popped on that call. Finally, we count the sum of the potentials expended by all `multipop` operations, which is at most m_1, the number of `push` operations. This total cost can therefore be expressed as

$$m_1 + (m_2 + m_1) = \Theta(m_1 + m_2).$$

Our final example uses amortized analysis to prove a relationship between the cost of the move-to-front self-organizing list heuristic from Section 9.2 and the cost for the optimal static ordering of the list.

Recall that, for a series of search operations, the minimum cost for a static list results when the list is sorted by frequency of access to its records. This is the optimal ordering for the records if we never allow the positions of records to change, since the most frequently accessed record is first (and thus has least cost), followed by the next most frequently accessed record, and so on.

Theorem 14.2 *The total number of comparisons required by any series S of n or more searches on a self-organizing list of length n using the move-to-front heuristic is never more than twice the total number of comparisons required when series S is applied to the list stored in its optimal static order.*

Proof: Each comparison of the search key with a record in the list is either successful or unsuccessful. For m searches, there must be exactly m successful comparisons for both the self-organizing list and the static list. The total number of unsuccessful comparisons in the self-organizing list is the sum, over all pairs of distinct keys, of the number of unsuccessful comparisons made between that pair.

Consider a particular pair of keys *A* and *B*. For any sequence of searches *S*, the total number of (unsuccessful) comparisons between *A* and *B* is identical to the number of comparisons between *A* and *B* required for the subsequence of *S* made up only of searches for *A* or *B*. Call this subsequence S_{AB}. In other words, including searches for other keys does not affect the relative position of *A* and *B* and so does not affect the relative contribution to the total cost of the unsuccessful comparisons between *A* and *B*.

The number of unsuccessful comparisons between *A* and *B* made by the move-to-front heuristic on subsequence S_{AB} is at most twice the number of unsuccessful comparisons between *A* and *B* required when S_{AB} is applied to the optimal static ordering for the list. To see this, assume that S_{AB} contains i *A*s and j *B*s, with $i \leq j$. Under the optimal static ordering, i unsuccessful comparisons are required since *B* must appear before *A* in the list (since its access frequency is higher). Move-to-front will yield an unsuccessful comparison whenever the request sequence changes from *A* to *B* or from *B* to *A*. The total number of such changes possible is $2i$ since each change involves an *A* and each *A* can be part of at most two changes.

Since the total number of unsuccessful comparisons required by move-to-front for any given pair of keys is at most twice that required by the optimal static ordering, the total number of unsuccessful comparisons required by move-to-front for all pairs of keys is also at most twice as high. Since the number of successful comparisons is the same for both methods, the total number of comparisons required by move-to-front is less than twice the number of comparisons required by the optimal static ordering. □

14.4 Further Reading

A good introduction to solving recurrence relations is *Applied Combinatorics* by Fred S. Roberts [Rob84]. For a more advanced treatment, see *Concrete Mathematics* by Graham, Knuth, and Patashnik [GKP89].

Cormen, Leiserson, and Rivest provide a good discussion on various methods of performing amortized analysis in *Introduction to Algorithms* [CLR90]. For an amortized analysis that the splay tree requires $m \log n$ time to perform a series of m operations on n nodes when $m > n$, see "Self-Adjusting Binary Search Trees" by Sleator and Tarjan [ST85]. The proof for Theorem 14.2 comes from "Amortized Analysis of Self-Organizing Sequential Search Heuristics" by Bentley and McGeoch [BM85].

14.5 Exercises

14.1 Use the technique of guessing a polynomial and deriving the coefficients to solve the summation
$$\sum_{i=1}^{n} i^2.$$

14.2 Use the technique of guessing a polynomial and deriving the coefficients to solve the summation
$$\sum_{i=1}^{n} i^3.$$

14.3 Find, and prove correct, a closed-form solution for
$$\sum_{i=a}^{b} i^2.$$

14.4 Use the shifting method to solve the summation

$$\sum_{i=1}^{n} i.$$

14.5 Use the shifting method to solve the summation

$$\sum_{i=1}^{n} 2^i.$$

14.6 Use the shifting method to solve the summation

$$\sum_{i=1}^{n} i2^{n-i}.$$

14.7 Prove that the number of moves required for function TOH from Section 2.4 is $2^n - 1$.

14.8 Give and prove the closed-form solution for the recurrence relation $\mathbf{T}(n) = \mathbf{T}(n-1) + c$, $\mathbf{T}(1) = c$.

14.9 Prove by induction that the closed-form solution for the recurrence relation

$$\mathbf{T}(n) = 2\mathbf{T}(n/2) + n; \quad \mathbf{T}(2) = 1$$

is in $\Omega(n \log n)$.

14.10 Find the solution (in asymptotic terms, not precise constants) for the recurrence relation

$$\mathbf{T}(n) = \mathbf{T}(n/2) + \sqrt{n}; \quad \mathbf{T}(1) = 1.$$

14.11 Using the technique of expanding the recurrence, find the exact closed-form solution for the recurrence relation

$$\mathbf{T}(n) = 2\mathbf{T}(n/2) + n; \quad \mathbf{T}(2) = 2.$$

14.12 Use Theorem 14.1 to prove that binary search requires $\Theta(\log n)$ time.

14.13 Recall that when a hash table gets to be more than about one half full, its performance quickly degrades. One solution to this problem is to reinsert all elements of the hash table into a new hash table that is twice as large. Assuming that the (expected) average case cost to insert into a hash table is $\Theta(1)$, prove that the average cost to insert is still $\Theta(1)$ when this reinsertion policy is used.

14.14 The standard algorithm for multiplying two $n \times n$ matrices requires $\Theta(n^3)$ time. It is possible to do better than this by rearranging and grouping the multiplications in various ways. One example of this is known as Strassen's matrix multiplication algorithm. Assume that n is a power of two. In the following, *A* and *B* are $n \times n$ arrays, while A_{ij} and B_{ij} refer to arrays of size $n/2 \times n/2$. Strassen's algorithm is to multiply the subarrays together in a particular order, as expressed by the following equation:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{bmatrix}.$$

In other words, the result of the multiplication for an $n \times n$ array is obtained by a series of matrix multiplications and additions for $n/2 \times n/2$ arrays. Multiplications between subarrays also use Strassen's algorithm, and the addition of two subarrays requires $\Theta(n^2)$ time. The subfactors are defined as follows:

$$\begin{aligned} s_1 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\ s_2 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ s_3 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \\ s_4 &= (A_{11} + A_{12}) \cdot B_{22} \\ s_5 &= A_{11} \cdot (B_{12} - B_{22}) \\ s_6 &= A_{22} \cdot (B_{21} - B_{11}) \\ s_7 &= (A_{21} + A_{22}) \cdot B_{11}. \end{aligned}$$

(a) Show that Strassen's algorithm is correct.

(b) How many multiplications of subarrays and how many additions are required by Strassen's algorithm? How many would be required by normal matrix multiplication if it were defined in terms of subarrays in the same manner? Show the recurrence relations for both Strassen's algorithm and the normal matrix multiplication algorithm.

(c) Derive the closed-form solution for the recurrence relation you gave for Strassen's algorithm (use Theorem 14.1).

(d) Give your opinion on the practicality of Strassen's algorithm.

14.15 Given a 2-3 tree with N nodes, prove that inserting M additional nodes requires $O(M + N)$ node splits.

14.16 One approach to implementing an array-based list where the list size is unknown is to let the array grow and shrink. This is known as a **dynamic array**.

When necessary, we can grow or shrink the array by copying the array's contents to a new array. If we are careful about the size of the new array, this copy operation can be done rarely enough so as not to affect the amortized cost of the operations.

(a) What is the amortized cost of inserting elements into the list if the array is initially of size 1 and we double the array size whenever the number of elements that we wish to store exceeds the size of the array? Assume that the insert itself cost $O(1)$ time per operation and so we are just concerned with minimizing the copy time to the new array.

(b) Consider an underflow strategy that cuts the array size in half whenever the array falls below half full. Give an example where this strategy leads to a bad amortized cost. Again, we are only interested in measuring the time of the array copy operations.

(c) Give a better underflow strategy than that suggested in part (b). Your goal is to find a strategy whose amortized analysis shows that array copy requires $O(n)$ time for a series of n operations.

14.17 Recall that two vertices in an undirected graph are in the same connected component if there is a path connecting them. A good algorithm to find the connected components of an undirected graph begins by calling a DFS on the first vertex. All vertices reached by the DFS are in the same connected component and are so marked. We then look through the vertex `mark` array until an unmarked vertex i is found. Again calling the DFS on i, all vertices reachable from i are in a second connected component. We continue working through the `mark` array until all vertices have been assigned to some connected component. A sketch of the algorithm is as follows:

```
for (i=0; i<G->n(); i++) // For n vertices in the graph
  G->setMark(i, 0);   // Vertices start in no component
compcount = 1;        // Counter for current component
for (i=0; i<G->n(); i++)
  if (G->getMark(i) == 0) { // Start a new component
    DFS_component(G, i, compcount);
    compcount++;
  }

void DFS_component(Graph* G, int v, int compcount) {
  G->setMark(v, compcount);
  for (int w=G->first(v); w<G->n(); w = G->next(v,w))
    if (G->getMark(w) == 0)
      DFS_component(G, w, compcount);
}
```

Use the concept of potential from amortized analysis to explain why the total cost of this algorithm is $\Theta(|V| + |E|)$. (Note that this will not be a true amortized analysis since this algorithm does not allow an arbitrary series of DFS operations but rather is fixed to do a single call to DFS from each vertex.)

14.18 Give a proof similar to that used for Theorem 14.2 to show that the total number of comparisons required by any series of n or more searches *S* on a self-organizing list of length n using the count heuristic is never more than twice the total number of comparisons required when series *S* is applied to the list stored in its optimal static order.

14.6 Projects

14.1 Implement the UNION/FIND algorithm of Section 6.2 using both path compression and the weighted union rule. Count the total number of node accesses required for various series of equivalences to determine if the actual performance of the algorithm matches the expected cost of $\Theta(n \log^* n)$.

14.2 Implement both a standard $\Theta(n^3)$ matrix multiplication algorithm and Strassen's matrix multiplication algorithm (see Exercise 14.14). Using empirical testing, try to estimate the constant factors for the runtime equations of the two algorithms. How big must n be before Strassen's algorithm becomes more efficient than the standard algorithm?

15 Limits to Computation

This book contains many examples of data structures used to solve a wide variety of problems. There are also many examples of efficient algorithms. In general, our search algorithms strive to be at worst in $\mathrm{O}(\log n)$, while our sorting algorithms strive to be in $\mathrm{O}(n \log n)$. A few algorithms, such as the all-pairs shortest-paths algorithms, have higher asymptotic complexity, with Floyd's all-pairs shortest-paths algorithm being $\Theta(n^3)$.

Part of the reason why we can solve many problems efficiently has to do with the fact that we use efficient algorithms. Given any problem for which you know *some* algorithm, it is always possible to write an inefficient algorithm to "solve" the problem. For example, consider a sorting algorithm that tests every possible permutation of its input until it finds the correct permutation that provides a sorted list. The running time for this algorithm would be unacceptably high, since it is proportional to the number of permutations which is $n!$ for n inputs. When solving the minimum-cost spanning tree problem, if we were to test every possible subset of edges to see which forms the shortest minimum spanning tree, the amount of work would be proportional to $2^{|\mathrm{E}|}$ for a graph with $|\mathrm{E}|$ edges. Fortunately, for both of these problems we have more clever algorithms that allow us to find answers (relatively) quickly.

Unfortunately, in real life there are many computing problems that must be solved for which the best possible algorithm takes a long time. A simple example is the Towers of Hanoi problem, which requires 2^n moves to "solve" a tower with n disks. It is not possible for any computer program that solves the Towers of Hanoi problem to run in less than $\Omega(2^n)$ time, since that many moves must be printed out.

Besides those problems whose solutions *must* take a long time to run, there are also many problems for which we simply do not know if there are efficient algorithms or not. The best algorithms that we know for such problems are very slow, but perhaps there are better ones waiting to be discovered. Of course, while having a problem with high running time is bad, it is even worse to have a problem that cannot be solved at all! Problems of the later type do exist, and some are presented in Section 15.3.

This chapter presents a brief introduction to the theory of expensive and impossible problems. Section 15.1 presents the concept of a reduction, which is the central tool used for analyzing the difficulty of a problem (as opposed to analyzing an algorithm). Reductions allow us to relate the difficulty of various problems, which is often much easier than doing the analysis for a problem from first principles. Section 15.2 discusses "hard" problems, by which we mean problems that require, or at least appear to require, time exponential on the problem size. Finally, Section 15.3 considers various problems that, while often simple to define and comprehend, are in fact impossible to solve using a computer program. The classic example of such a problem is deciding whether an arbitrary computer program will go into an infinite loop when processing a specified input. This is known as the **halting problem**.

15.1 Reductions

We begin with an important concept for understanding the relationships between problems, called **reduction**. Reduction allows us to solve one problem in terms of another. Equally importantly, when we wish to understand the difficulty of a problem, reduction allows us to make relative statements about upper and lower bounds on the cost of a problem (as opposed to an algorithm or program).

Since the concept of problems is discussed extensively in this chapter, we begin with notation to simplify problem descriptions. Throughout this chapter, a problem will be defined in terms of a mapping between inputs and outputs, and the name of the problem will be given in all capital letters. Thus, a complete definition of the sorting problem could appear as follows:

SORTING:
Input: A sequence of integers $x_0, x_1, x_2, ..., x_{n-1}$.
Output: A permutation $y_0, y_1, y_2, ..., y_{n-1}$ of the sequence such that $y_i \leq y_j$ whenever $i < j$.

Once you have bought or written a program to solve one problem, such as sorting, you may be able to use it as a tool to solve a different problem. This is

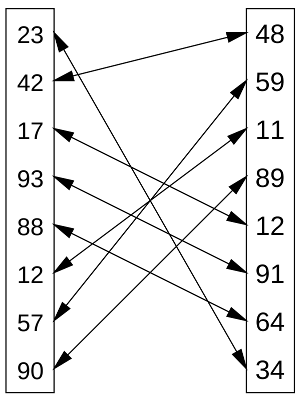

Figure 15.1 An illustration of PAIRING. The two lists of numbers are paired up so that the least values from each list make a pair, the next smallest values from each list make a pair, and so on.

known in software engineering as **software reuse**. To illustrate this, let us consider another problem.

PAIRING:
Input: Two sequences of integers $X = (x_0, x_1, ..., x_{n-1})$ and $Y = (y_0, y_1, ..., y_{n-1})$.
Output: A pairing of the elements in the two sequences such that the least value in X is paired with the least value in Y, the next least value in X is paired with the next least value in Y, and so on.

Figure 15.1 illustrates PAIRING. One way to solve PAIRING is to use an existing sorting program by sorting each of the two sequences, and then pairing-off items based on their position in sorted order. Technically, we say that PAIRING is **reduced** to SORTING, since SORTING is used to solve PAIRING.

Notice that reduction is a three-step process. The first step is to convert an instance of PAIRING into two instances of SORTING. The conversion step is not very interesting; it simply takes each sequence and assigns it to an array to be passed to SORTING. The second step is to sort the two arrays (i.e., apply SORTING to each array). The third step is to convert the output of SORTING to the output for PAIRING. This is done by pairing the first elements in the sorted arrays, the second elements, and so on.

The reduction of PAIRING to SORTING helps to establish an upper bound on the cost of PAIRING. In terms of asymptotic notation, assuming that we can find one method to convert the inputs to PAIRING into inputs to SORTING "fast enough," and a second method to convert the result of SORTING back to the correct result for PAIRING "fast enough," then the asymptotic cost of PAIRING cannot be more than the cost of SORTING. In this case, there is little work to be done to convert from PAIRING to SORTING, or to convert the answer from SORTING back to the answer for PAIRING, so the dominant cost of this solution is performing the sort operation. Thus, an upper bound for PAIRING is in $O(n \log n)$.

It is important to note that the pairing problem does *not* require that elements of the two sequences be sorted. This is merely one possible way to solve the problem. PAIRING only requires that the elements of the sequences be paired correctly. Perhaps there is another way to do it? Certainly if we use sorting to solve PAIRING, the algorithms will require $\Omega(n \log n)$ time. But, another approach might conceivably be faster.

There is another use of reductions aside from using an old algorithm to solve a new problem (and coincidentally establishing an upper bound for the new problem). That is to prove a lower bound on the cost of a new problem by showing that it could be used as a solution for an old problem with a known lower bound.

Assume we can go the other way and convert SORTING to PAIRING "fast enough." What does this say about the minimum cost of PAIRING? We know from Section 7.9 that the cost of SORTING in the worst and average cases is in $\Omega(n \log n)$. In other words, the best possible algorithm for sorting requires at least $n \log n$ time.

Assume that PAIRING could be done in $O(n)$ time. Then, one way to create a sorting algorithm would be to convert SORTING into PAIRING, run the algorithm for PAIRING, and finally convert the answer back to the answer for SORTING. Provided that we can convert SORTING to/from PAIRING "fast enough," this process would yield an $O(n)$ algorithm for sorting! Since this contradicts what we know about the lower bound for SORTING, and the only flaw in the reasoning is the initial assumption that PAIRING can be done in $O(n)$ time, we can conclude that there is no $O(n)$ time algorithm for PAIRING. This reduction process tells us that PAIRING must be at least as expensive as SORTING and so must itself have a lower bound in $\Omega(n \log n)$.

To complete this proof regarding the lower bound for PAIRING, we need now to find a way to reduce SORTING to PAIRING. This is easily done. Take an instance of SORTING (i.e., an array *A* of n elements). A second array *B* is generated that simply stores i in position i for $0 \leq i < n$. Pass the two arrays to PAIRING. Take the resulting set of pairs, and use the value from the *B* half of the pair to tell which position in the sorted array the *A* half should take; that is, we can now reorder the records in the A array using the corresponding value in the B array as the sort key and running a simple $\Theta(n)$ Binsort. The conversion of SORTING to PAIRING can be done in $O(n)$ time, and likewise the conversion of the output of PAIRING can be converted to the correct output for SORTING in $O(n)$ time. Thus, the cost of this "sorting algorithm" is dominated by the cost for PAIRING.

Consider any two problems for which a suitable reduction from one to the other can be found. The first problem takes an arbitrary instance of its input, which we will call **I**, and transforms **I** to a solution, which we will call **SLN**. The second problem takes an arbitrary instance of its input, which we will call $\mathbf{I'}$, and transforms $\mathbf{I'}$ to a solution, which we will call $\mathbf{SLN'}$. We can define reduction more formally as a three-step process:

1. Transform an arbitrary instance of the first problem to an instance of the second problem. In other words, there must be a transformation from any instance **I** of the first problem to an instance $\mathbf{I'}$ of the second problem.
2. Apply an algorithm for the second problem to the instance $\mathbf{I'}$, yielding a solution $\mathbf{SLN'}$.
3. Transform $\mathbf{SLN'}$ to the solution of **I**, known as **SLN**. Note that **SLN** must in fact be the correct solution for **I** for the reduction to be acceptable.

It is important to note that the reduction process does not give us an algorithm for solving either problem by itself. It merely gives us a method for solving the first problem given that we already have a solution to the second. More importantly for the topics to be discussed in the remainder of this chapter, reduction gives us a way to understand the bounds of one problem in terms of another. Specifically, given efficient transformations, the upper bound of the first problem is at most the upper bound of the second. Conversely, the lower bound of the second problem is at least the lower bound of the first.

As a second example of reduction, consider the simple problem of multiplying two n-digit numbers. The standard long-hand method for multiplication is to multiply the last digit of the first number by the second number (taking $\Theta(n)$ time), multiply the second digit of the first number by the second number (again taking $\Theta(n)$ time), and so on for each of the n digits of the first number. Finally, the intermediate results are added together. Note that adding two numbers of length M and N can easily be done in $\Theta(M + N)$ time. Since each digit of the first number is multiplied against each digit of the second, this algorithm requires $\Theta(n^2)$ time. Asymptotically faster (but more complicated) algorithms are known, but none is so fast as to be in $O(n)$.

Next we ask the question: Is squaring an n-digit number as difficult as multiplying two n-digit numbers? We might hope that something about this special case will allow for a faster algorithm than is required by the more general multiplication problem. However, a simple reduction proof serves to show that squaring is "as hard" as multiplying.

The key to the reduction is the following formula:

$$X \times Y = \frac{(X+Y)^2 - (X-Y)^2}{4}.$$

The significance of this formula is that it allows us to convert an arbitrary instance of multiplication to a series of operations involving three addition/subtractions (each of which can be done in linear time), two squarings, and a division by 4. Note that the division by 4 can be done in linear time (simply convert to binary, shift by two digits, and convert back).

This reduction shows that if a linear time algorithm for squaring can be found, it can be used to construct a linear time algorithm for multiplication.

An example of a useful reduction is multiplication through the use of logarithms. Multiplication is considerably more difficult than addition, since the cost to multiply two n-bit numbers directly is $\mathrm{O}(n^2)$, while addition of two n-bit numbers is $\mathrm{O}(n)$. Recall from Section 2.3 that one property of logarithms is

$$\log nm = \log n + \log m.$$

Thus, if taking logarithms and anti-logarithms were cheap, then we could reduce multiplication to addition by taking the log of the two operands, adding, and then taking the anti-log of the sum.

Under normal circumstances, taking logarithms and anti-logarithms is expensive, and so this reduction would not be considered practical. However, this reduction is precisely the basis for the slide rule. The slide rule uses a logarithmic scale to measure the lengths of two numbers, in effect doing the conversion to logarithms automatically. These two lengths are then added together, and the inverse logarithm of the sum is read off another logarithmic scale. The part normally considered expensive (taking logarithms and anti-logarithms) is cheap since it is a physical part of the slide rule. Thus, the entire multiplication process can be done cheaply via a reduction to addition.

Our next example of reduction concerns the multiplication of two $n \times n$ matrices. For this problem, we will assume that the values stored in the matrices are simple integers and that multiplying two simple integers takes constant time. The standard algorithm for multiplying two matrices is to multiply each element of the first matrix's first row by the corresponding element of the second matrix's first column, then adding the numbers. This takes $\Theta(n)$ time. Each of the n^2 elements of the solution are computed in similar fashion, requiring a total of $\Theta(n^3)$ time. Faster algorithms are known, but none are so fast as to be in $O(n^2)$.

Now, consider the case of multiplying two **symmetric** matrices. A symmetric matrix is one in which entry ij is equal to entry ji; that is, the upper-right triangle of the matrix is a mirror image of the lower-left triangle. Is there something about this restricted case that allows us to multiply two symmetric matrices faster than in the general case? The answer is no, as can be seen by the following reduction. Assume that we have been given two $n \times n$ matrices A and B. We can construct a $2n \times 2n$ symmetric matrix from an arbitrary matrix A as follows:

$$\begin{bmatrix} 0 & A \\ A^{\mathrm{T}} & 0 \end{bmatrix}.$$

Here 0 stands for an $n \times n$ matrix composed of zero values, A is the original array, and A^{T} stands for the transpose of matrix A.[1] Note that the resulting matrix is now symmetric. We can convert matrix B to a symmetric matrix in a similar manner. If symmetric matrices could be multiplied "quickly" (in particular, if they could be multiplied together in $\Theta(n^2)$ time), then we could find the result of multiplying two arbitrary $n \times n$ matrices in $\Theta(n^2)$ time by taking advantage of the following observation:

$$\begin{bmatrix} 0 & A \\ A^{\mathrm{T}} & 0 \end{bmatrix} \begin{bmatrix} 0 & B^{\mathrm{T}} \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^{\mathrm{T}}B^{\mathrm{T}} \end{bmatrix}.$$

In the above formula, AB is the result of multiplying matrices A and B together.

15.2 Hard Problems

This section discusses some really "hard" problems. There are several ways that a problem could be considered hard. First, we might have trouble understanding the definition of the problem itself. Second, we might have trouble finding or understanding an algorithm to solve a problem. Neither of these is what is commonly meant when a computer theoretician uses the word "hard." Throughout this section, "hard" means that the best-known algorithm for the problem is expensive in its running time. One example of a hard problem is Towers of Hanoi. It is simple to understand this problem and its solution. It is also simple to write a program to solve this problem. But, it takes an extremely long time to run for any "reasonably" large value of n. Try running a program to solve Towers of Hanoi for only 30 disks!

[1]The transpose operation takes position ij of the original matrix and places it in position ji of the transpose matrix. This can easily be done in n^2 time for an $n \times n$ matrix.

The Towers of Hanoi problem takes exponential time, that is, its running time is $\Theta(2^n)$. This is radically different than an algorithm that takes $\Theta(n \log n)$ time or $\Theta(n^2)$ time. It is even radically different from a problem that takes $\Theta(n^4)$ time. These are all examples of polynomial running time, since the exponents for all terms of these equations are constants. Recall from Chapter 3 that if we buy a new computer that runs twice as fast, the size of problem with complexity $\Theta(n^4)$ that we can solve in a certain amount of time is increased by the fourth root of two. In other words, there is a multiplicative factor increase, even if it is a rather small one. This is true for any algorithm whose running time can be represented by a polynomial.

Consider what happens if you buy a computer that is twice as fast and try to solve a bigger Towers of Hanoi problem in a given amount of time. Since its complexity is $\Theta(2^n)$, we can solve a problem only one disk bigger! There is no multiplicative factor, and this is true for any exponential algorithm: A constant factor increase in processing power results in only a fixed addition in problem-solving power.

For the rest of this chapter, we define a **hard algorithm** to be one that runs in exponential time, that is, in $\Omega(c^n)$ for some constant $c > 1$. A definition for a hard *problem* will be presented in the next section.

15.2.1 $\mathcal{NP}$-Completeness

Imagine a magical computer that works by guessing the correct solution from among all of the possible solutions to a problem. Another way to look at this is to imagine a super parallel computer that could test all possible solutions simultaneously. Certainly this magical computer can do anything a normal computer can do. It might also solve some problems more quickly than a normal computer can. Consider some problem where, given a guess for a solution, checking the solution to see if it is correct can be done in polynomial time. Even if the number of possible solutions is exponential, any given guess can be checked in polynomial time (equivalently, all possible solutions are checked simultaneously in polynomial time), and thus the problem can be solved in polynomial time. Conversely, if you cannot get the answer to a problem in polynomial time by guessing the right answer and then checking it, you cannot do it in polynomial time in any other way.

The idea of "guessing" the right answer to a problem – or checking all possible solutions in parallel to determine which is correct – is called **non-determinism**. An algorithm that works in this manner is called a **non-deterministic algorithm**, and any problem with an algorithm that runs on a non-deterministic machine in polynomial time is given a special name: It is said to be a problem in $\mathcal{NP}$. Thus,

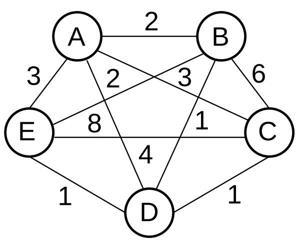

Figure 15.2 An illustration of the TRAVELING SALESMAN problem. Five vertices are shown, with edges between each pair of cities. The problem is to visit all of the cities exactly once, returning to the start city, with the least total cost.

problems in $\mathcal{NP}$ are those problems that can be solved in polynomial time on a non-deterministic machine.

Not all problems requiring exponential time on a regular computer are in $\mathcal{NP}$. For example, Towers of Hanoi is *not* in $\mathcal{NP}$, since it must print out $O(2^n)$ moves for n disks. A non-deterministic machine cannot "guess" and print the correct answer in polynomial time.

On the other hand, consider that is commonly known as the Traveling Salesman problem.

TRAVELING SALESMAN (1)
Input: A complete, directed graph **G** with distances assigned to each edge in the graph.
Output: The shortest simple cycle that includes every vertex.

Figure 15.2 illustrates this problem. Five vertices are shown, with edges and associated costs between each pair of edges. (For simplicity, we assume that the cost is the same in both directions, though this need not be the case.) If the salesman visits the cities in the order ABCDEA, he will travel a total distance of 13. A better route would be ABDCEA, with cost 11. The best route for this particular graph would be ABEDCA, with cost 9.

We cannot solve this problem in polynomial time with a non-deterministic computer. The problem is that, given a candidate cycle, while we can quickly check that the answer is a cycle of the appropriate form, we have no easy way of knowing if it is in fact the *shortest* such cycle. However, we can solve a variant of this problem, which is in the form of a **decision problem**. A decision problem is simply one whose answer is either YES or NO. The decision problem form of TRAVELING SALESMAN is as follows:

TRAVELING SALESMAN (2)

Input: A complete, directed graph **G** with distances assigned to each edge in the graph, and an integer K.

Output: YES if there is a simple cycle with total distance $\leq K$ containing every vertex in **G**, and NO otherwise.

We can solve this version of the problem in polynomial time with a non-deterministic computer. The non-deterministic algorithm simply checks all of the possible subsets of edges in the graph, in parallel. If any subset of the edges is an appropriate cycle of total length less than or equal to K, the answer is YES; otherwise the answer is NO. Note that it is only necessary that *some* subset meet the requirement; it does not matter how many subsets fail. Checking a particular subset is done in polynomial time by adding the distances of the edges and verifying that the edges form a cycle that visits each vertex exactly once. Thus, the checking algorithm runs in polynomial time. Unfortunately, there are $|\mathrm{E}|!$ subsets to check, so this algorithm cannot be converted to a polynomial time algorithm on a regular computer. Nor does anybody in the world know of any other polynomial time algorithm to solve TRAVELING SALESMAN on a regular computer, despite the fact that the problem has been studied extensively by many computer scientists for many years.

It turns out that there is a large collection of problems with this property: We know efficient non-deterministic algorithms, but we do not know if there are efficient deterministic algorithms. At the same time, we cannot prove that any of these problems do *not* have efficient deterministic algorithms. This class of problems is called **$\mathcal{NP}$-complete**. What is truly strange and fascinating about $\mathcal{NP}$-complete problems is that if anybody ever finds the solution to any one of them that runs in polynomial time on a regular computer, then by a series of reductions, every other problem that is in $\mathcal{NP}$ can also be solved in polynomial time on a regular computer!

A problem *X* is defined to be $\mathcal{NP}$-complete if

1. *X* is in $\mathcal{NP}$, and
2. Every other problem in $\mathcal{NP}$ can be reduced to *X* in polynomial time.

This second requirement may seem to be impossible, but in fact there are hundreds of such problems, including TRAVELING SALESMAN. Another such problem is called CLIQUE. CLIQUE asks, given an arbitrary undirected graph **G**, if there is a complete subgraph of at least k vertices. Nobody knows whether there is a polynomial time solution for CLIQUE, but if such an algorithm is found for

CLIQUE *or* for TRAVELING SALESMAN, then that solution can be modified to solve the other, or any other problem in $\mathcal{NP}$, in polynomial time.

The primary theoretical advantage of knowing that a problem P1 is $\mathcal{NP}$-complete is that it can be used to show that another problem P2 is $\mathcal{NP}$-complete. This is done by finding a polynomial time reduction of P1 to P2. Since we already know that all problems in $\mathcal{NP}$ can be reduced to P1 in polynomial time (by the definition of $\mathcal{NP}$-complete), we now know that all problems can be reduced to P2 as well by the simple algorithm of reducing to P1 and then from there reducing to P2.

There is an extremely practical advantage to knowing that a problem is $\mathcal{NP}$-complete. It relates to knowing that if a polynomial time solution can be found for *any* problem that is $\mathcal{NP}$-complete, then a polynomial solution can be found for *all* such problems. The implication is that,

1. Since no one has yet found such a solution, it must be difficult or impossible to do; and
2. Effort to find a polynomial time solution for one $\mathcal{NP}$-complete problem can be considered to have been expended for all $\mathcal{NP}$-complete problems.

How is $\mathcal{NP}$-completeness of practical significance for typical programmers? Well, if your boss demands that you provide a fast algorithm to solve a problem, she will not be happy if you come back saying that the best you could do was an exponential time algorithm. But, if you can find that the problem is $\mathcal{NP}$-complete, while she still won't be happy, at least she should not be mad at you! By showing that your problem is $\mathcal{NP}$-complete, you are in effect saying that the most brilliant computer scientists for the last 40 years or more have been trying and failing to find a polynomial time algorithm for your problem.

Problems that are solvable in polynomial time on a regular computer are said to be in class $\mathcal{P}$. Clearly, all problems in $\mathcal{P}$ are solvable in polynomial time on a non-deterministic computer simply by neglecting to use the non-deterministic capability. Some problems in $\mathcal{NP}$ are $\mathcal{NP}$-complete. We can consider all problems solvable in exponential time or better as an even bigger class of problems since all problems solvable in polynomial time are solvable in exponential time. Thus, we can view the world of exponential-time-or-better problems in terms of Figure 15.3.

The most important unanswered question in theoretical computer science is whether $\mathcal{P} = \mathcal{NP}$. If they are equal, then there is a polynomial time algorithm for TRAVELING SALESMAN and all related problems. Since TRAVELING SALESMAN is known to be $\mathcal{NP}$-complete, if a polynomial time algorithm were to be found for this problem, then *all* problems in $\mathcal{NP}$ would also be solvable in polynomial time. Conversely, if we were able to prove that TRAVELING SALESMAN has an exponential time lower bound, then we would know that $\mathcal{P} \neq \mathcal{NP}$.

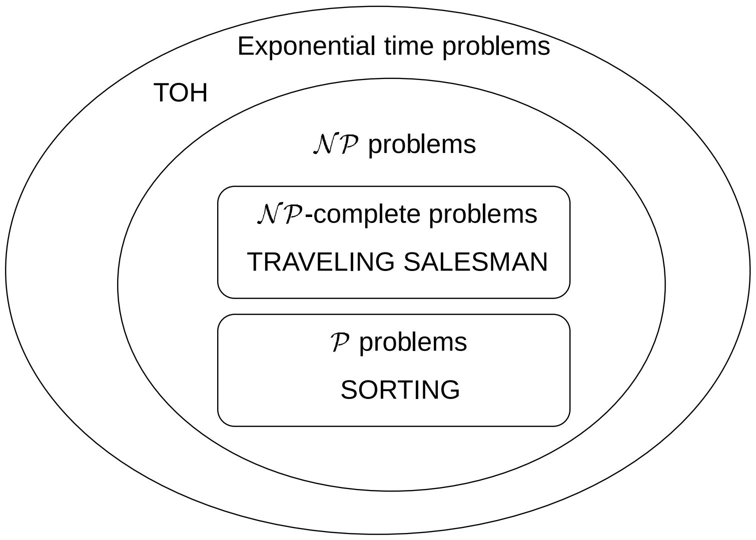

Figure 15.3 Our knowledge regarding the world of problems requiring exponential time or less. Some of these problems are solvable in polynomial time by a non-deterministic computer. Of these, some are known to be $\mathcal{NP}$-complete, and some are known to be solvable in polynomial time on a regular computer.

The following example provides a model for how a complete $\mathcal{NP}$-completeness proof is done. The fundamental idea is to take some problem known to already be $\mathcal{NP}$-complete, and use a reduction argument to show that the problem in question must also be $\mathcal{NP}$-complete because, if it were doable in polynomial time, it could be used to solve the known $\mathcal{NP}$-complete problem in polynomial time. First we define two problems that we can use for the example.

VERTEX COVER

Input: A graph **G** and an integer k.

Output: YES if there is a subset **S** of the vertices in **G** of size k or less such that every edge of **G** has at least one of its endpoints in **S**, and NO otherwise.

CLIQUE

Input: A graph **G** and an integer k.

Output: YES if there is a subset **S** of the vertices in **G** of size k or greater such that **S** is a complete graph, and NO otherwise.

Example 15.1 Theorem 15.1 *VERTEX COVER is $\mathcal{NP}$-complete.*

Proof: To prove that a problem is $\mathcal{NP}$-complete, we must do two things. The first is to prove that the problem can be solved in polynomial time by a non-deterministic algorithm. The second is to give a reduction from a problem known already to be $\mathcal{NP}$-complete to the problem we are trying to show is $\mathcal{NP}$-complete.[2] For the first step, we can easily write a non-deterministic algorithm that simply "guesses" a subset of the graph and determines in polynomial time whether that subset is in fact a vertex cover of size k or less. For the second step, we will assume that CLIQUE is already known to be $\mathcal{NP}$-complete (the proof for this is beyond the scope of this chapter, but is typically given in any more advanced treatment of the topic).

Given that CLIQUE is $\mathcal{NP}$-complete, we need to find a polynomial-time transformation from the input to CLIQUE to the input to VERTEX COVER, and another polynomial-time transformation from the output for VERTEX COVER to the output for CLIQUE. This turns out to be a simple matter, given the following observation. Consider a graph **G** and a vertex cover **S** on **G**. Denote by **S′** the set of vertices in **G** but not in **S**. There can be no edge connecting any two vertices in **S′** because, if there were, then **S** would not be a vertex cover. Denote by **G′** the inverse graph for **G**, that is, the graph formed from the edges **not** in **G**. If **S** is of size k, then **S′** forms a clique of size $n - k$ in graph **G′**. Thus, we can reduce CLIQUE to VERTEX COVER simply by converting graph **G** to **G′**, and asking if **G′** has a VERTEX COVER of size $n - k$ or smaller. If YES, then there is a clique in **G** of size k; if NO then there is not. □

15.2.2 Getting Around $\mathcal{NP}$-Complete Problems

Unfortunately, finding that your problem is $\mathcal{NP}$-complete may not mean that you can just forget about it. Traveling salesmen need to find reasonable sales routes regardless of the complexity of the problem. What do you do when faced with an $\mathcal{NP}$-complete problem that you must solve?

There are several techniques to try. One approach is to run only small instances of the problem. For some problems, this is not acceptable. For example, TRAVELING SALESMAN grows so quickly that it cannot be run on modern computers for

[2]This formulation for proving a problem is $\mathcal{NP}$-complete is not precisely correct, but it is close enough to understand the process.

problem sizes much over 20 cities. However, some other problems in $\mathcal{NP}$, while requiring exponential time, still grow slowly enough that they allow solutions for problems of a useful size. One such example is the KNAPSACK problem. Given a set of items each with given size and each with given value, and a knapsack of size k, is there a subset of the items whose total size is less than or equal to k and whose total value is greater than or equal to v? While this problem is $\mathcal{NP}$-complete, and so the best-known solution requires exponential running time, it is still solvable for dozens of items with k and v in the thousands.

A second approach to handling $\mathcal{NP}$-complete problems is to solve a special instance of the problem that is not so hard. For example, many problems on graphs are $\mathcal{NP}$-complete, but the same problem on certain restricted types of graphs is not so difficult. For example, while the VERTEX COVER problem is $\mathcal{NP}$-complete in general, there is a polynomial time solution for bipartite graphs (i.e., graphs whose vertices can be separated into two subsets such that no pair of vertices within one of the subsets has an edge between them).

A third approach is to find an approximate solution to the problem. There are a number of approaches to finding approximate solutions. One way is to use a heuristic to solve the problem, that is, an algorithm based on a "rule of thumb" that does not always give the correct answer. For example, the TRAVELING SALESMAN problem can be solved approximately by using the heuristic that we start at an arbitrary city and then always proceed to the next unvisited city that is closest. This rarely gives the shortest path, but the solution may be good enough. There are many other heuristics for TRAVELING SALESMAN that do a better job. For some problems, an approximation algorithm can give guaranteed performance, perhaps that the answer will be within a certain percentage of the best possible answer.

15.3 Impossible Problems

Every day professional programmers write programs that go into an infinite loop. Of course, when a program is in an infinite loop, you do not know for sure if it is just a slow program or a program in an infinite loop. After "enough time," you shut it down. Wouldn't it be great if your compiler could look at your program and tell you before you run it that it might get into an infinite loop? Alternatively, given a program and a particular input, it would be useful to know if executing the program on that input will result in an infinite loop without actually running the program.

Unfortunately, the **Halting Problem**, as this is called, cannot be solved. There will never be a computer program that can positively determine, for an arbitrary program **P**, if **P** will halt for all input. Nor will there ever be a computer program that can positively determine if arbitrary program **P** will halt for a specified input.

How can this be? Programmers look at programs regularly to determine if they will halt. Surely this can be programmed. As a warning to those who believe any program can be analyzed, carefully examine the following code fragment before reading on.

```
while (n > 1)
  if (ODD(n))
    n = 3 * n + 1;
  else
    n = n / 2;
```

This is a famous piece of code. The sequence of values that is assigned to n by this code is sometimes called the **Collatz sequence** for input value n. Does this code fragment halt for all values of n? Nobody knows the answer. Every input that has been tried halts. But does it always halt? Note that for this code fragment, since we do not know if it halts, we also do not know an upper bound for its running time. As for the lower bound, we can easily show $\Omega(\log n)$(see Exercise 3.13).

Personally, I have faith that someday some smart person will completely analyze this program and prove once and for all that the code fragment halts for all values of n. Doing so may well give us techniques that advance our ability to analyze programs in general. Unfortunately, proofs from **computability** – the branch of computer science that studies what is impossible to do with a computer – compel us to believe that there will always be another program that we cannot analyze. This comes as a result of the fact that the Halting Problem is unsolvable.

15.3.1 Uncountability

Before proving that the Halting Problem is unsolvable, we first prove that not all functions can be programmed. This is so because the number of programs is much smaller than the number of possible functions.

A set is said to be **countable** if every member of the set can be uniquely assigned to a positive integer. A set is said to be **uncountable** if it is not possible to assign every member of the set to a positive integer.

To understand what is meant when we say "assigned to a positive integer," imagine that there is an infinite row of bins, labeled 1, 2, 3, and so on. Take a set and start placing members of the set into bins, with at most one member per bin. If we can find a way to assign all of the members to bins, then the set is countable. For example, consider the set of positive even integers 2, 4, and so on. We can assign an integer i to bin $i/2$ (or, if we don't mind skipping some bins, then we can assign even number i to bin i). Thus, the set of even integers is countable. This should be no surprise, since there seems to be "fewer" positive even integers than there are positive integers. Interestingly, there are not really any more positive integers than there are positive even integers, since we can uniquely assign every positive integer to some positive even integer by simply assigning positive integer i to positive even integer $2i$.

On the other hand, the set of all integers is also countable, even though this set appears to be "bigger" than the set of positive integers. This is true because we can assign 0 to positive integer 1, 1 to positive integer 2, -1 to positive integer 3, 2 to positive integer 4, -2 to positive integer 5, and so on. In general, assign positive integer value i to positive integer value $2i$, and assign negative integer value $-i$ to positive integer value $2i + 1$. We will never run out of positive integers to assign, so every integer gets an assignment.

Are the number of programs countable or uncountable? A program can be viewed as simply a string of characters (including special punctuation, spaces, and line breaks). Let us assume that the number of different characters that can appear in a program is P. (In the ASCII character set, P must be less than 128, but the actual number does not matter). If the number of strings is countable, then surely the number of programs is also countable. We can assign strings to the bins as follows. Assign the null string to the first bin. Now, take all strings of one character, and assign them to the next P bins in "alphabetic" or ASCII code order. Next, take all strings of two characters, and assign them to the next P^2 bins, again in ASCII code order working from left to right. Strings of three characters are likewise assigned to bins, then strings of length four, and so on. In this way, a string of any given length can be assigned to some bin.

By this process, any string of finite length is assigned to some bin. So any program, which is merely a string of finite length, is assigned to some bin. Since all programs are assigned to some bin, the set of all programs is countable. Naturally most of the strings in the bins are not legal programs, but this is irrelevant. All that matters is that the strings that *do* correspond to programs are also in the bins.

Now we consider the number of possible functions. To keep things simple, assume that all functions take a single positive integer as input and yield a single positive integer as output. We will call such functions **integer functions**. A function is simply a mapping from input values to output values. Of course, not all computer programs literally take integers as input and yield integers as output. However, everything that computers read and write is essentially a series of numbers, which may be interpreted as letters or something else. Any useful computer program's input and output can be coded as integer values, so our simple model of computer input and output is sufficiently general to cover all possible computer programs.

1		2		3		4		5
x	$f_1(x)$	x	$f_2(x)$	x	$f_3(x)$	x	$f_4(x)$	
1	1	1	1	1	7	1	15	
2	1	2	2	2	9	2	1	
3	1	3	3	3	11	3	7	
4	1	4	4	4	13	4	13	
5	1	5	5	5	15	5	2	
6	1	6	6	6	17	6	7	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

Figure 15.4 An illustration of assigning functions to bins.

We now wish to see if it is possible to assign all of the integer functions to the infinite set of bins. If so, then the number of functions is countable, and it might then be possible to assign every integer function to a program. If the set of integer functions cannot be assigned to bins, then there will be integer functions that must have no corresponding program.

Imagine each integer function as a table with two columns and an infinite number of rows. The first column lists the positive integers starting at 1. The second column lists the output of the function when given the value in the first column as input. Thus, the table explicitly describes the mapping from input to output for each function. Call this a **function table**.

Next we will try to assign function tables to bins. To do so we must order the functions, but it does not matter what order we choose. For example, bin 1 could store the function that always returns 1 regardless of the input value. Bin 2 could store the function that returns its input. Bin 3 could store the function that doubles its input and adds 5. Bin 4 could store a function for which we can see no simple relationship between input and output.[3] These four functions as assigned to the first four bins are shown in Figure 15.4.

Can we assign every function to a bin? The answer is no, because there is always a way to create a new function that is not in any of the bins. Suppose that somebody presents a way of assigning functions to bins that they claim includes all of the functions. We can build a new function that has not been assigned to any bin, as follows. Take the output value for input 1 from the function in the first bin. Call this value $F_1(1)$. Add 1 to it, and assign the result as the output of a new function

[3]There is no requirement for a function to have any discernible relationship between input and output. A function is simply a mapping of inputs to outputs, with no constraint on how the mapping is determined.

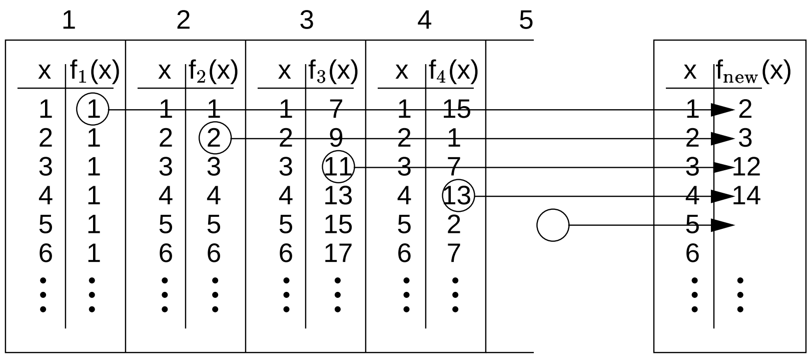

Figure 15.5 Illustration for the argument that the number of integer functions is uncountable.

for input value 1. Regardless of the remaining values assigned to our new function, it must be different from the first function in the table, since the two give different outputs for input 1. Now take the output value for 2 from the second function in the table (known as $F_2(2)$). Add 1 to this value and assign it as the output for 2 in our new function. Thus, our new function must be different from function 2, since they will differ at least at the second value. Continue in this manner, assigning $F_{new}(i) = F_i(i) + 1$ for all values i. Thus, the new function must be different from any function F_i at least at position i. Since the new function is different from every other function, it must not already be in the table. This is true no matter how we try to assign functions to bins, and so the number of integer functions is uncountable. The significance of this is that not all functions can possibly be assigned to programs; there *must* be functions with no corresponding program. Figure 15.5 illustrates this argument.

15.3.2 The Halting Problem Is Unsolvable

While it is theoretically interesting to know that there exists *some* function that cannot be computed by a computer program, does this mean that there is any *useful* problem that cannot be computed? Now we will prove that the Halting Problem cannot be computed by any computer program. The proof is by contradiction.

We begin by assuming that there is a function named `halt` that can solve the Halting Problem. Obviously, it is not possible to write out something that does not exist, but here is a plausible sketch of what a function to solve the Halting Problem might look like if it did exist. Function `halt` takes two inputs: a string representing the source code for a **C**++ program or function, and another string representing the input that we wish to determine if the input program or function halts on. Function halt returns true if the input program or function does halt on the given input, and false otherwise.

```
bool halt(char* prog, char* input)
{
  Code to solve halting problem
  if (prog does halt on input) then
    return(true);
  else
    return(false);
}
```

We now will examine two simple functions that clearly can exist since the complete **C**++ code for them is presented here:

```
bool selfhalt(char *prog) {
  // Return TRUE if prog halts when given itself as input.
  if (halt(prog, prog))
    return(true);
  else
    return(false);
}


void contrary(char *prog) {
  if (selfhalt(prog))
    while(true); // Go into an infinite loop
}
```

What happens if we make a program whose sole purpose is to execute contrary and run that program with itself as input? One possibility is that the call to selfhalt returns true; that is, selfhalt claims that contrary will halt when run on itself. In that case, contrary goes into an infinite loop (and thus does not halt). On the other hand, if selfhalt returns false, then halt is proclaiming that contrary does not halt on itself, and contrary then returns, that is, it halts. Thus, contrary does the contrary of what halt says that it will do.

The action of contrary is logically inconsistent with the assumption that halt solves the Halting Problem correctly. There are no other assumptions we made that might cause this inconsistency. Thus, by contradiction, we have proved that halt cannot solve the Halting Problem correctly, and thus there is no program that can solve the Halting Problem.

Now that we have proved that the Halting Problem is unsolvable, we can use reduction arguments to prove that other problems are also unsolvable. The strategy is to assume the existence of a computer program that solves the problem in question and use that program to solve another problem that is already known to be unsolvable.

For example, consider the following variation on the Halting Problem. Given a computer program, will it halt when its input is the empty string (i.e., will it halt when it is given no input)? To prove that this problem is unsolvable, we will employ a standard technique for computability proofs: Use a computer program to modify another computer program.

Assume that there is a function `Ehalt` that determines whether a given program halts when given no input. Recall that our proof for the Halting Problem involved functions that took as parameters a string representing a program and another string representing an input. Consider another function `combine` that takes a program P and an input string I as parameters. Function `combine` modifies P to store I as a static variable S and further modifies all calls to input functions within P to instead get their input from S. Call the resulting program P'. It should take no stretch of the imagination to believe that any decent compiler could be modified to take computer programs and input strings and produce a new computer program that has been modified in this way. Now, take P' and feed it to `Ehalt`. If `Ehalt` says that P' will halt, then we know that P would halt on input I. In other words, we now have a procedure for solving the original Halting Problem. The only assumption that we made was the existence of `Ehalt`. Thus, the problem of determining if a program will halt on no input must be unsolvable.

15.3.3 Determining Program Behavior Is Unsolvable

There are many things that we would like to have a computer do that are unsolvable. Many of these have to do with program behavior. For example, proving that a program is "correct," that is, proving that a program computes a particular function, is a proof regarding program behavior. As such, what can be accomplished is severely limited. In particular, it is not possible to reliably determine in all cases if a particular program computes a particular function. Nor is it possible to determine whether a particular line of code in a particular program will ever be executed.

This does *not* mean that a computer program cannot be written that works on special cases, possibly even on most programs that we would be interested in checking. For example, some **C** compilers will check if the control expression for a `while` loop is a constant expression that evaluates to `false`. If it is, the compiler will issue a warning that the `while` loop code will never be executed. However, it is not possible to write a computer program that can check for *all* input programs whether a specified line of code will be executed when the program is given some specified input.

Another unsolvable problem is whether a program contains a computer virus. The property "contains a computer virus" is a matter of behavior. Thus, it is not possible to determine positively whether an arbitrary program contains a computer virus. Fortunately, there are many good heuristics for determining if a program is likely to contain a virus, and it is usually possible to determine if a program contains a particular virus, at least for the ones that are now known. Real virus checkers do a pretty good job, but, it will always be possible for malicious people to invent new viruses that no existing virus checker can recognize.

15.4 Further Reading

The classic text on the theory of $\mathcal{NP}$-completeness is *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-completeness* by Garey and Johnston [GJ79]. *The Traveling Salesman Problem*, edited by Lawler et al. [LLKS85], discusses many approaches to finding an acceptable solution to this particular $\mathcal{NP}$-complete problem in a reasonable amount of time.

For more information about the Collatz function see "On the Ups and Downs of Hailstone Numbers" by B. Hayes, "Computer Recreations" in *Scientific American*, January 1984, and "The $3x + 1$ Problem and its Generalizations" by J.C. Lagarias in *American Mathematical Monthly*, January 1985.

For an introduction to the field of computability and impossible problems, see *Discrete Structures, Logic, and Computability* by James L. Hein [Hei95].

15.5 Exercises

15.1 Consider this algorithm for finding the maximum element in an array: Find the maximum element by first sorting the array and then selecting the last (maximum) element. What (if anything) does this reduction tell us about the upper and lower bounds to the problem of finding the maximum element in a sequence? Why can we not reduce SORTING to finding the maximum element?

15.2 Use a reduction to prove that squaring an $n \times n$ matrix is just as expensive as multiplying two $n \times n$ matrices.

15.3 Use a reduction to prove that multiplying two upper triangular $n \times n$ matrices is just as expensive as multiplying two arbitrary $n \times n$ matrices.

15.4 **(a)** Explain why computing the factorial of n by multiplying all values from 1 to n together is an exponential time algorithm.

(b) Explain why computing an approximation to the factorial of n by making use of Stirling's formula (see Section 2.2) is a polynomial time algorithm.

15.5 A **Hamiltonian cycle** in graph G is a cycle that visits every vertex in the graph exactly once before returning to the start vertex. The problem HAMILTONIAN CYCLE asks whether graph G does in fact contain a Hamiltonian cycle. Assuming that HAMILTONIAN CYCLE is $\mathcal{NP}$-complete, prove that TRAVELING SALESMAN is $\mathcal{NP}$-complete.

15.6 Assuming that VERTEX COVER is $\mathcal{NP}$-complete, prove that CLIQUE is $\mathcal{NP}$-complete by finding a polynomial time reduction from VERTEX COVER to CLIQUE.

15.7 We define the problem INDEPENDENT SET as follows:

INDEPENDENT SET
Input: A graph **G** and an integer k.
Output: YES if there is a subset **S** of the vertices in **G** of size k or greater such that no edge connects any two vertices in **S**, and NO otherwise.

Assuming that CLIQUE is $\mathcal{NP}$-complete, prove that INDEPENDENT SET is $\mathcal{NP}$-complete.

15.8 Prove that the set of real numbers is uncountable. Use a proof similar to the one used in Section 15.3.1 to prove that the set of integer functions is uncountable.

15.9 Here is another version of the knapsack problem, which we will call EXACT KNAPSACK. Given a set of items each with given integer size, and a knapsack of size integer k, is there a subset of the items which fits exactly within the knapsack?

Assuming that EXACT KNAPSACK is $\mathcal{NP}$-complete, use a reduction argument to prove that KNAPSACK is $\mathcal{NP}$-complete.

15.10 Prove, using a reduction argument such as given in Section 15.3.2, that the problem of determining if a program will print any output is unsolvable.

15.11 Prove, using a reduction argument such as given in Section 15.3.2, that the problem of determining if a program executes a particular statement within that program is unsolvable.

15.12 Prove, using a reduction argument such as given in Section 15.3.2, that the problem of determining if two programs halt on exactly the same inputs is unsolvable.

15.13 Prove, using a reduction argument such as given in Section 15.3.2, that the problem of determining whether there is some input on which two programs will both halt is unsolvable.

15.6 Projects

15.1 Implement VERTEX COVER; that is, given graph **G** and integer K, answer the question of whether or not there is a vertex cover of size K or less. Begin by using a brute-force algorithm of checking all possible sets of vertices of size K to find an acceptable vertex cover, and measure the running time on a number of input graphs. Then try to reduce the running time through the use of any heuristics you can think of. Next, try to find approximate solutions to the problem in the sense of finding the smallest set of vertices that forms a vertex cover.

15.2 Implement KNAPSACK. Measure its running time on a number of inputs. What is the largest practical input size for this problem?

15.3 Implement an approximation of TRAVELING SALESMAN; that is, given a graph **G** with costs for all edges, find the cheapest cycle that visits all vertices in **G**. Try various heuristics to find the best approximations for a wide variety of input graphs.

15.4 Write a program that, given a positive integer n as input, prints out the Collatz sequence for that number. What can you say about the types of integers that have long Collatz sequences? What can you say about the length of the Collatz sequence for various groups of integers?