PART V

Theory of Algorithms

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 easy to invent an equation to model the behavior of the algorithm or data structure in question, and also easy 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 8.5.2). In this case, once the snowplow argument is understood, the resulting equations are simple. Sometimes, 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 7.5 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 discussion of techniques for solving recurrences is provided in Section 14.2. These sections extend the introduction to summations and recurrences provided in Section 2.4, so 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

We begin our study of techniques for finding the closed-form solution to a summation by considering the simple example

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

In Section 2.6.3 it was proved by induction that this summation has the well-known closed form $n(n+1)/2$. But while induction is a good technique for proving that a proposed closed-form expression is correct, how do we find a candidate closed-form expression to test in the first place? Let us try to approach this summation from first principles, as though we had never seen it before.

A good place to begin analyzing a summation it is to give an estimate of its value for a given n. Observe that the biggest term for this summation is n, and there are n terms being summed up. So the total must be less than n^2. Actually, most terms are much less than n, and the sizes of the terms grows linearly. If we were to draw a picture with bars for the size of the terms, their heights would form a line, and we could enclose them in a box n units wide and n units high. It is easy to see from this that a closer estimate for the summation is about $(n^2)/2$. Having this estimate in hand helps us when trying to determine an exact closed-form solution, because we will hopefully recognize if our proposed solution is badly wrong.

Let us now consider some ways that we might hit upon an exact value for the closed form solution to this summation. One particularly clever approach we can take is to observe that we can "pair up" the first and last terms, the second and $(n-1)$th terms, and so on. Each pair sums to $n+1$. The number of pairs is $n/2$. Thus, the solution is $n(n+1)/2$. This is pretty, and there's no doubt about it being correct. The problem is that it is not a useful technique for solving many other summations.

Now let us try to do something a bit more general. We already recognized that, because the largest term is n and there are n terms, the summation is less than n^2. If we are lucky, the closed form solution is a polynomial. Using that as a working assumption, we can invoke a technique called **guess-and-test**. We will 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. Because 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 $c_2 = 1/2$. Thus, 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 still must do the "test" part of the guess-and-test approach. We can use an induction proof to verify whether our candidate closed-form solution is correct. In this case it is indeed correct, as shown by Example 2.11. The induction proof is necessary because our initial assumption that the solution is a simple polynomial could be wrong. For example, it might have been possible that the true solution includes a logarithmic term, such as $c_1n^2 + c_2n \log n$. The process shown here is essentially fitting a curve to a fixed number of points. Because there is always an n-degree polynomial that fits $n + 1$ points, we had not done enough work to be sure that we to know the true equation without the induction proof.

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. Why is this not a universal approach to solving summations? Because many summations do not have a polynomial as their closed form solution.

A more general approach is based on the **subtract-and-guess** or **divide-and-guess** strategies. One form of subtract-and-guess is known as the **shifting method**. 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. To solve sum f, we pick a known function g and find a pattern in terms of $f(n) - g(n)$ or $f(n)/g(n)$.

Example 14.1 Find the closed form solution for $\sum_{i=1}^{n} i$ using the divide-and-guess approach. We will try two example functions to illustrate the divide-and-guess method: dividing by n and dividing by $f(n-1)$. Our goal is to find patterns that we can use to guess a closed-form expression as our candidate for testing with an induction proof. To aid us in finding such patterns, we can construct a table showing the first few numbers of each function, and the result of dividing one by the other, as follows.

n	1	2	3	4	5	6	7	8	9	10
$f(n)$	1	3	6	10	15	21	28	36	46	57
n	1	2	3	4	5	6	7	8	9	10
$f(n)/n$	2/2	3/2	4/2	5/2	6/2	7/2	8/2	9/2	10/2	11/2
$f(n-1)$	0	1	3	6	10	15	21	28	36	46
$f(n)/f(n-1)$		3/1	4/2	5/3	6/4	7/5	8/6	9/7	10/8	11/9

Dividing by both n and $f(n-1)$ happen to give us useful patterns to work with. $\frac{f(n)}{n} = \frac{n+1}{2}$, and $\frac{f(n)}{f(n-1)} = \frac{n+1}{n-1}$. Of course, lots of other approaches do not work. For example, $f(n)-n = f(n-1)$. Knowing that $f(n) = f(n-1)+n$ is not useful for determining the closed form solution to this summation. Or consider $f(n) - f(n-1) = n$. Again, knowing that $f(n) = f(n-1) + n$ is not useful. Finding the right combination of equations can be like finding a needle in a haystack.

In our first example, we can see directly what the closed-form solution should be.

$$\frac{f(n)}{n} = \frac{n+1}{2}$$

Obviously, $f(n) = n(n+1)/2$.

Dividing $f(n)$ by $f(n-1)$ does not give so obvious a result, but it provides another useful illustration.

$$\begin{aligned}
\frac{f(n)}{f(n-1)} &= \frac{n+1}{n-1} \\
f(n)(n-1) &= (n+1)f(n-1) \\
f(n)(n-1) &= (n+1)(f(n)-n) \\
nf(n) - f(n) &= nf(n) + f(n) - n^2 - n \\
2f(n) &= n^2 + n = n(n+1) \\
f(n) &= \frac{n(n+1)}{2}
\end{aligned}$$

Once again, we still do not have a proof that $f(n) = n(n+1)/2$. Why? Because we did not prove that $f(n)/n = (n+1)/2$ nor that $f(n)/f(n-1) = (n+1)(n-1)$. We merely hypothesized patterns from looking at a few terms. Fortunately, it is easy to check our hypothesis with induction.

Example 14.2 Solve 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. Because 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:

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

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 7.4) 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 $\mathrm{O}(n^2)$. 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 because $c2n = 2cn$ 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 $\Theta(n \log n)$.

Example 14.5 We know that the factorial function grows exponentially. How does it compare to 2^n? To n^n? Do they all grow "equally fast" (in an asymptotic sense)? We can begin by looking at a few initial terms.

n	1	2	3	4	5	6	7	8	9
$n!$	1	2	6	24	120	720	5040	40320	362880
2^n	2	4	8	16	32	64	128	256	512
n^n	1	4	9	256	3125	46656	823543	16777216	387420489

We can also look at these functions in terms of their recurrences.

$$n! = \begin{cases} 1 & n = 1 \\ n(n-1)! & n > 1 \end{cases}$$

$$2^n = \begin{cases} 2 & n = 1 \\ 2(2^{n-1}) & n > 1 \end{cases}$$

$$n^n = \begin{cases} n & n = 1 \\ n(n^{n-1}) & n > 1 \end{cases}$$

At this point, our intuition should be telling us pretty clearly the relative growth rates of these three functions. But how do we prove formally which grows the fastest? And how do we decide if the differences are significant in an asymptotic sense, or just constant factor differences?

We can use logarithms to help us get an idea about the relative growth rates of these functions. Clearly, $\log 2^n = n$. Equally clearly, $\log n^n = n \log n$. We can easily see from this that 2^n is $o(n^n)$, that is, n^n grows asymptotically faster than 2^n.

How does $n!$ fit into this? We can again take advantage of logarithms. Obviously $n! \leq n^n$, so we know that $\log n!$ is $\mathrm{O}(n \log n)$. But what about a lower bound for the factorial function? Consider the following.

$$\begin{aligned} n! &= n \times (n-1) \times \cdots \times \frac{n}{2} \times (\frac{n}{2} - 1) \times \cdots \times 2 \times 1 \\ &\geq \frac{n}{2} \times \frac{n}{2} \times \cdots \times \frac{n}{2} \times 1 \times \cdots \times 1 \times 1 \\ &= (\frac{n}{2})^{n/2} \end{aligned}$$

Therefore

$$\log n! \geq \log(\frac{n}{2})^{n/2} = (\frac{n}{2}) \log(\frac{n}{2}).$$

In other words, $\log n!$ is in $\Omega(n \log n)$. Thus, $\log n! = \Theta(n \log n)$.

Note that this does **not** mean that $n! = \Theta(n^n)$. Because $\log n^2 = 2 \log n$, it follows that $\log n = \Theta(\log n^2)$ but $n \neq \Theta(n^2)$. The log function often works as a "flattener" when dealing with asymptotics. (And the anti-log works as a booster.) That is, whenever $\log f(n)$ is in $O(\log g(n))$ we know that $f(n)$ is in $\mathrm{O}(g(n))$. But knowing that $\log f(n) = \Theta(\log g(n))$ does not necessarily mean that $f(n) = \Theta(g(n))$.

Example 14.6 What is the growth rate of the fibonacci sequence $f(n) = f(n-1) + f(n-2)$ for $n \geq 2$; $f(0) = f(1) = 1$?

In this case it is useful to compare the ratio of $f(n)$ to $f(n-1)$. The following table shows the first few values.

n	1	2	3	4	5	6	7
$f(n)$	1	2	3	5	8	13	21
$f(n)/f(n-1)$	1	2	1.5	1.666	1.625	1.615	1.619

Following this out a few more terms, it appears to settle to a ratio of approximately 1.618. Assuming $f(n)/f(n-1)$ really does tend to a fixed value, we can determine what that value must be.

$$\frac{f(n)}{f(n-2)} = \frac{f(n-1)}{f(n-2)} + \frac{f(n-2)}{f(n-2)} \rightarrow x + 1$$

This comes from knowing that $f(n) = f(n-1) + f(n-2)$. We divide by $f(n-2)$ to make the second term go away, and we also get something useful in the first term. Remember that the goal of such manipulations is to give us an equation that relates $f(n)$ to something without recursive calls.

For large n, we also observe that:

$$\frac{f(n)}{f(n-2)} = \frac{f(n)}{f(n-1)} \frac{f(n-1)}{f(n-2)} \rightarrow x^2$$

as n gets big. This comes from multiplying $f(n)/f(n-2)$ by $f(n-1)/f(n-1)$ and rearranging.

If x exists, then $x^2 - x - 1 \rightarrow 0$. Using the quadratic equation, the only solution greater than one is

$$x = \frac{1+\sqrt{5}}{2} \approx 1.618.$$

This expression also has the name ϕ. What does this say about the growth rate of the fibonacci sequence? It is exponential, with $f(n) = \Theta(\phi^n)$. More precisely, $f(n)$ converges to

$$\frac{\phi^n - (1-\phi)^n}{\sqrt{5}}.$$

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 2.4. A more complex example is given below.

Example 14.7 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:

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

From Equation 2.6, 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.

Example 14.8 Our next example comes from the algorithm to build a heap. Recall from Section 5.5 that to build a heap, we first heapify the two subheaps, then push down the root to its proper position. The cost is:

$$f(n) \leq 2f(n/2) + 2\log n.$$

Let us find a closed form solution for this recurrence. We can expand the recurrence a few times to see that

$$\begin{aligned} f(n) &\leq 2f(n/2) + 2\log n \\ &\leq 2[2f(n/4) + 2\log n/2] + 2\log n \\ &\leq 2[2(2f(n/8) + 2\log n/4) + 2\log n/2] + 2\log n \end{aligned}$$

We can deduce from this expansion that this recurrence is equivalent to following summation and its derivation:

$$\begin{aligned} f(n) &\leq \sum_{i=0}^{\log n-1} 2^{i+1}\log(n/2^i) \\ &= 2\sum_{i=0}^{\log n-1} 2^i(\log n - i) \\ &= 2\log n \sum_{i=0}^{\log n-1} 2^i - 4\sum_{i=0}^{\log n-1} i2^{i-1} \\ &= 2n\log n - 2\log n - 2n\log n + 4n - 4 \\ &= 4n - 2\log n - 4. \end{aligned}$$

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. $r = 1$. Because $r = b^k/a$, 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,

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

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

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

3. $r > 1$. From Equation 2.5,

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

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, sometimes referred to as the **Master Theorem**.

Theorem 14.1 ***(The Master Theorem)*** *For any recurrance relation of the form* $\mathbf{T}(n) = a\mathbf{T}(n/b) + cn^k, \mathbf{T}(1) = c$*, the following relationships hold.*

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

This theorem may be applied whenever appropriate, rather than re-deriving the solution for the recurrence.

Example 14.9 Apply the theorem to solve

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

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

Example 14.10 Use the theorem to solve the recurrence relation for Mergesort:

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

Because $a = 2$, $b = 2$, $c = 1$, 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 7.5, 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 assuming that the partitioning element is equally likely to occur in any position k. It can be simplified by observing that the two recurrence terms $\mathbf{T}(k)$ and $\mathbf{T}(n-1-k)$ are equivalent, because 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

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

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 taken as a whole. 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 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<A.length) && (A[i] == 1)); i++)
  A[i] = 0;
if (i < A.length)
  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, because 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 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

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

A similar argument was used in our analysis for the partition function in the Quicksort algorithm (Section 7.5). While on any given pass through the while loop the left or right pointers might move all the way through the remainder of the partition, doing so would reduce the number of times that the while loop can be further executed.

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, because 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 because *B* must appear before *A* in the list (because 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 $2i$ because each change involves an *A* and each *A* can be part of at most two changes.

Because 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. Because 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 appears in *Applied Combinatorics* by Fred S. Roberts [Rob84]. For a more advanced treatment, see *Concrete Mathematics* by Graham, Knuth, and Patashnik [GKP94].

Cormen, Leiserson, and Rivest provide a good discussion on various methods for performing amortized analysis in *Introduction to Algorithms* [CLRS01]. 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 Provide a summation for the value of `sum` in the following code fragment. Find and prove correct a closed form solution to the summation.

```
sum = 0; inc = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++) {
    sum = sum + inc;
    inc++;
  }
```

14.8 A chocolate company decides to promote its chocolate bars by including a coupon with each bar. A bar costs a dollar, and with c coupons you get a free bar. So depending on the value of c, you get more than one bar of chocolate for a dollar when considering the value of the coupons. How much chocolate is a dollar worth (as a function of c)?

14.9 Write and solve a recurrence relation to compute the number of times Fibr is called in the Fibr function of Exercise 2.11.

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

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

14.12 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 $\Omega(n \log n)$.

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

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

You may assume that n is a power of 2.

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

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

You may assume that n is a power of 2.

14.15 For the following recurrence, give a closed-form solution. You should not give an exact solution, but only an asymptotic solution (i.e., using Θ notation). You may assume that n is a power of 2. Prove that your answer is correct.

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

14.16 Section 5.5 provides an asymptotic analysis for the worst-case cost of function **`buildHeap`**. Give an exact worst-case analysis for **`buildHeap`**.

14.17 For each of the following recurrences, find and then prove (using induction) an exact closed-form solution. When convenient, you may assume that n is a power of 2.

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

(b) $\mathbf{T}(n) = 2\mathbf{T}(n/2) + n$ for $n > 2$; $\mathbf{T}(2) = 2$.

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

14.19 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.

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

14.21 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 $\mathrm{O}(1)$ 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 $\mathrm{O}(n)$ time for a series of n operations.

14.22 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:

```
static void concom(Graph G) {
  int i;
  for (i=0; i<G.n(); i++)  // For n vertices in graph
    G.setMark(i, 0);       // Vertex i in no component
  int comp = 1;            // Current component
  for (i=0; i<G.n(); i++)
    if (G.getMark(i) == 0) // Start a new component
      DFS_component(G, i, comp++);
  for (i=0; i<G.n(); i++)
    out.append(i + " " + G.getMark(i) + " ");
}

static void DFS_component(Graph G, int v, int comp) {
  G.setMark(v, comp);
  for (int w = G.first(v); w < G.n(); w = G.next(v, w))
    if (G.getMark(w) == 0)
      DFS_component(G, w, comp);
}
```

Use the concept of potential from amortized analysis to explain why the total cost of this algorithm is $\Theta(|\mathrm{V}| + |\mathrm{E}|)$. (Note that this will not be a true amortized analysis because 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.)

14.23 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.

14.24 Use mathematical induction to prove that

$$\sum_{i=1}^{n} Fib(i) = Fib(n-2) - 1, \text{for } n \geq 1.$$

14.25 Use mathematical induction to prove that Fib(i) is even if and only if n is divisible by 3.

14.26 Use mathematical induction to prove that for $n \geq 6$, $fib(n) > (3/2)^{n-1}$.

14.27 Find closed forms for each of the following recurrences.

(a) $F(n) = F(n-1) + 3; F(1) = 2.$
(b) $F(n) = 2F(n-1); F(0) = 1.$
(c) $F(n) = 2F(n-1) + 1; F(1) = 1.$
(d) $F(n) = 2nF(n-1); F(0) = 1.$

(e) $F(n) = 2^n F(n-1); F(0) = 1.$
(f) $F(n) = 2 + \sum_{i=1}^{n-1} F(i); F(1) = 1.$

14.28 Find Θ for each of the following recurrence relations.
(a) $T(n) = 2T(n/2) + n^2.$
(b) $T(n) = 2T(n/2) + 5.$
(c) $T(n) = 4T(n/2) + n.$
(d) $T(n) = 2T(n/2) + n^2.$
(e) $T(n) = 4T(n/2) + n^3.$
(f) $T(n) = 4T(n/3) + n.$
(g) $T(n) = 4T(n/3) + n^2.$
(h) $T(n) = 2T(n/2) + \log n.$
(i) $T(n) = 2T(n/2) + n \log n.$

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 $\Theta(n \log^* n)$.