Chapter 2

The Mathematical tools of algorithmic analysis I

2.0 Preliminaries

This section begins by reviewing elementary mathematics that is essential for subsequent chapters.

Exponentiation

\[ a^k = a \times a \times a \times \cdots \times a, \] where there are \( k \) \( a \)'s being multiplied together. In words, \( a^k \) is called \( a \) to the \( k \)'th power, or simply \( a \) to the \( k \), or \( a \) raised to the \( k \).

A consequence of this definition of \( a^k \) is that

\[ a^x \times a^y = a^{x+y}. \]  \hspace{1cm} \text{rule(1)}

the product of \( x \) \( a \)'s and \( y \) more \( a \)'s is the same as the product of \( x+y \) \( a \)'s. In words, rule(1) says, when the bases are the same (i.e., \( a \)), the exponent of the product is the sum of the exponents of each factor; \( a \) winds up with the exponent \( (x+y) \).

Similarly, we have

\[ (a^x)^y = a^{xy}. \]  \hspace{1cm} \text{rule(2)}

This follows because \( (a^x)^y = a^x \times a^x \times \cdots \times a^x \), where there are \( y \) clusters of \( a^x \). Since each cluster represents \( x \) \( a \)'s multiplied together, there are \( xy \) \( a \)'s being multiplied altogether.

Trivial consequence:

These formulas for the behavior of exponents extend to real numbers, both positive and negative.

In particular,

\[ a^{-x} = \frac{1}{a^x}. \]  \hspace{1cm} \text{rule(3)}

Thus, \( a^x \times a^{-y} = a^x \times \frac{1}{a^y} = a^{x-y} \). The fact that \( a^x \times a^{-y} = a^{x-y} \) follows from cancelling out \( y \) \( a \)'s from the product of \( x \) \( a \)'s in \( a^x \). The result is still correct if \( x-y \) is negative.

Trivial consequence: \( \frac{a^x}{a^x} = a^x \times a^{-x} = a^{x-x} = a^0 = 1. \)
The final rule for exponents is that 

\[ a^{1/x} = \sqrt[x]{a}, \]

that is, \( a^{1/x} \) is the \( x \)th root of \( a \). We know this because \((a^{1/x})^x = a^{x/x} = a^1 = a \). In words, the \( x \)th root of \( a \) is the value that, when raised to the \( x \)th power, gives \( a \).

**Examples**

\[
2^0 = 1. \quad 100^0 = 1. \quad (3^5)^4 = 3^{20}. \quad 8^{2/3} = (2^3)^{2/3} = 2^{3 \times 2/3} = 2^2 = 4. \quad 8^{2/3} = (8^{1/3})^2 = (\sqrt[3]{8})^2 = \frac{(2)^2}{1} = \frac{1}{\sqrt[3]{4}} = \frac{1}{\sqrt[3]{4}} = \frac{1}{\sqrt[3]{4}}.
\]

\( 0^0 \) is undefined, although if we take it to be the limit of \( e^x \), as \( e \) gets closer and closer to zero, then, as we shall see, there is a natural interpretation for the expression, which equals one, in this case.

It should be noted that these rules are actually consequences (theorems) that result from the simple definition for \( a^k \). That is why each rule is accompanied by a reason to explain why it holds.

There is one convention, which is not a theorem, but is a matter of good sense: The expression \( a^k \) is defined to be \( a^{(k^e)} \) (and not \( (a^k)^e \)). Thus \( 3^{3^3} = 3^{(3^3)} = 3^{27} \) (and not \( (3^3)^3 = 3^9 \)). There is no theorem that demands this interpretation, but the alternative would give the same as \( a^{k \times e} \), which would seem to be a waste of notation. Higher towers of exponents can be defined, and the precedence rule is that when parentheses are absent, the evaluation occurs at the high exponents first.

**Logarithms**

Logarithms are nothing more than a restatement of the exponentiation rules 1 through 4 from the exponent's point of view, and a notation to match.

In the expression \( e^k \), \( e \) is the base and \( k \) is the exponent. The log function returns the exponent for a specified base:

\[ \log_e(e^k) = k. \]

More abstractly, \( \log_e(x) \) is the exponent, which when base \( e \) is raised to this exponent, yields the value \( e \) the exponent = \( x \):

\[ e^{\log_e(x)} = x. \]

This last equation is a perfectly good definition of \( \log_e(x) \): it is the exponent that \( e \) must be raised to, in order to get the answer \( x \).

Again, there are consequences, theorems, or rules. In the rules that follow, we shall not explicitly state what the base is if its value has no bearing on the identity. It will be assumed that all bases must be the same, unless different values are explicitly written. Rules that are used to attain the identities listed are named by number as a subscript for the equals sign. Thus

\begin{equation}
expression1 = expression2
\end{equation}

means: applying rule(3) shows that

\begin{equation}
expression1 and expression2 are equal.
\end{equation}

\[ \log(r \times s) = \log(r) + \log(s). \]

**rule(6)**

Reason: \( e^{\log_e(r) \times \log_e(s)} = e^{\log_e(r)} e^{\log_e(s)} = r s = e^{\log_e(rs)} \). The first and last exponents must be equal.

Application: \( \log(1) = 0. \) Reason: \( \log(x) = e^{\log_e(x)} + \log(1) \). Alternative reason: \( e^0 = 1, \) for \( e \neq 0 \).
\[
\log(1/b) = -\log(b). \\
\text{rule(7)}
\]

Reason: \(c^{-\log_c b} = \frac{1}{c^{\log_c b}} = \frac{1}{b} = c^{\log_c(1/b)}\). The first and last exponents must be equal. Alternative reason: \(0 = \log(1) = \log\left(\frac{1}{b}\right) = \log(b) + \log\left(\frac{1}{b}\right)\).

Application: \(\log(a/b) = \log(a) - \log(b)\). Reason: \(\log(a \times \frac{1}{b}) = \log(a) + \log\left(\frac{1}{b}\right) = \log(a) - \log(b)\).

\[
\log(a^x) = x \log(a). \\
\text{rule(8)}
\]

Reason: \(c^{\log_c(a^x)} = (a)^x = c^{\log_c(a)^x} = 2c^{x \log_c a}\). The first and last exponents must be equal.

\[
\log_a(z) = \frac{\log_b(z)}{\log_b(a)}. \\
\text{rule(9)}
\]

Reason: \(a^{\log_b(x)} = b^{\log_b(x)}\). Taking \(\log_b\) of both sides (via rules 8 and 5) gives \(\log_b(z) \times \log_b(a) = \log_b(z)\). Dividing by \(\log_b(a)\) gives rule(9).

**Notation**

**Sets, Sequences, Functions, and Number Ranges**

**Sequences of identifiers and values**

We shall use \(a_n, a_i, b_n\), and the like to denote sequences of arbitrary values or variables. There is really no difference between the sequence \(a_1, a_2, \ldots, a_i, \ldots\) and a function \(a(i)\) defined for \(i = 1, 2, 3, \ldots\).

**Sets and Multisets**

We sometimes write down a set by listing its members within the curly brackets \{\}: \{1, 2, 6\} is a set containing the elements 1, 2, and 6.

We may say that \(x\) is an element of \(S\) by writing \(x \in S\); this is read, "\(x\) is in \(S\)," or "\(x\) is an element of \(S\)."

The ordering of elements in a set does not matter, so that \(\{6, 1, 2\} = \{1, 2, 6\}\); the sets are the same.

If \(S\) is a set, we use \(|S|\) to denote the number of elements in \(S\).

The set notation

\[
\{x \mid \text{predicate}(x)\}
\]

means the set of all \(x\) where the boolean \(\text{predicate}(x)\) is true. If there are several predicates separated by commas, we require that all predicates be true.

If \(A\) and \(B\) are sets, \(A \cup B\) denotes the union, and \(A \cap B\) denotes the intersection. Using set notation, we may write:

\[
A \cap B = \{x \mid x \in A, x \in B\} = \{x \mid x \in A \text{ and } x \in B\}.
\]

\[
A \cup B = \{x \mid x \in A \text{ or } x \in B\}.
\]

By definition, a value appears in a set just once, so that \(M = \{1, 1, 2\}\) is not a set. The name **multiset** is used to describe the variant where elements can appear more than once. As written, \(M\) is a multiset, and its size is \(|M| = 3\).
Ordered pairs and Tuples

An ordered pair comprising elements \(x\) and \(y\) is written \((x, y)\). Here the ordering matters, so that \((x, y) \neq (y, x)\) unless \(x = y\). A \(k\)-tuple is an ordered sequence of \(k\) items, such as \((a_1, a_2, \ldots, a_k)\).

Number ranges

We shall use the notation \([1..n]\) to denote the integer sequence 1, 2, \ldots, \(n\), or more generally \([i..j]\) to denote \(i, i + 1, i + 2, \ldots, j\).

The interval of real numbers, for \(a \leq x \leq b\) will be denoted by \([a, b]\).

Summations

\[ \sum_{i=1}^{j} f(i) \equiv f(1) + f(2) + \cdots + f(j) \]  
More generally, \(\sum_{i=r}^{s} a_i \equiv a_r + a_{r+1} + a_{r+2} + \cdots + a_s\). Here, the \(a_i\) are any sequence of variables or expressions indexed by \(i\) in the range from \(r\) to \(s\). The range values \(r\) and \(s\) must be integers. If \(r = s\), then there is just one term in the summation. If \(r > s\), then by convention the sum is empty and equals zero. We also use \(\sum_{r \leq i \leq s} a_i\) to represent \(\sum_{i=r}^{s} a_i\).

In displaystyle, we write: \(\sum_{r \leq i \leq s} a_i\) and \(\sum_{i=r}^{s} a_i\). If \(S\) is a set, we may write \(\sum_{x \in S} f(x)\) or \(\sum_{x \in S} f(x)\) to denote the sum of \(f\) as applied to each of the elements in the set \(S\).

Examples

\[ \sum_{i=1}^{5} i = 1 + 2 + 3 + 4 + 5 = 15. \quad \sum_{i=1}^{4} (i^2 + 1) = (1^2 + 1) + (2^2 + 1) + (3^2 + 1) + (4^2 + 1) = 34. \]  
\[ \sum_{i=1}^{3} i = 1 + 1 + 1 = 3. \quad \sum_{i=3}^{2} i = 0. \]  
Let \(S = \{3, 7, 2\}\), and \(f(x) = x^3\). Then \(\sum_{s \in S} f(s) = 3^3 + 7^3 + 2^3\).

Products

Although less common, product notation, which is defined quite similarly to sums, does arise upon occasion. \(\prod_{i=r}^{s} a_i \equiv a_r \times a_{r+1} \times a_{r+2} \cdots \times a_s\). As before, the index range must be within the integers. If \(r > s\), then by convention the product is empty and equals one.

Examples

\[ \prod_{i=1}^{3} i = 1 \times 2 \times 3 \times 4 \times 5 = 120. \quad \prod_{i=3}^{2} i = 1. \]

Minimum and maximum

By definition, \(\min(a, b)\) is defined to equal \(a\), if \(a < b\), and \(b\) otherwise. This notation can be extended to multiple arguments, so that \(\min(a, b, c, d, e, f)\) equals the smallest value among the six arguments. The notation extends naturally to sequences and formulae, so that, for example, \(\min_{i \leq i \leq s} a_i \equiv \min(a_r, a_{r+1}, a_{r+2}, \ldots, a_s)\). If \(S\) is a set of real numbers, we may represent its smallest value by \(\min_{a \in S} a\). We could identify the set of records with smallest keys in a set \(R\) by \(\{x \mid x \in R, x.data = \min_{r \in R} r.data\}\).

Analogous notation is used with max.

Floor and Ceiling

We use \([x]\) (the floor of \(x\)) to round down: \([x]\) equals the largest integer that does not exceed \(x\). For example, \([7/2] = 3\), \([9.1] = 9\), and \([-5.2] = -6\).

Similarly, \([n]\) (the ceiling of \(x\)) rounds up: \([n]\) equals the smallest integer that is at least as large as \(x\). Thus, \([7/2] = 4\), \([9.1] = 10\), and \([-5.2] = -5\).

Notice that if \(n\) is an integer, then \(n = [\frac{n}{1}] + [\frac{n}{2}]\).

Combinatorial formulae

\[ n! \equiv 1 \times 2 \times 3 \cdots (n - 1) \times n. \]  
In words, \(n!\) is \(n\) factorial.
The binomial coefficient \( \binom{n}{k} \) is defined as \( \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\cdots(n-k+2)(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \). In words, \( \binom{n}{k} \) is \( n \) choose \( k \).

**Inverse and derivative**

If \( f \) is a function, let \( f^{inv} \) denote its inverse, so that \( f^{inv}(y) = x \) means that \( f(x) = y \). Of course \( f^{inv} \) must have a unique value for this definition to make sense, which means that for each \( y \) value in the range, there can be at most one \( x \) where \( f(x) = y \).

Let \( f'(x) \) denote \( \frac{d}{dx} f(x) \).

## 2.1 Introduction

Many problems can be solved by a variety of different algorithms, and a common subtask is to determine which algorithm is the "best" for a problem at hand. One way to distinguish between two competing algorithms is to assign a complexity measure to the algorithms. Having done this, one may choose the algorithm with the lowest complexity.

There are two kinds of complexity measures:

- **Fixed**: A complexity measure is fixed, if it does not depend on the input values. An example of such a measure is the *program length*. Another such measure would be the time to program a given algorithm. Such measures are relevant if the algorithm is to be used only a few times, or temporarily within a large system that is under development.

- **Data Dependent**: A complexity measure is data dependent if it varies with the input values. Some examples of measures include the *running time* and the *storage space* for an algorithm. These measures are functions of the input values. These measures are important for algorithms that are to be run many times on 'large' sets of input values.

In most practical applications, the data dependent complexity measures are more significant, since production software is assumed to run a large number of times, and the costs of producing and maintaining the software (which depend on static complexities) are amortized rather quickly. We shall use the running time as our complexity measure, since almost all the algorithms we consider have a space bound that is a linear function of the input size. Furthermore, our analyses of running times will ignore constant factors, and will concentrate only on the orders of growth. There are three reasons for doing so:

1. This allows us to ignore details of the machine model, (such as, the hardware of the machine, the instruction sets of the computer, the memory structure of the computer, the quality of code generated by the compiler, etc.) thus giving us a *machine-independent* complexity measure.

2. This allows us to ignore insignificant data dependent details of the algorithm, such as a conditional branch that inserts a few extra operations.

3. For large enough problem sizes the relative efficiencies of two algorithms depend on the running times as an *asymptotic* function of input size, independent of constant factors.
We shall generally measure the running time, \( T(n) \), as a function of the worst-case input data of size \( n \); that is, \( T(n) \) is the maximum, over all inputs of size \( n \), of the running time on that input. The worst-case analysis provides a performance guarantee, but may be overly pessimistic, if the worst-case inputs seldom occur.

An alternative is an average-case analysis: we measure the running time, \( T_{\text{avg}}(n) \), as the average, over all possible inputs of size \( n \), of the running time on that input. Unfortunately, such an analysis is frequently mathematically intractable. Moreover, a realistic averaging ought to be based on the kind of input data that occurs in real applications, but this may be very difficult to determine. We shall restrict ourselves principally to worst-case complexity analysis, and will study average-case behavior for just a few problems, where such an analysis is appropriate.

For completeness, we note that fixed and data dependent complexity measures are often named, respectively, static and dynamic complexity. It is also worth emphasizing that the running time \( T(n) \), for a given algorithm, is indexed by the size of the input set. If the algorithm operates on an array of \( m \) elements, then \( n \) is taken to be \( m \); if the algorithm inputs an object \( X \), then \( n = \text{Size}(X) \), which might be the number of words needed to store \( X \). Sometimes the meaning of \('n'\) in \( T(n) \) is determined by the context; for instance, we could let \( T(n) \) be the number of operations needed to multiply two \( n \times n \) matrices, even though an \( n \times n \) matrix comprises \( n^2 \) words of data.

### 2.2 Big Omicron, Big Omega, Big Theta, Little Omicron

There is a very convenient notation for describing the growth behavior of functions. We can describe the asymptotic growth of a function \( T(n) \) using the expressions: \( O(\cdot) \) (known as big-omicron or more popularly, big-oh), \( \Omega(\cdot) \) (big-omega), \( \Theta(\cdot) \) (big-theta), and \( o(\cdot) \) (little-omicron or more popularly, little-oh). For our computer science applications, all functions will be positive and defined only for integers \( n \geq 0 \). The reason for this simplicity is that we will be typically using this notation to characterize running times as a function of problem size. Sizes are integer-based, while programs and subroutines have positive execution times.

The formal definitions of these notations are as follows:

- We write \( T(n) = O(f(n)) \), if there are two positive constants \( C \) and \( n_0 \) such that

\[
T(n) \leq C \cdot f(n), \text{ for all } n \geq n_0.
\]

- \( T(n) = \Omega(f(n)) \), if there are two positive constants \( C \) and \( n_0 \) such that

\[
T(n) \geq C \cdot f(n), \text{ for all } n \geq n_0.
\]

\(^1\) Some define this differently: \( T(n) = \Omega(f(n)) \), if there is a positive constant \( C \) such that \( T(n) \geq C \cdot f(n) \) infinitely often (for infinitely many integer values). This is a weaker definition, and seems more useful for lower-bound proofs—but, for all practical purposes, the stronger definition works fairly well. To see the difference in definitions, let \( f(n) = (1 + (-1)^n)n^2 + n \), and \( g(n) = n \). Then by this weaker definition, \( f(n) = \Omega(g(n)) \) and \( g(n) = \Omega(f(n)) \).
• \( T(n) = \Theta(f(n)) \), if there are three positive constants \( C_1, C_2, \) and \( n_0 \) such that
\[
\frac{f(n)}{C_1} \leq T(n) \leq C_2 \cdot f(n), \quad \text{for all } n \geq n_0.
\]

• \( T(n) = o(f(n)) \) if
\[
\lim_{n \to \infty} \frac{T(n)}{f(n)} = 0.
\]

Equivalently, \( f(n) = \Theta(g(n)) \) means: \( f(n) = O(g(n)) \), and \( g(n) = O(f(n)) \).

It is worth noting that because our functions will be positive and confined over the domain of integers, the constants \( n_0 \), in these definitions could be all changed to zero simply by making the constants \( C, C_1 \) and \( C_2 \) large enough. Similarly, we could replace \( C_1 \) and \( C_2 \) by just one value \( C \) that equals, say the maximum of \( C_1 \) and \( C_2 \).

For applications in general mathematics, the \( n_0 \) is needed, because \( f \) might become zero for small values. In general mathematical settings, the inequalities are always used for the magnitude of \( T \). For example, the mathematical world writes \( T(x) = O(f(x)) \), if there are two positive constants \( C \) and \( x_0 \) such that
\[
|T(x)| \leq C \cdot f(x), \text{ for all } x \geq x_0.
\]

Similarly, the very expressive \( f(x) = g(x) + O(h(x)) \) means that \( |f(x) - g(x)| = O(h(x)) \).

2.2.1 Further Remarks about \( O, \Theta, \) etc.

By definition, the notation \( T(n) = O(f(n)) \) means that for large \( n \), \( T \) grows no bigger than \( f \), up to some (possibly large, but fixed) constant factor. Thus \( n = O(\frac{n}{100}) \), since we can choose \( C = 100 \).

In fact, \( n + 100 = O(\frac{n-200}{300}) \) (we might choose say, \( n_0 = 500 \), and \( C = 600 \). It is not hard to see that the definitions hold for these constants. One could pick smaller constants which would also work, but the point of this notation is to capture the gist of the growth behavior; there is no point in finding a minimal pair \( C \) and \( n_0 \), should such a pair exist.)

Here are a few additional examples:

\[
\begin{align*}
5(n+1)n^3 & = \Theta(n^3), \\
(2 + \sin(n))n^{100} & = \Theta(n^{100}), \\
10000n^{100} & = o(n^{100}), \\
100 & = O(1), \\
1 & = O(100), \\
100 & = \Theta(1). \\
n & = O(n) \\
100n & = O(n), \\
100n & = O(n^2). \\
(2 + (-1)^n)n^3 & = \Theta(n^3).
\end{align*}
\]

It should be evident that \( n = O(n^{100}) \), that is, \( n \) does not grow bigger than \( n^{100} \). While such a statement is clearly true, it shows us that the big-oh notation may not always contain much
information, and more importantly, it shows that big-oh expressions are not actual equality statements. Thus, $O$ is analogous to $\leq$, and not to $=$. In more formal terms, the $O$ relation is not symmetric: if $T(n) = O(f(n))$, it is not true, in general that $f(n) = O(T(n))$. We also note that the statement $O(f(n)) = T(n)$ has no meaning. Thus the use of the equal sign, in this notation, is rather misleading. A better notation would be set membership: $T(n) \in O(f(n))$, since $O(f(n))$ might be viewed as the set of all functions that grow no bigger than $f$, apart from constant factors. We use the $\preceq$ sign for historical reasons; it was perpetrated by the mathematical community years ago.

Notice that the $\Theta$ notation conveys more information than the big-oh notation: $T(n) = \Theta(f(n))$ says that for large enough $n$, the functions $f$ and $T$ are, in fact, no more than a constant factor apart from each other. If $T(n)$ is the number of operations that occur for an algorithm, then the characterization $T(n) = \Theta(n^3)$ says that we have analyzed the performance of the algorithm reasonably well. For example, the program will be able to solve eight problems of size $n$ in about the time it takes to solve a single problem of size $2n$.

Moreover, $\Theta$ is symmetric: if $g(n) = \Theta(f(n))$, then $f(n) = \Theta(g(n))$. Notice that the $\Theta$ notation does not say that two functions are proportional, that is, related by a constant factor. The "constant" can wiggle around in value, like the factor $(2 + (-1)^n)$ in the last example.

Little-oh would seem to convey less information than big-oh, since $T(n) = o(f(n))$ says that $f$ grows bigger than any constant times $T$. In practice, little-oh is frequently used a little differently. Its chief purpose is to capture the notion "plus less important terms." For example, $n^2 + 3n + 100$ equals $n^2$ "plus less important terms." We can write this as $n^2 + 3n + 100 = n^2 + o(n^2)$. The right-hand side might also be written as $n^2(1 + o(1))$. The expression $o(1)$ refers to functions that go to zero (so that no matter what constant you multiply the function by, the product, for large enough $n$ will be less than 1.) Thus $n^2(1 + o(1))$, which equals $n^2 + n^2 \cdot o(1)$, describes functions $n^2 +$ terms that are insignificant when compared to $n^2$. Evidently, there is an underlying algebra associated with these growth notations; we explore some of these rules in the next section.

### 2.2.2 Algebra on $O$

Many of the rules of algebra work with $O$-notations, but some don't. The main subtlety, as we have observed, is that in the expression $T(n) = O(f(n))$, the $\preceq$ is a one-way equality (unlike true equality, it is not symmetric.)

So how do you manipulate expressions with $O$'s? Since we are about to illustrate some of properties associated with these growth operators in a format that technically violates our definitions, a few preliminary words of explanation would seem appropriate. The key point is that $O$, and $o$ are transitive. For example, transitivity says that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. With a slight abuse (i.e., misuse) of notation, we could express this by writing: if $g(n) = O(h(n))$, then $O(g(n)) = O(h(n))$, which is interpreted as meaning, "Anything that has growth no bigger than $O(g(n))$ must also have growth no bigger than $O(h(n))", since

---

2 $\Omega$ is also transitive as we defined it, although not as defined in footnote 1. $\Theta$ is symmetric as well as transitive.
Here, then, are some of the properties of $O$-notation:

\[
\begin{align*}
    f(n) &= O(f(n)), \\
    c \cdot O(f(n)) &= O(f(n)), & \text{c is a constant,} \\
    O(O(f(n))) &= O(f(n)), \\
    O(f(n)) + O(g(n)) &= O(f(n) + g(n)), \\
    O(f(n)) + o(f(n)) &= O(f(n)), \\
    O(f(n)) O(g(n)) &= O(f(n) g(n)), \\
    O(f(n) g(n)) &= f(n) O(g(n)), \\
    O(f(n)) &= f(n) O(1).
\end{align*}
\]

Interestingly, all but the first expression above turn out to be true identities; the other seven statements are true if the left and right sides are exchanged (and $c > 0$ in the second case). Notice also that the last relation is a special case of its predecessor (with $g = 1$).

It is also worth noting that (for positive functions) $\max(f(n), g(n)) = \Theta(f(n) + g(n))$. This is readily proved by the inequalities: $\max(f(n), g(n)) \leq f(n) + g(n) \leq \max(f(n), g(n)) + \max(f(n), g(n))$. Of course the statement is false if the functions could be negative: notice what happens if, for example, $f(n)$ is actually $-g(n)$.

Consider the program $P$:

\[
\begin{align*}
    \text{procedure } P(n); \\
    \text{begin} \\
    P_1(n); P_2(n) \\
    \text{end.}
\end{align*}
\]

Let $T_1(n) = O(f_1(n))$ be the running time of the program $P_1$, and $T_2(n) = O(f_2(n))$, the running time of $P_2$. Program $P$ has a running time $T_1(n) + T_2(n)$. It is clear that

\[
\begin{align*}
    T_1(n) + T_2(n) &= O(f_1(n)) + O(f_2(n)) \\
                     &= O(f_1(n) + f_2(n)) \\
                     &= O(\max(f_1(n), f_2(n))).
\end{align*}
\]

Big-oh, little-oh and $\Theta$ notations all satisfy some fairly simple algebraic properties—so surprisingly simple that just a bit of practice ought to suffice to acquaint the reader with the identities and their manipulation. The $\Omega$ notation as defined in footnote 1 is a little tricky, due to the technical definition of the relation, but these pathologies seldom occur in practice.

### 2.2.3 Ordering functions by asymptotic growth

This section is a practicum on how to determine asymptotic function growth. From an abstract point of view, it contains no theorems. Indeed, the material is already implicit in our understanding

---

\(^3\)Growth notations were only defined when they appear on the right-hand side of an equal sign. However, we can view $O(f(n))$ as a set of functions and interpret the equal sign as set containment `$\subseteq$', when $O$ appears on the left as well as the right, and set membership `$\in$' when the expression on the left is simply a function. This overloading of the equal sign with contextually dependent meanings may not be very elegant, but it does not lead to any confusion in practice.
of logarithms and our notation about function growth. From a problem solving perspective, it is informative.

We start with a few facts about logarithms.

0) Fact: The logarithm is an order preserving (i.e., monotone) function: if $0 < x < y$, then $\log x < \log y$. This follows from the definition of the logarithm: recall that $\log_b x$ is the (possibly fractional) number of $b$'s that when multiplied together give $x$. Larger values need more $b$'s. (We suppose, of course, that $b > 1$). A more precise and sophisticated proof comes from calculus, where we note that $\log_b x$ is increasing since its derivative is the positive function $\frac{1}{x \log_b b}$, which is positive as long as $b > 1$.

1) Fact: $\log n << n$, or more formally, $\log n = o(n)$, which means that $\lim_{n \to \infty} \frac{\log n}{n} = 0$.

Let, for specificity, the base of the logarithm be 2. We can prove this fact in many ways. A crude but adequate way is to observe what happens if we increase $n$ by a factor of 4, when $n > 4$. Evidently the denominator increases by a factor of 4. The numerator, however, is

$$\log_2(4n) = \log_2 n + \log_2 2^2 = \log_2 n + 2,$$

which increases by just 2. For $n \geq 4$, this value is at most double $\log_2 n$. Thus every time $n$ increases by a factor of four, $\frac{\log_2 n}{n}$ becomes half as large, or even smaller. The positive value decreases toward zero.

All this is evident at an informal level in the graphs of the two functions; $f(x) = x$ is a straight line, while $f(x) = \log x$ becomes flatter and flatter, since $\log_2 x$, which is the number of 2's needed to be multiplied together to get $x$, increases by a modest 1 when $x$ is doubled.

Well, so much for the harder part. Since the ratio goes to zero, we have that $c \log n = o(n)$. We can substitute $n = d^m$, for some fixed $d > 1$, which guarantees that $n$ grows as $m$ grows. The change of variables gives,

$$\log(d^m) = o(d^m),$$

which, by the rule of logarithms becomes, $m \log d = o(d^m)$.

We may drop the (positive) constant $\log d$ to get that

2) Fact: $m = o(d^m)$, for $d > 1$.

We may raise both sides to the power $r > 0$ to get $m^r = o(d^{rm})$, and observe that $d^{rm} = (d^r)^m$, and for arbitrary $d > 1$, $r > 0$, $d^r$ is just some other constant $c > 1$. Substituting gives

3) Fact: $m^r = o(c^m)$.

Example: $m^{1000000000} = o((1.000000001)^m)$, That is, any polynomial is eventually dwarfed by any (growing) exponential. We only require that the base be bigger than 1. Polynomial growth is any growth that can be bounded by $O(n^c)$ for some possibly huge constant $c$. Exponential
growth is of the form $c^n$, or equivalently $c^{dn}$, for some $c > 1$, $d > 0$. More precisely, we would say that $f$ grows exponentially if, for some $d > 1$ and $c > 1$, $f(n) = O(d^n)$, so that $f$ is bounded by some exponential and $c^n = O(f(n))$, so that $f$ grows at least as fast as some exponential.

A somewhat tedious, but technically simple method can allow us to compare the growth behavior of the kinds of functions that arise in computer science quite easily.

Suppose we wish to compare the growth of a pair of specific functions, for large and growing $n$. For example, which is eventually bigger, $.001n^3 - 100n^2$, or $1000n^2 + 400000000n^{3/2}$? Which grows faster, $n^{\log_2 n}$, or $\sqrt{\log(\log n)}$? These examples are more complicated than the overwhelming majority of function behavior encountered in computer science, although there are exceptions.

4) Technique: Dropping less significant terms.

To decide which is more powerful, an elephant or a lion plus an ant, we may (provisionally) drop the ant and compare the lion and the elephant. If we conclude that the elephant is more powerful than 2 lions, say, then the deleting of the ant was justified. If, after dropping all ants, a tie occurs, then we may have to back up to consider which set of deleted terms is more significant.

5) Reduction in strength, so that rule 4 (or 7) may be applied,

5.1) Simplifying exponentials.

When comparing, say, $(f(n))^{g(n)}$ and $h(n)$ to see which is eventually larger, taking the logarithm of both expressions may simplify the task. For then we are comparing $g(n) \log (f(n))$ and $\log (h(n))$. Upon renaming the functions, we are comparing $g(n) \times w(n)$ and $x(n)$, for suitable $w(n)$ and $x(n)$. The gain is that exponentiation is replaced by multiplication, which is conceptually simpler. This is a form of reduction in strength, which occurs in computation: for example, an optimizing compiler might replace the expression $2 \times x$ by $x + x$.

5.2) Simplifying products.

When comparing $f(n) \times g(n)$ with $h(n)$, we may be able to take the log of both sides, and thereby compare $\log (f(n)) + \log (g(n))$ with $\log (h(n))$. Then rule 4 may apply.

6) Technique: Changing variables.

Sometimes changing variables, in simple ways, will simplify the expressions being compared, and even change them into functions where the larger is already known. For example, selecting the larger of $n$ or $2^n$ is the same as selecting the larger of $\log_2 x$ or $x$: we set $n = \log_2 x$, or equivalently, $x = 2^n$. The need for substitution typically results from reductions in strength 5).

7) Technique: Simplify by applying the rules of algebra. Cancel common factors and terms.

**Examples**

We start with some examples to illustrate the point that these rules are more or less mechanical. Judgement, reason and experience can provide short cuts so that the basic steps become justifications for rapid estimations rather that operations to be meticulously applied.

$n^4$ versus $10000n^3$.

Divide by $n^3$. We see that $n$ grows faster than 1000. Alternatively, take the logarithm, which gives $4 \log n$ versus $3 \log n + \log 10000$. 

©Siegel, Cule 1994

rec/1AAFunct.tex
Drop the constant ant \( \log 10000 \). This leaves the simplified problem:

\[ 4 \log n \text{ versus } 3 \log n. \]

Cannell by dividing by \( \log n \) gives 4 versus 3. The 4 is bigger. Alternatively, cancel by subtracting \( 3 \log n \) from both sides gives \( \log n \) versus 0; the left hand side again wins.

\[ 1000000n^{1000} + 1000000000000000n^{99} \text{ versus } (1.000000000001)^{.000001n}. \]

Drop the ant \( 1000000000000000n^{99} \), which has an exponent that is too small (as is verifiable if we use the logarithm, the dropping of constant ants, and cancellation to compare these two terms). This leaves the simpler problem:

\[ 1000000n^{1000} \text{ versus } (1.000000000001)^{.000001n}. \]

Take the logarithm, and drop the constant ants. We get the simpler problem:

\[ 1000 \log n \text{ versus } .000001n \log(1.000000000001). \]

Observe that \( \log 1 = 0 \), the right hand side is positive, since the logarithm is being applied to a number slightly larger than 1. Take the logarithm, drop the constant ants. This leaves the simplified problem:

\[ \log \log n \text{ versus } \log n. \]

Let \( y = \log n \). This leaves the simplified problem:

\[ \log y \text{ versus } y. \]

The original problem has been reduced to a standard problem; the \( y \) (right hand side) wins.

\[ (\sqrt{n})^n \text{ versus } n^{n/2}. \]

Take the logarithm. We get:

\[ n \log \sqrt{n} = n \log(n^{1/2}) = \frac{n}{2} \log n \text{ versus } \sqrt{n} \log n. \]

Cancel logs. This leaves the simplified problem:

\[ \frac{n}{2} \text{ versus } \sqrt{n}. \]

(Could set \( n = x^2 \), to get \( x^{2/2} \text{ versus } x \)) Or take the logarithm. This leaves the simplified problem:

\[ \log n + \log \frac{1}{2} \text{ versus } \frac{1}{2} \log n. \]

Drop constant ant. This leaves the simplified problem:

\[ \log n \text{ versus } \frac{1}{2} \log n. \]

Cancel by subtracting \( \frac{1}{2} \log n \) from each side. This leaves the simplified problem:

\[ \frac{1}{2} \log n \text{ versus } 0. \]

The left hand side grows more rapidly.

Notice that these steps first drop additive ants, then (a logarithm later) drop what are multiplicative ants. The process could continue. Notice that the highest level of exponent (such as the \( z \) in \( w^{xz} \)) is the most important, since more applications of the logarithm are required to bring it down to the base level. Yet sometimes the lower levels could matter as well. These rules just provide a formal framework for accurately identifying the critical exponents to assert that, for example, \( 10000000000x \log x \ll 1000x^{1.2} \ll (1.000001)^x \).

It is possible to be careless by dropping some ants but not all; but this will only matter when there is a tie at the end, or the elephants are cancelled out. It is also worth noting that the notion of an additive ant can be very precise: \( f \) is an ant relative to \( g \) if and only if \( f = o(g) \).

** There are also circumstances where this crude comparison method is inadequate. L'Hospital's rule can improve the process, although the use of Taylor series approximations will give the same answer with less effort, because it allows inessential terms to be dropped and not carried about in
pointless exercises of differentiation.

Interestingly, these techniques are very powerful. Let \( A \) be the (smallest) algebra that contains the real numbers, the variable \( x \), and is closed under addition (if \( f \) and \( g \) are in \( A \), then \( f + g \) is on \( A \)), closed under multiplication, division, exponentiation (if \( f \) is in \( A \), then \( e^{f} \) is in \( A \)), max (if \( f \) and \( g \) are in \( A \), then so is the function \( h(x) = \max(f(x), g(x)) \)). We may also close the algebra under a restricted \( \log^{+} \), where, say, \( \log^{+}(f) = \log |f| \). Then any two functions in \( A \) are orderable: if \( f \) and \( g \) are in \( A \), then either \( f = o(g) \), or \( g = o(f) \), or \( g = \Theta(f) \). Moreover, the techniques listed in this section are powerful enough to prove the relationship! The algebra is quite large; for example, \( e^{\sqrt{1+x^2}} \) is in \( A \). On the other hand, \( A \) is not closed under infinite summations and, for example, the algebra does not contain \( \cos(x) \), which would destroy the property that any pair of functions is orderable.

Warning: not all constants are the same

While the process of taking logarithms can simplify the task of determining asymptotic behavior, it can also lead to errors, if carelessly done. For example, it is clearly the case that \( n^2 = o(n^3) \); these two functions do not satisfy the \( \Theta \) relation. Yet if we take the log of both sides, we get \( 2 \log n \) and \( 3 \log n \), which do satisfy \( 2 \log n = \Theta(3 \log n) \). The explanation for this, of course, is that changing an exponent by a constant factor is much more significant than increasing an expression by a constant factor. Abstractly, \( f = \Theta(g) \) does not ensure that \( 2^f = \Theta(2^g) \). This is the reason that the techniques for ordering functions by asymptotic growth typically used such notions as "bigger," and avoided, for the most part, the asymptotic formulations such as \( \Theta \), which are used throughout the text. As a practical matter, it is never particularly difficult to unwind a sequence of simplifications and logarithm applications to determine if two functions have an asymptotic growth that is different by a multiplicative constant or by a constant that involves more rapid growth due to different constants within some exponential formulation.

2.3 The Analysis of Algorithms and Recurrences

Once an algorithm is specified, for example, by writing it in a high level language, we will want to determine its complexity. In general, the complexity measure is an estimate of the running time of the algorithm, or more precisely, an estimate of the algorithm’s operation count. There are two steps required to attain this estimate. First, we must write a mathematical formulation that captures, up to constant factors, the number of computational steps executed by the algorithm. Second, we must solve the equation(s) that occur in the formulation, and simplify the resulting expressions.

For the most part, we are interested in techniques to analyze algorithms that are composed from a set of mutually recursive algorithms, which are recursive algorithms that can call each other. In contrast, algorithms comprised of fixed iteration loops are typically easy to analyze, and require very little mathematical sophistication. In what follows, we will concentrate on self-recursive algorithms (i.e., on the type of algorithm that calls itself), since these illustrate all the main ideas, and the generalization to mutually recursive algorithms is straightforward, although
the resulting equations typically must be solved by techniques that are beyond the scope of this course.

Let us begin by finding the recurrence equations that describe the running time for the following recursive formulation of Bubblesort. For the Bubblesort code listed below, we let $T(n)$ be the number of operations needed to execute the procedure on $n$ numbers, the (estimated) operation counts for each line of code, and the number of times each line is executed are shown on the right.

\begin{verbatim}
procedure Bubblesort(n: integer; var X[1..n] of real);
1    var j: integer;
2    if n > 1 then
3        for j ← 1 to n - 1 do
4            if X[j] < X[j + 1] then swap(X[j] and X[j + 1])
5        endfor;
6    Bubblesort(n - 1, X[1..n - 1])
7endif
end P.Bubble.
\end{verbatim}

Suppose $n > 1$. In this case, the \textit{if} statement in line 2 will be true. It requires, perhaps, one or two operations, since there is a comparison and a conditional jump. Line 3 might correspond to an increment, a test $j < n - 1$ and a conditional jump that are repeated $n - 1$ times. Line 4 is executed $n - 1$ times. It might include a successful comparison, and, possibly, three data transfers. Line 5 might represent an unconditional jump. The time to execute line 6 is, by definition just $T(n - 1)$. Obviously, the exact running time will be data (and compiler) dependent, but the cost of line 4 varies only by a factor of 3 (\textit{i.e.} one operation or four). Evidently, the lines that cause execution will initiate between 1 and perhaps 6 operations. We conclude that $T(n) < 2 + 9(n - 1) + T(n - 1)$, $n > 1$.

Our first effort at deriving an equation for $T$ might read:

\[
\begin{align*}
T(1) &= 2; \\
T(n) &= T(n - 1) + 9n - 6, \quad \text{if } n > 1.
\end{align*}
\]

Even in this simple case, we can only be certain that the operation count is accurate to within some moderate constant factor. (And even this "careful" formulation has been rather optimistic by ignoring the cost to call a procedure as in line 6; context switches typically take over 100 operations.) Moreover, for most of the algorithms we shall encounter, the most important properties of their running times will be the asymptotic growth, where constant factors are less significant than their asymptotic rate of growth. Running times $\Theta(T(n))$ will serve to identify the better algorithms, in most cases.

So we shall, for the most part, be ready to drop constant factors that we cannot handle precisely anyway. Since constants do not matter, we might as well write:

\[
\begin{align*}
T(1) &= 1; \\
T(n) &= n + T(n - 1), \quad \text{if } n > 1.
\end{align*}
\]

A more formal derivation would use inequalities (which are called recurrence relations); a more
Section 2.3

THE ANALYSIS OF ALGORITHMS AND RECURRENCES

37

conservative formulation might read:
\[
\begin{aligned}
1 & \leq T(1) \leq 2; \\
n + T(n - 1) & \leq T(n) \leq 9n + T(n - 1), \quad \text{if } n > 1.
\end{aligned}
\] (2.2)

Induction can be used to show that any function which satisfies formulation (2.2) must lie between \(T\) and \(9T\), where \(T\) solves (2.1). See Section 2.4.1. It turns out that constant factors can always be dropped in formulating the problem and that the inequalities in the recurrence relations can always be converted to recurrence equations (i.e. equalities) for these types of problems. The only requirement, essentially, is that the problem model operation counts so that no negative terms appear in the right-hand side for the expression for \(T(n)\). Section 2.4 formalizes these facts as a series of exercises.

It is very important, however, to understand that some constant factors matter tremendously. Consider the equation
\[
\begin{aligned}
T(1) &= C_1; \\
T(n) &= aT(n - 1) + bT(n - 2) + C_n, \quad \text{if } n > 1.
\end{aligned}
\]

The first equation is called the initial condition. Changing \(C_1\) by a constant factor, as we shall see, will only change the solution by at most that same factor. In the second equation, only one term, \(C_n\), is not multiplying an expression in \(T\). This term, called the inhomogeneous term, also can be changed by a constant factor and the solution will only be changed by a factor that is the same or closer to 1. We will see that the solution is much more sensitive to factors such as \(a\) and \(b\), which multiply expressions in \(T\).

Most self-recursive algorithms can be written in the following schematic form:

\[
\begin{aligned}
\text{procedure } P(X : \text{data}); \\
\quad \text{Statement1;} \\
\quad P(X_1); \\
\quad \text{Statement2;} \\
\quad P(X_2); \\
\quad \text{Statement3;} \\
\quad \vdots \\
\quad \text{Statementl;} \\
\quad P(X_l); \\
\quad \text{Statement}(l + 1); \\
\end{aligned}
\]

\text{end}_P.

We can think of a program as transformed into this form after resolving if-then-else's and unrolling while-loop's; that is, the \(l\) pairs [Statement\text{j}, \(P(X_j)\)] may be coded in a loop that iterates \(l\) times, and the value of \(l\) might be even be some function \(\text{Size}(X)\).

This schema is somewhat simplistic, but not overly restrictive. Let \(\text{Size}(X) = n\), \(\text{Size}(X_1) = n_1\), \(\ldots\), \(\text{Size}(X_l) = n_l\). Assume that the time spent by \(\text{Statement1}, \text{Statement2}, \ldots, \text{Statement}(l + 1)\) is overestimated by \(f_k(n)\). Then we can formulate the complexity of the algorithm as follows:

\[
T(n) \leq \sum_{i=1}^{l} T(n_i) + f(n).
\]
Such an inequality is called a recurrence relation.

Because each $T(n_i)$ term on the right-hand side of the equation for $T(n)$ is positive, the inequality can be replaced by an equality and the solution will still be an overestimate for $T$. (Verifying this fact will be a home work problem. The point is that the change from an inequality to an equality must give larger values for $T$, and increasing each $T(n_i)$ can only increase $T(n)$, and cannot possibly decrease the resulting value.) So we will consider recurrence equations of the form:

$$T(n) = \sum_{i=1}^{l} T(n_i) + f(n).$$

In Section 2.3.3, we present techniques to solve the kinds of recurrence equations that typically arise in computer science. But first, we elaborate on how to transform a recursive program into a recurrence equation that governs its running time.

### 2.3.1 Purely iterative Programs

As a preliminary step to the derivation of recurrence equations, we will consider the running time for a few examples of iterative code. Consider the following code fragment.

1 var $A[1..n]$ of integer;
2 for $i \leftarrow 1$ to $n$ do
3 \hspace{1em} $A[i] \leftarrow i$
4 endfor;
5 for $j \leftarrow 1$ to $n$ do
7 endfor;

The code comprises two consecutive loops. Each loop has $\Theta(1)$ operations per iteration. The running time for this code, up to a constant factor can be formulated as

$$T(n) = \left( \sum_{1 \leq i \leq n} 1 \right) + \left( \sum_{1 \leq j \leq n} 1 \right) = \Theta(n).$$

Now consider the code to assign zeros to the upper triangle of an $n \times n$ array:

1 var $A[1..n, 1..n]$ of integer;
2 for $i \leftarrow 1$ to $n$ do
3 \hspace{1em} for $j \leftarrow i$ to $n$ do
4 \hspace{2em} $A[i, j] \leftarrow 0$
5 endfor;
6 endfor;

Here we have nested loops. The running time for this code, up to a constant factor can be formulated as

$$T(n) = \sum_{1 \leq i \leq n} \left( \sum_{1 \leq j \leq n} 1 \right) = \sum_{1 \leq i \leq n} (n - i + 1).$$
Here the two summations are nested because the loops are. The bounds on the summation are also copied directly from the loops. The operation count for the innermost instruction is taken to be the constant 1. To formulate a summation that captures the running time of an iterative program, we simply change every loop into a summation, and note the scope of each loop, so that nested loops yield nested summations, and sequential loops yield a sum of summations.

Finally, let \( \text{zoop}(n) \) have a running time of \( f(n) \), up to a constant factor. Now consider the code:

```plaintext
1  var A[1..n, 1..n] of integer;
2  for i ← 1 to n do
3     for j ← i to n do
4         A[i, j] ← 0
5     endfor;
6  zoop(i);
7  endfor;
```

The running time for this code, up to a constant factor can be formulated as

\[
T(n) = \sum_{1 \leq i \leq n} \left( f(i) + \sum_{i \leq j \leq n} 1 \right).
\]

If \( f(i) \) is smaller than \( \sum_{i \leq j \leq n} 1 = n + 1 - i \), we might drop the term, since the sum would only change by a constant factor. Similarly, if \( f(i) \) dominated the summation, we could elect to keep the \( f(i) \) term and drop the inner summation. We might also be forced to keep both terms, if each is sufficiently larger in various number ranges.

As these examples suggest, the work to execute loops over fixed index ranges maps directly into fixed summations. Rather little technique is required to formulate their operation counts as explicit summations. The Appendix presents, in terse form, formulas and brief explanations for evaluating the overwhelming majority of summations that actually arise in this context.

When the running time of a recursive program is sought, a little more technique is necessary.

### 2.3.2 Deriving Recurrence Equations

Recurrence equations that bound running times or that measure other properties of recursive programs can almost always be written down simply by observing (and quantifying) what happens at the top level of a recursive call. There are, as a rule, two cases: no recursive calls are to be made, and the general case, where recursive calls occur. The first case gives the initial condition, and the second the recursive part of the recurrence equation.
Consider the following code fragment, for even \( n \).

**procedure** Mergesort\((n: \text{integer}; \text{var} \ X[1..n] \text{ of real})\);

\[ \text{var} \ A[1..n/2], B[1..n/2] \text{ of integer}; \]

\[ \text{if} \ n > 1 \text{ then begin} \]

\[ A \leftarrow X[1..n/2]; \quad \text{Copy the first half of} \ X \text{ into} \ A. \]

\[ B \leftarrow X[n/2 + 1..n]; \quad \text{Copy the second half of} \ X \text{ into} \ B. \]

\[ \text{Mergesort}(n/2, A); \]

\[ \text{Mergesort}(n/2, B); \]

\[ X \leftarrow \text{Merge}(n, A, B) \]

**end Mergesort.**

The procedure \( \text{Merge} \) runs in \( \Theta(n) \) time.

Let \( T(n) \) be, up to a constant factor, a bound of the running time for Mergesort. We see that \( T(1) = 1 \), since the procedure just exits if \( n = 1 \). For larger values of \( n \), the procedure, at level \( n \), calls Mergesort twice, each with data sets of size \( n/2 \). Then it calls for \( \Theta(n) \) additional work via \( \text{Merge} \). We could therefore write that the operation count satisfies: \( T(n) = T(n/2) + T(n/2) + n \). A more careful estimate might charge \( n \) work to copy the array data from \( X \) into two arrays of size \( n/2 \) in the recursive calls. Each procedure call requires a (nontrivial) constant number of operations even if no data is transferred, so a more general formulation would be: \( T(n) = T(n/2) + T(n/2) + cn + 3d \).

Constant \( c \) covers all movement of the \( n \) items of data, and \( d \) is the overhead for a procedure call. We will see in the next section that the constants \( c \) and \( d \) can only affect the solution by a constant factor. It suffices to use the equation \( T(n) = 2T(n/2) + n \). The factor 2, which multiplies the recursively defined \( T(n) \) will turn out to affect the solution enormously. We cannot be so casual about its value. In any case, the principle needed to write a correct recurrence equation is as follows. The charges for \( T(n) \) are read directly from the algorithm and formulated as the right hand side of the equality \( T(n) = \text{expression} \).

![This expression is just a count of the operations at the current level \( n \), plus an accounting, in terms of \( T \), for each recursive call issued from the level \( n \) code.]

Thus operations at inner levels will be recursively charged by the recurrence equation at deeper levels of the recurrence.

There is an abstract conceptualization very well worth making, because it appears in so many contexts and is so useful. The simple definition of \( T(n) \) as the running time (up to constant factors) of the algorithm is a kind of encapsulation. The analysis of the running time only requires examination at a single level of the algorithm, level \( n \). What happens at level \( n \)? Well if \( n = 1 \), we see that a constant number of operations occur; but no more. For bigger values of \( n \), we see two calls to Mergesort plus an additional \( \Theta(n) \) operations to merge the two sorted subsets. Thanks to our encapsulation, we need not look into the actions that occur as a consequence of the two calls to Mergesort. The equation for \( T(n) \) is fully determined by the algorithmic activities at level \( n \).

As a more instructive example, consider the following code fragment.
function Rand(N);
  if N \leq 1 then return(1)
  else
    assign x one of the values 0, 1, 2, each with probability \( \frac{1}{3} \);
    if x = 0 then return(Rand(N))
    elseif x = 1 then return(Rand(N - 1) + 1)
    elseif x = 2 then return(3 \times Rand(N - 1) + Rand(N - 1) + 1)
  endif
  endif
end Rand;

Three problems we might consider are:

a) Give the recurrence equation for the asymptotic expected running time of Rand, which means the average running time, ignoring constant factors.

b) Write down the EXACT recurrence equation for the expected number of recursive calls executed by a call to Rand(N). Recall that a recursive call is a call from inside the code that defines Rand, so that the original call to evaluate Rand(N) does not count as recursive.

c) Find the exact recurrence equation for the expected number of times the return on line 7 is executed, in all calls to Rand, recursive or not.

a) This case is actually no different from Mergesort. If n = 1, the work is just a constant, which we may take to be 1. For higher levels, there is constant work plus the work from recursive calls. Here there are three cases, which each occur with probability \( \frac{1}{3} \). (If you play a three-way game where each possibility occurs with probability \( \frac{1}{3} \), and have winnings X, Y, and Z in the three cases, then the expected winnings is \( \frac{X+Y+Z}{3} \), and this is true even if the X, Y, and Z are themselves expectations (averages) of random winnings.) Formally, the solution is:

\[
\begin{align*}
  T(n) &= 1 + \frac{T(n)}{3} + \frac{T(n-1)+1}{3} + \frac{2T(n-1)+2}{3}, \quad \text{if } n > 1. \\
\end{align*}
\] (2.3)

We will simplify this recurrence in a minute. First, a few comments are needed. The \( (T(n-1) + T(n-1)) \) (divided by 3) comes from the two individual calls to Rand in the case x = 2. Multiplying the Rand value by 3 takes unit time, but calling the function twice does just that: it initiates two recursive computations. We cannot expect a compiler to reuse the value from the first call to Rand(n - 1) because the compiler may not be able to tell (in general) if a function call causes side effects, which might even cause two different answers to be returned. The second remark is that a more conservative accounting might charge for the multiplication and addition that occur in the individual branches, so that one might write: \( T(n) = 1 + \frac{T(n)}{3} + \frac{T(n-1)+1}{3} + \frac{2T(n-1)+2}{3} \), if \( n > 1 \). This is of the form \( T(n) = c + \frac{T(n)}{3} + \frac{T(n-1)+1}{3} + \frac{2T(n-1)+2}{3} \). As we will soon show, all constants (other than those multiplying T) can be set to 1.

To simplify the recurrence, we subtract \( \frac{T(n)}{3} \) from both sides, multiply by 3/2, reset our additive constant to 1, and get:

\[
\begin{align*}
  T(1) &= 1; \\
  T(n) &= 1 + \frac{3T(n-1)}{2}, \quad \text{if } n > 1. \\
\end{align*}
\] (2.4)
b) Here we use exact constants, and must be careful to charge only what happens at level $n$, and add the subsequent charges recursively. If $n = 1$, then there are no additional recursive calls: $R(1) = 0$. At level $n$, there is one call if $x = 0$ plus the subsequent calls to compute (the recursive call to) $\text{Rand}(n)$. If $x = 1$, there is a single recursive call plus the subsequent calls to compute (the recursive call to) $\text{Rand}(n - 1)$. If $x = 2$, there are two recursive calls plus the subsequent calls to compute (the two recursive calls to) $\text{Rand}(n - 1)$. Each possibility occurs with probability $1/3$. We get:

$$\begin{align*}
\begin{cases}
R(1) = 0; \\
R(n) = \frac{1 + R(n)}{3} + \frac{1 + R(n - 1)}{3} + \frac{2 + 2R(n - 1)}{3}, & \text{if } n > 1.
\end{cases}
\end{align*}$$

(2.5)

Simplifying gives:

$$\begin{align*}
\begin{cases}
R(1) = 0; \\
R(n) = \frac{2n}{2} + \frac{3R(n - 1)}{2}, & \text{if } n > 1.
\end{cases}
\end{align*}$$

(2.6)

Because we are computing an exact value, nothing is simplified to be one.

c) We again use exact constants. Let $C(n)$ be the expected number of returns from line 7 of $\text{Rand}$, in all calls. The unsimplified solution reads:

$$\begin{align*}
\begin{cases}
C(1) = 0; \\
C(n) = \frac{C(n)}{3} + \frac{C(n - 1)}{3} + \frac{1 + 2C(n - 1)}{3}, & \text{if } n > 1.
\end{cases}
\end{align*}$$

(2.7)

The initial condition reads $C(1) = 0$ because the return of line 7 is not executed when $n = 1$. For larger $n$, the return in line 5 is executed with probability $\frac{1}{3}$. Although this return is not to be counted, it does issue a call to $\text{Rand}(n)$, which will, on average, execute $C(n)$ returns that must be counted. Similarly, line 6 will contribute $C(n - 1)$ such returns with probability $\frac{1}{3}$. Line 7 initiates, with probability $\frac{1}{3}$, a single return that must be counted along with the recursive returns that occur from two independent calls to $\text{Rand}(n - 1)$. (If we were counting the number of recursive calls from line 7, the term would be $\frac{2 + 2R(n - 1)}{3}$, because the line issues two recursive calls, and causes an expected $2R(n - 1)$ additional calls to occur.)

2.3.3 Solving Recurrence Equations

Recurrence equations (also called difference equations) arise not only in computer science, but also in many other technical fields, such as combinatorics, probability theory, discrete-time control theory and economics. The mathematical techniques available to solve these problems include summation factors, generating functions, $z$-transformations, operator methods, and more. But most of these techniques are beyond the scope of this text. We develop only a few techniques—though simple, they are sufficient to solve more than 95% of the problems that occur in computer science.

A method that will be discouraged is guess-work.

Guess-Work

The idea is to guess a solution, and then verify the guess by a mathematical induction over the integers. This method requires you to know the solution in order to get the solution, and does not work. Use it only when you have no other way of solving the equation—try several guesses; if you
are lucky, one of them will be correct. If you are luckier still, the inductive proof will not be too gruesome.

**Example. 2.3.1** Consider the following recurrence equation:

\[
\begin{align*}
T(1) &= c_1; \\
T(n) &= 2T\left(\frac{n}{2}\right) + c_2 n, \text{ if } n > 1.
\end{align*}
\]  

(2.8)

Suppose we guess that \(T(n) = c_2 n \log_2 n + c_1 n\), then by mathematical induction over all positive integers, we can show the validity of our guess. Hence \(T(n) = O(n \log_2 n)\). It is left as an exercise for you to do the steps of the inductive proof.  

**Telescoping**

The techniques of **telescoping, domain and range transformations** and **differencing** are rather simple, and are developed step-by-step in this and the next section. We start out by showing how to use telescoping to solve a very simple recurrence equation. But this method, when combined with the transformation and differencing techniques of the next section, becomes quite powerful and handles almost all equations in computer science; indeed, every recurrence equation that occurs in this or any of the standard algorithms texts can be solved by these methods. Please master this technique by exercising it on several recurrence equations—**this is the official technique for this text**.

Consider the following system, which we shall call the World’s Simplest Recurrence:

\[
\begin{align*}
T(0) &= 1; \\
T(n) &= T(n-1) + 1, \text{ if } n \geq 1.
\end{align*}
\]  

(2.9)

We can write down the above equation for the arguments \(n, n - 1, n - 2, \ldots, 1\) as follows:

\[
\begin{align*}
T(n) &- T(n-1) = 1 \\
T(n-1) &- T(n-2) = 1 \\
&\vdots \\
T(1) &- T(0) = 1
\end{align*}
\]

Adding these equations gives:

\[
\frac{T(n)}{T(n)} - T(0) = n
\]

The simplification is obtained by canceling \(-T(i)\) of the current line by the \(T(i)\) of the next line. All but the first and last terms cancel out when the complete sequence of equations is added together. We say the sum telescopes. Because of the telescoping, the resulting sum reads \(T(n) - T(0) = n\), and the final answer is \(T(n) = n + T(0) = n + 1\).

Finding the solution above was almost trivial. Notice that the 1’s on the right-hand side could actually be any expression that does not contain \(T\). Furthermore, the initial condition \(T(0) = 1\) could just as well have been \(T(m) = a_m\), for some fixed constant \(m\). In this case, we simply telescope the equations for \(T(n), T(n - 1), \ldots, T(m + 1)\). Consider, for example:

\[
\begin{align*}
T(m) &= a_m; \\
T(n) &= T(n-1) + a_n, \text{ if } n > m.
\end{align*}
\]  

(2.10)
If we again write these equations for arguments \( m, m - 1, m - 2, \ldots, n \) and sum them, telescoping occurs to give
\[
T(n) - T(m) = a_{m+1} + \cdots + a_n, \quad n \geq m,
\]
so that
\[
T(n) = a_m + a_{m+1} + \cdots + a_n, \quad n \geq m, \tag{2.11}
\]
The \( a_m \)'s can be any expression in \( m \). For Bubblesort and equation (2.1), we observe that \( a_j = j \), for \( j = 1, 2, \ldots, n \). Consequently, \( T(n) = 1 + 2 + 3 + \cdots + n \). According to the Appendix, this sum equals \( (n)(n + 1)/2 \), so that \( T(n) = \Theta(n^2) \). Evidently, it is helpful to be able to express sums in closed form (i.e. by a formula), when possible, and to approximate them when necessary. The Appendix contains a selection of such sums, together with some information on how the results are derived.

Notice that the telescoping method applies for expressions of the form
\[
\text{expr}(n) - \text{expr}(n - 1) = c_n,
\]
that is, the cancellation works as long as we have a difference of two terms where the second term is exactly the same as the first, except that there is an \( n - 1 \) in the second expression wherever there is an \( n \) in the first. We could just as easily use telescoping to solve:
\[
\begin{align*}
T(1) &= a_1; \\
n^2 \log(T(n) + n) &= (n - 1)^2 \log(T(n - 1) + n - 1) + a_n, \quad \text{if } n > 1. \tag{2.12}
\end{align*}
\]

Sometimes we have to use a little trickery to make the sum of successive equations telescope. This trickery is actually comprised of three techniques, \textit{domain transformations}, \textit{range transformations} and \textit{differencing}. These techniques give a systematic way to reduce a huge number of recurrence equations to the World's Simplest Recurrence. They cannot solve everything, but are adequate for well over 95% of the recurrences that arise in Computer Science.

\textbf{Exercise} Suppose some of the \( a_i \)'s, in equation (2.11), are changed in value by as much as 10%. What is the largest percentage change that will result in the sum \( T(n) \)? Does the answer depend on whether some of the \( a_i \)'s can be negative?

\textbf{Transformations and Differencing}

Sometimes a change of variables can transform one recurrence equation into another that we already know how to solve. The function \( T \) maps the integer \( n \) in its \textit{domain} to a real \( T(n) \) in its \textit{range}. We call a transformation (or change of variable) for the index \( n \) a \textit{domain transformation} and a transformation on the values of the sequence \( T(n) \) a \textit{range transformation}. We illustrate the ideas using examples.

\textbf{Domain Transformations}

In binary search of a sorted set, we halve the size of the search range by testing the middle item against the search key, and selecting either the upper range, or the lower, (or the middle item itself), depending on the result of the outcome. A (slightly loose) formulation for the running time would read:
Example. 2.3.2

\[
\begin{align*}
T(1) &= 1; \\
T(n) &= T\left(\frac{n}{2}\right) + 1, \quad \text{if } n \geq 2.
\end{align*}
\] (2.13)

The evident difficulty with the recurrence equation as stated is that the \( T\left(\frac{n}{2}\right) \) term is in the wrong form; we prefer \( T(n-1) \). This suggests that we change variables by defining \( k \) and \( S \) so that \( n = 2^k \), \( k = \log_2 n \), and \( S(k) = T(2^k) \). Hence \( T(n) = T(2^k) = S(k) \), and \( T\left(\frac{n}{2}\right) = T(2^{k-1}) = S(k-1) \). The above equation becomes

\[
\begin{align*}
S(0) &= 1; \\
S(k) &= S(k-1) + 1, \quad \text{if } k \geq 1.
\end{align*}
\]

Now we can use telescoping to conclude that \( S(k) = k + 1 \). Back-substituting gives \( T(n) = S(k) = k + 1 = \log_2 n + 1 \), for \( n \) a power of 2.

Suppose the original system had been:

Example. 2.3.3

\[
\begin{align*}
T(1) &= 1; \\
T(n) &= T\left(\frac{n}{2}\right) + n, \quad \text{if } n > 1.
\end{align*}
\] (2.14)

The same approach works for this case, and we again let \( n = 2^k \), \( (k = \log_2 n) \) and \( T(n) = S(k) \). We get:

\[
\begin{align*}
S(0) &= 1; \\
S(k) &= S(k-1) + 2^k, \quad \text{if } k \geq 1.
\end{align*}
\]

Now we can use telescoping to get:

\[ S(k) - S(0) = 2 + 4 + 8 + 16 + \cdots + 2^k, \]

so \( S(k) = 1 + 2 + 4 + 8 + 16 + \cdots + 2^k \). The Appendix shows how to sum this geometric series. The answer is \( S(k) = 2^{k+1} - 1 \). Since \( S(k) = T(2^k) \), back-substituting \( n \) for \( 2^k \) and \( \log_2 n \) for \( k \) gives,

\[ T(n) = 2n - 1. \]

Evidently, the \( T\left(\frac{n}{2}\right) \) term in (2.14) could just as well have been \( T\left(\frac{n}{c}\right) \) for any fixed \( c > 1 \), and the corresponding transformation would be \( n = c^k \).

Range Transformations

Recall that the Tower of Hanoi puzzle is the following. There are three poles, named \( A \), \( B \), and \( C \). A stack of \( n \) rings (disks, with a hole punched through the middle) sit on pole \( A \). The disks have increasing diameters, so that the largest is at the bottom of the stack. There two rules:

i) Only the top ring on any stack can be moved to any other stack.

ii) A ring can be placed on a stack only if it is empty or the top ring has a larger diameter.

The objective is to transfer the rings from pole \( A \) to \( B \).

How can we solve the problem, and how long will such a task take? The recursive algorithm is listed below. A little more discussion of the recursion is found in Chapter 1.4. In the following
high level algorithm, \( S\left[\frac{1}{n}\right] \) is again the stack of rings on the first pole; they are numbered in order, with the bottom ring assigned \( n \).

\[\text{procedure TowersOfH}(n: \text{stack height}; \ \text{var} \ A, B, C: \text{pole}; \ \text{var} \ S\left[\frac{1}{n}\right]: \text{stack of } n \text{ rings on pole } A)\]  
\{ Moves stack of \( n \geq 1 \) rings from \( A \) to \( B \). \}
if \( n = 1 \) then move the top ring on \( A \) to \( B \)
else
\begin{align*}
\text{TowersOfH}(n - 1, A, C, B, S\left[\frac{1}{n-1}\right]); \\
\text{move the top ring on } A \text{ to } B; \\
\text{TowersOfH}(n - 1, C, B, A, S\left[\frac{1}{n-1}\right])
\end{align*}
endif
end.TowersOfH.

The recurrence equation for the running time reads,
\[
\begin{cases}
T(1) = 1; \\
T(n) = 2T(n - 1) + 1, \text{ if } n > 1.
\end{cases}
\]

This recurrence equation is not quite in our standard form because of the factor 2 in the term \( 2T(n - 1) \). In this case, another trick is needed to transform the equation \( T(n) - 2T(n - 1) = 1 \) into one that telescopes. Upon multiplying the equation by the \textit{summation factor} \( 2^{-n} \), there results, \( 2^{-n}T(n) - 2^{-(n-1)}T(n - 1) = 2^{-n} \). This expression has the form \( expr(n) - expr(n - 1) = c_n \). So making the range transformation \( S(n) = 2^{-n}T(n) \), we get:
\[
\begin{cases}
S(1) = \frac{1}{2}; \\
S(n) = S(n - 1) + \frac{1}{2^n}, \text{ if } n > 1.
\end{cases}
\]

Telescoping gives the summation
\[
\begin{align*}
S(n) &= \frac{1}{2^n} + \frac{1}{2^{n-1}} + \frac{1}{2^{n-2}} + \cdots + \frac{1}{2^2} + S(1) \\
&= \frac{3}{2^n} + \frac{1}{2^{n-1}} + \frac{1}{2^{n-2}} + \cdots + \frac{1}{2^2} + \frac{1}{2} \\
&= 1 - \frac{1}{2^n},
\end{align*}
\]
where we used the Appendix to compute the sum of the geometric series. Back-solving gives \( T(n) = 2^nS(n) = 2^n - 1 \). It is easy to check that the initial condition is satisfied, and the answer is correct for small \( n \); we could even verify the answer by induction. It is indeed correct. Even if the Monks manage to move their 64 disks at the rate of 1 move per second, their task will take some \( 2^{64} \approx 25 \times 10^{18} \) seconds \( \approx 800 \text{ billion years} \). Incidentally, recall from Chapter 1.4 that their recursive algorithm is optimal; the problem cannot be solved with fewer moves.

Returning to more earthly concerns, we have seen that Bubblesort has a quadratic running time. Mergesort is a more interesting algorithm. It sorts an array of \( n \) elements by divide-and-conquer. The algorithm divides the input data into two equally sized sets, recursively sorts both, and then
completes the sort by merging the two sorted subsets:

procedure Mergesort(n: integer; var X[1..n] of real);
  var A[1..n/2], B[1..n/2] of integer;
  if n > 1 then begin
    A ← X[1..n/2];  \textit{Copy the first half of} X \textit{into} A.
    B ← X[n/2 + 1..n];  \textit{Copy the second half of} X \textit{into} A.
    Mergesort(n/2, A);
    Mergesort(n/2, B);
    X ← Merge(n, A, B)
  endif
end Mergesort.

The function \textit{Merge} is quite simple; it merges the two sorted arrays \textit{A} and \textit{B} by starting with location 1 in each, checking to see which element is smaller, and incrementing the corresponding array index to search for the next smallest value. Since each comparison produces a new element for the merged set \textit{X}, the total time for \textit{Merge} is \( \Theta(n) \). The recurrence equation \footnote{We are only going to solve for \( T(n) \) when \( n \) is a power of 2. Strictly speaking, the recurrence equation is not formulated for other values of \( n \). A more precise formulation would read,}

\[
\begin{align*}
  T(1) &= 1; \\
  T(n) &= 2T\left(\frac{n}{2}\right) + n, \text{ if } n > 1.
\end{align*}
\]  

(2.15)

Applying the domain transformation \( k = \log_2 n \), \( S(k) = T(2^k) \), gives

\[
\begin{align*}
  S(0) &= 1; \\
  S(k) &= 2S(k-1) + 2^k, \text{ if } k > 0.
\end{align*}
\]  

(2.16)

This recurrence equation is still not quite in our standard form because of the factor 2 in the term \( 2S(k-1) \). Upon multiplying the equation by the \textit{summation factor} \( 2^{-k} \), there results, \( 2^{-k}S(k) - 2^{-(k+1)}S(k-1) = 1 \). This is has the form \( expr(k) - expr(k-1) = c_k \). So making the range transformation \( R(k) = 2^{-k}S(k) \), we get:

\[
\begin{align*}
  R(0) &= 1; \\
  R(k) &= R(k-1) + 1, \text{ if } k > 0.
\end{align*}
\]  

(2.17)

Thus \( R(k) = k + 1 \); back-substituting gives \( S(k) = (k+1)2^k \), and \( T(n) = n \log_2 n + n \). So Mergesort has a \( \Theta(n \log_2 n) \) running time.

It is worth emphasizing that \footnote{It is not difficult to show inductively that \( T(2^{k-1}) \leq T(n) \leq T(2^k) \), where \( k = \lceil \log_2 n \rceil \). Furthermore, since \( T(n) = \Theta(T(2n)) \) for the typical kinds of problems we study, it suffices to solve for \( T \) on the sparse sets such as \( n = 2^k, k = 1, 2, \ldots \); see Section 2.4.2.}

\textbf{domain transformations should be done before range transformations;}

a domain transformation can destroy the benefit of a preceeding range transformation, but not vice versa.
Mistakes to avoid. Most of the common recurrence equations that occur in computer science are easy to solve. A little discipline can help eliminate the more common errors that students make when solving these systems. The first tip is to change variables whenever necessary, and to never reuse variable names. If the problem starts out with the identifier $T$, there is plenty of room to switch to, $V$, and to use $W$ for the next transformation of the function, if necessary. While this wastes letters, it helps at back-solving time, by creating a chain of substitutions where intermediate steps are unlikely to be missed. The same is true for the domain transformations: a function of $n$ can be replaced by $k$, which, in turn can give way to $j$, etc. (Are $m$ and $o$ good letters, do you suppose?)

The next observation is that initial conditions deserve to undergo the same sequencing of substitutions that is applied to the recurrence equation. Frequently an initial condition $T(1) = 1$ winds up as $V(0) = 1$ several transformations and substitutions later. If the initial condition is not brought along with the step-by-step transformations, there is a temptation to select the initial condition carelessly as, say, $V(1) = 1$.

Always compute an exact solution if possible. That way you can check the formula with the hand computed answers, for, say $n = 1, 2, 3$. While this checking does not guarantee that the answer is correct, it is less work than induction, and will catch most errors.

Donald Knuth recommends that this simple check be executed frequently, if possible. For example, for equation (2.15), we have that $T(1) = 1$, $T(2) = 2 \times 1 + 2 = 4$, and $T(4) = 2 \times 4 + 4 = 12$. We note that the values $n = 1, 2, 4$ correspond to $k = 0, 1, 2$, and can check that the recurrence for $S(k)$ gives $S(0) = 1$, $S(1) = 4$, and $S(2) = 12$, which is indeed the case. When we get to $R$, in equation (2.17), we see that $R(0) = 1$, compute that $R(1) = 1 + 1 = 2$, and $R(2) = 3$. Back-substituting for $S(k) = 2^k R(k)$ again gives the right values for $S$. Finally, we check the formula $T(n) = n \log_2 n + n$ for $n = 1, 2, 4$, and see that all is well. This example was actually too simple to need this much checking. On the other hand, this checking can also be applied later in a search for errors, if, for some reason, the formula fails to agree with the hand computed answers.

For, say a ten page calculation (which will never arise in this text), such checking may be helpful every half page or so. This extra investment of time is worth the effort, for if a mistake occurs on page 1, of what good are the following nine pages?

Differencing

It is worth observing that the method of telescoping uses summation to invert a difference operator. More formally, given a function $T(\cdot)$, we may define $(\Delta T)(n) = T(n) - T(n-1)$. This says that if we take the difference operator $\Delta$ and apply it to $T$, we get a new function whose value for argument $n$ is $T(n) - T(n - 1)$. We have seen how to solve for $T$, given the initial condition $T(0) = c_0$, and the difference equation $(\Delta T)(n) = c_n$. We solve for $T$ by summing (and adding a constant): $T(n) = T(0) + \sum_{0 < j \leq n} (\Delta T)(j)$. Just as integration and differentiation are inverse operators, so are the summing and differencing operators. In fact, the similarity is even closer, since the differentiation and integration operations are just summation and differencing when taken to the limit. The point of this observation becomes apparent when we analyze the expected running time for Quicksort. Since the algorithm deserves a detailed explanation, we will present the resulting recurrence equation, and save the algorithmic issues for the section on sorting. Before solving the equation that results from Quicksort, we will study a slightly simpler example.
Example 2.3.4 Consider the following recurrence equation:

\[
\begin{cases}
T(1) = 1; \\
T(n) = n + \sum_{1 \leq j < n} T(j), & \text{if } n > 1.
\end{cases}
\]  

(2.18)

The apparent difficulty with this recurrence equation is the presence of the terms \(T(1), T(2), \ldots, T(n-1)\) in the expression for \(T(n)\). We have only studied how to solve equations where one term was present. The solution is to use differencing to eliminate the summation operator. We write

\[
T(n) = n + T(1) + T(2) + \cdots + T(n-2) + T(n-1)
\]

\[
T(n-1) = n - 1 + T(1) + T(2) + \cdots + T(n-2)
\]

Subtracting gives:

\[
\frac{T(n) - T(n-1)}{n} = \frac{1}{T(n-1)}
\]

the summation has been differenced away. We have \(T(n) = 2T(n-1) + 1\). A standard range transformation shows that \(T(n) = 2^n - 1\).

When analyzing the average case running time for Quicksort, the following difference equation arises:

\[
\begin{cases}
T(1) = 1; \\
T(n) = n + \frac{2}{n} \sum_{1 \leq j < n} T(j), & \text{if } n > 1.
\end{cases}
\]

A first effort might be to try to form the difference \(T(n) - T(n-1)\), but the resulting terms will not quite cancel. The reason is that the sums in the two expressions are multiplied by different coefficients (i.e. \(\frac{2}{n}\) and \(\frac{2}{n-1}\)). Consequently, we first multiply the equations by \(n\) and \(n-1\) (respectively) and then subtract. We have:

\[
nT(n) = n^2 + 2T(1) + \cdots + 2T(n-2) + 2T(n-1)
\]

\[
(n-1)T(n-1) = (n-1)^2 + 2T(1) + \cdots + 2T(n-2)
\]

Subtracting gives:

\[
\frac{nT(n) - (n-1)T(n-1)}{2} = 2n - 1
\]

so

\[
nT(n) - (n+1)T(n-1) = 2n - 1.
\]

This last equation is not quite in the correct form; while \(nT(n)\) is clearly an expression in \(n\), the term \((n+1)T(n-1)\) is not the same expression with \(n-1\) replacing \(n\). A summation factor is needed. Now there is a complicated formula that can be used to produce the appropriate factor, but with a few observations, we can produce the appropriate expression by inspection. The difficulty is in the factors \(n\) and \((n+1)\); the \((n+1)\) should be associated with the \(T(n)\) and the \(n\) should be associated with the \(T(n-1)\). Accordingly, we multiply each term in the equation by the summation factor \(\frac{1}{n(n+1)}\). The result is

\[
\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2n - 1}{n(n+1)}.
\]

This is in the correct form. Letting \(S(n) = \frac{T(n)}{n+1}\) and rewriting the right-hand side slightly, we get:

\[
\begin{cases}
S(1) = \frac{1}{2}; \\
S(n) = S(n-1) + \frac{2}{n+1} - \frac{1}{n(n+1)}, & \text{if } n > 1.
\end{cases}
\]
Thus \( S(n) = \sum_{2 \leq k \leq n} \frac{2}{k+1} - \sum_{2 \leq k \leq n} \frac{1}{k(k+1)} + \frac{1}{2} \). From the Appendix, we see that the first sum is within a constant of 2 log \( n \), and the second sum is bounded by a constant; so \( S(n) \) is within an additive constant of 2 log \( n \): \( S(n) = 2 \log n - \Theta(1) \). Thus \( T(n) = \Theta(n \log n) \).

There is a subtle mistake that is easy to make when differencing. Let us reexamine example 2.3.4, but with just one simple change:

\[
\begin{align*}
T(1) &= 1; \\
T(n) &= n + 1 + \sum_{1 \leq j < n} T(j), \quad \text{if } n > 1.
\end{align*}
\]

(2.19)  
(2.20)

Naturally we can difference and solve as before. But if we follow these steps blindly, we will get exactly the same answer as for system (2.18), where \( T(n) = n + \sum_{1 \leq j < n} T(j) \), if \( n > 1 \). (Try it!) The error comes from the implicit assumption, when computing the difference \( T(2) - T(1) \), that \( T(1) \) satisfies equation (2.20), which it does not. The differences \( T(3) - T(2) \), \( T(4) - T(3) \), etc., do fit our pattern because both terms are defined by the equation (2.20). For these differences, all but the last term in the summation of equation (2.20) are indeed cancelled out. But since \( T(1) \) does not fit the pattern of (2.20), the cancellation will not occur in the expression for \( T(2) - T(1) \).

The easiest fix is to solve directly for \( T(2) = 3 + 1 = 4 \), and write the system, after differencing, as

\[
\begin{align*}
T(2) &= 4; \\
T(n) - T(n - 1) &= 1 + T(n - 1), \quad \text{if } n > 2,
\end{align*}
\]

A more sophisticated fix to the problem is to observe that the original system can be differenced down to \( T(2) - T(1) \); it is just that this last difference must be calculated explicitly (as 4 - 1), since the formula for \( T(n) \) is inapplicable when \( n = 1 \). We get:

\[
\begin{align*}
T(1) &= 1; \\
T(n) - T(n - 1) &= 1 + T(n - 1) + \delta(n, 2), \quad \text{if } n > 1,
\end{align*}
\]

where \( \delta(i, j) \) equals 1 if \( i = j \) and zero otherwise. Despite the irregularity of this additional term, our range transformation works perfectly well to give \( T(n) = 2^n - 1 + 2^{n-2} = 5 \times 2^{n-2} - 1 \), for \( n > 1 \). The extra \( 2^{n-2} \) is due to the \( \delta(n, 2) \) term.

The other examples in this section work precisely because the initial conditions are selected to be consistent with their recurrence formulations. Thus in the Quicksort example, \( T(1) = 1 + \frac{2}{3} \sum_{1 \leq j < 1} T(j) = 1 + 0 = 1 \).

* Solving for the Domain and Range Transformations

It is evident that range and domain transformations can be used to solve, for fixed constants \( a \) and \( b \) (\( b > 1 \)):

\[
\begin{align*}
T(0) &= c_0; \\
T(n) &= a T\left(\frac{n}{b}\right) + c_n, \quad \text{if } n > 0.
\end{align*}
\]

Range transformations can also be used to solve more general recurrence equations such as:

**Example. 2.3.5**

\[
\begin{align*}
T(0) &= c_0; \\
T(n) &= a_n T(n - 1) + c_n, \quad \text{if } n > 0.
\end{align*}
\]
In this case, we must first solve for the function \( g(\cdot) \) that will serve as a summation factor. We have
\[
T(n) - a_n T(n - 1) = c_n, \text{ if } n > 0.
\]

Multiplying by \( g(n) \) gives
\[
g(n)T(n) - g(n)a_n T(n - 1) = g(n)c_n. \tag{2.21}
\]

Our requirement that \( g(n) \) be a summation factor will be satisfied if the first term in (2.21), with \( n - 1 \) substituted for \( n \), equals the second term. Thus we require
\[
g(n - 1)T(n - 1) = g(n)a_n T(n - 1).
\]

Simplifying gives
\[
\frac{g(n)}{g(n - 1)} = \frac{1}{a_n}. \tag{2.22}
\]

We may set \( g(0) = 1 \), and note that this recurrence equation for \( g(\cdot) \) telescopes via multiplication. Thus we can write the solution for \( g(n) \) as a product of \( n \) factors, use \( b \) in a range transformation to change \( T \) into a new function \( S \) that telescopes, solve for \( S \), and then back-solve to get \( T \). In conclusion, there is an easily derived formula for this problem, which is rather complicated. Indeed, it is too complicated to bother using. Accordingly, its derivation is left as an exercise.

This discussion (plus the exercise) shows that there is always a suitable range transformation for any recurrence equation of the form \( T(n) - a_n T(n - 1) + c_n \). We defer from presenting the formula because it is hideously complicated, and a little experience and insight will usually reveal the desired transformation in a simpler form.

Similarly, we can formalize requirements that must be met by a domain transformation, and can often solve the system in cases where the transformation might not be obvious. An example will illustrate the method. We used binary search as an example to introduce domain transformations, and commented that the recurrence equation was "loose." Let's take an ideal \( n \), and attain the recurrence equation for the exact number of comparisons that are needed, in the worst case. Our ideal \( n \) is odd; we compare the search key against the middle term, and then search the left \((n - 1)/2\) items or the right, depending on the outcome. The resulting system for the exact number of comparisons in binary search reads:

**Example. 2.3.6**

\[
\begin{cases}
T(1) = 1; \\
T(n) = T\left(\frac{n-1}{2}\right) + 1, & \text{if } n > 1.
\end{cases} \tag{2.23}
\]

We will assume, for the moment, that the resulting \( \frac{n-1}{2} \) is always odd as well.

The question is, what is the domain transformation? We know what we want: \( d(k) = n \), and \( d(k - 1) = \frac{n-1}{2} \). With this \( d(k) \), we can define \( n = d(k) \), \( S(k) = T(d(k)) \) so that \( S(k - 1) = T(d(k - 1)) = T\left(\frac{n-1}{2}\right) \). Then the system in \( S \) and \( k \) can be easily solved. So we wish to solve for \( d \):

\[
\begin{cases}
d(k - 1) = \frac{n-1}{2}; \\
d(k) = n.
\end{cases} \tag{2.24}
\]
This is a system where $n$ is easily eliminated. Subtracting twice the first equation from the second gives $d(k) - 2d(k - 1) = 1$. We may use an arbitrary initial condition such as $d(1) = 1$. An easy range transformation and the summation of a geometric series gives $d(k) = 2^k - 1$. Thus, we let $n = 2^k - 1$, and put $S(k) = T(2^k - 1)$. The resulting system in $S$ is

\[
\begin{aligned}
S(1) &= 1; \\
S(k) &= S(k - 1) + 1, \quad \text{if } k > 1,
\end{aligned}
\]

which has the solution $S(k) = k$. Back-solving gives $T(2^k - 1) = k$. If $2^k - 1 < n < 2^{k+1} - 1$, then binary search uses in the worst case, $k + 1$ comparisons. (This count will occur when the larger subinterval, at each search step, always winds up to having the item sought).

Most problems need domain transformations that are self-evident. For those rare occasions when the transformation is not obvious, it may be possible to solve for the transformation. Example 2.3.6 is actually quite general. If the recurrence had read, $T(n) = T(g(n)) + h(n)$, then the system for $d$ would be changed to read,

\[
\begin{aligned}
d(k - 1) &= g(n); \\
d(k) &= n.
\end{aligned}
\]

As a system in $k$, no domain transformation will be needed; it is already stated in terms of $d(k)$ and $d(k - 1)$. Whether it can be solved depends on the function $g$. We may substitute $d(k)$ for $n$ to get $d(k - 1) = g(d(k))$. If, for example, $g(n) = an + b$, then all is well, since the recurrence $d(k - 1) = ad(k) + b$ is easily rewritten in terms of $d(k)$, and solved with the help of a summation factor. As for Example 2.3.5, it was already quite general, and the resulting recurrence for the summation factor, equation (2.22) also turned out to be in a simplest form.

The exercises below ask you to derive the general formula for recurrences that need no domain transformation. We have chosen to emphasize the solution methods rather than the formula for three reasons:

1) The methods are easily remembered while the formula is not.

2) A recognizable answer is almost always more easily attained by solving the system directly, than by substituting the coefficients into the very complicated summation of products that is the formula, and trying to simplify the mess.

3) Solution methods can often be extended (via, for example, domain transformations and differencing) to solve new but related problems where a formula may no longer be applicable.

**Exercise** Solve

\[
\begin{aligned}
g(0) &= 1; \\
\frac{g(n)}{g(n-1)} &= \frac{1}{a_n}, \quad \text{if } n > 0.
\end{aligned}
\]

**Exercise** Use the solution to 1 to solve

\[
\begin{aligned}
T(0) &= c_0; \\
T(n) &= a_n T(n - 1) + c_n, \quad \text{if } n > 0.
\end{aligned}
\]
2.3.4 Examples

Example: Divide-and-Conquer

An important class of recurrence equations results from algorithms based on the ‘divide-and-conquer’ paradigm; such algorithms consist of three steps:

1. **DIVIDE:** Using at most \( g(n) \) time, break the problem of size \( n \) into \( a \) smaller subproblems each of size \( \frac{n}{b} \), and each of the same type as the original problem.

2. **RECUR:** Recursively solve each of the \( a \) subproblems using \( a \cdot T\left(\frac{n}{b}\right) \) time.

3. **MARRY:** Combine the solutions of the \( a \) subproblems to get a solution to the original problem, using no more than \( h(n) \) time.

The recurrence equation is of the form:

\[
T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \quad \text{where } f(n) = g(n) + h(n).
\] (2.26)

For Mergesort, the recurrence was

\[
\begin{align*}
T(1) &= 1; \\
T(n) &= 2T\left(\frac{n}{2}\right) + n, \quad \text{if } n > 1
\end{align*}
\]

and we saw that the domain transformation \( k = \log_2 n \) and substitution \( S(k) = T(2^k) \), \( S(0) = 1 \) gives

\[
\begin{align*}
S(0) &= 1; \\
S(k) &= 2S(k-1) + 2^k, \quad \text{if } k > 0.
\end{align*}
\]

Dividing by \( 2^k \) and exploiting the range transformation \( R(k) = \frac{S(k)}{2^k} \) gives

\[
\begin{align*}
R(0) &= 1; \\
R(k) &= R(k-1) + 1, \quad \text{if } k > 0.
\end{align*}
\]

Solving for \( R \) and back-solving for \( S \) and then \( T \) gives \( T(n) = n \log_2 n + n \), if \( n \) is a power of 2.

Quicksort gave a more complicated divide-and-conquer scheme that yielded the recurrence:

\[
\begin{align*}
T(1) &= 1; \\
T(n) &= n + \frac{2}{n} \sum_{1 \leq j < n} T(j), \quad \text{if } n > 1.
\end{align*}
\]

The analysis of some parallel algorithms has a different divide-and-conquer structure. Here problems of size \( n \) are sometimes partitioned into \( \sqrt{n} \) subproblems of size \( \sqrt{n} \) each. The recurrence for the total number of operations, up to a constant factor, might read:

\[
\begin{align*}
W(n) &= 1; \quad \text{if } n \leq 2 \\
W(n) &= \sqrt{n} W(\sqrt{n}) + n, \quad \text{if } n > 2.
\end{align*}
\]
We specified the initial condition $W(2) = 1$ and, in fact $W(n) = 1$ for any $n \leq 2$ so that the recursion would formally terminate as repeated square roots of $n$ are taken. The point is that by applying the process $n \leftarrow \sqrt{n}$ repeatedly, the resulting value will eventually become at most 2, for any positive $n$.

Notice that the exponent of the value $n^1$ is halved at each iteration. If we try the domain transformation $k = \log_2 n$, $V(k) = W(2^k)$, $V(k) = 1$ if $k \leq 1$, we get the recurrence:

$$
\begin{align*}
V(k) &= 1; & \text{if } k \leq 1 \\
V(k) &= 2^{k/2}V(\frac{k}{2}) + 2^k, & \text{if } k > 1.
\end{align*}
$$

This form begs for the domain transformation $j = \log_2 k$, $U(j) = V(2^j)$, $U(j) = 1$, for $j \leq 0$. Performing the substitutions gives:

$$
\begin{align*}
U(j) &= 1; & \text{if } j \leq 0 \\
U(j) &= 2^{2^{j-1}}U(j - 1) + 2^{2^j}, & \text{if } j > 0.
\end{align*}
$$

This equation is not as bad as it looks. The summation factor turns out to be $\frac{1}{2^{2^j}}$. Multiplying by it gives

$$
\begin{align*}
U(0) &= \frac{1}{2} \\
U(j) &= \frac{U(j - 1)}{2^{2^{j-1}}} + 1, & \text{if } j > 0,
\end{align*}
$$

where we used the fact that $\frac{2^{2^{j-1}}}{2^{2^j}} = \frac{1}{2^{2^j}}$. Now the transformation $T(j) = \frac{U(j)}{2^{2^j}}$, $T(j) = \frac{1}{2}$ for $j \leq 0$ gives:

$$
\begin{align*}
T(0) &= \frac{1}{2} \\
T(j) &= T(j - 1) + 1, & \text{if } j > 0.
\end{align*}
$$

So $T(j) = j + \frac{1}{2}$, and back-soloving for $W$ gives $W(n) = n \log_2 \log_2 n + \frac{n}{2}$, when $n = 2^{2^j}$.

We have solved the recurrence for a very sparse set of integers. How such a solution can be extended to all positive integers is the subject of the (more technical) Section 2.4.2.

### 2.4 Further properties of recurrence equations

#### 2.4.1 Verifying that constants do not matter, and understanding recurrences for upper and lower bounds

**Exercises.** For the exercises below, let $a_z$ and $b_z$ be a sequence of positive coefficients. Suppose that $f(z) \leq z - 1$ for all $z$, so that the iteration

$$
z \leftarrow n; \quad \text{repeat } n \text{ times : } z \leftarrow f(z)
$$

is guaranteed to leave $z$ with a value that is at most zero. This requirement will be sufficient to ensure that the recurrences given below may be solved by recursively unwinding (iterating) the
equations. Suppose further, that for some fixed constant \( \kappa > 1 \), \( b_z \leq c_z \leq \kappa \times b_z \); thus \( b_z \) is within a fixed constant factor of \( c_z \), for all \( z \geq 1 \). Consider the following recurrences, where \( f, a, \) and \( b \) are as above:

\[
\begin{align*}
T(x) &= b_1; & \text{if } x \leq 1, \\
T(z) &= a_z T(f(z)) + b_z; & \text{if } z > 1.
\end{align*}
\]

\[
\begin{align*}
U(x) &= c_1; & \text{if } x \leq 1, \\
U(z) &= a_z U(f(z)) + c_z; & \text{if } z > 1.
\end{align*}
\]

\[
\begin{align*}
V(x) &= \kappa \times b_1; & \text{if } x \leq 1, \\
V(z) &= a_z V(f(z)) + \kappa \times b_z; & \text{if } z > 1.
\end{align*}
\]

**Exercise** Use reasonably standard induction to show that \( T \leq U \leq V \). Are the positivity requirements for the \( a_z \) and \( b_z \) important? (Hint: the most common form of induction says, "suppose \textit{whatever}(n) is true for \( n = k - 1 \). We will now use this fact to prove that \textit{whatever}(k) also holds." This rigid approach will not work because \( T(z), U(z), \) and \( V(z) \) are not expressed in terms of \( T(z-1), U(z-1), \) and \( V(z-1), \) but rather \( T(f(z)), U(f(z)), \) and \( V(f(z)). \)

A professional proof might begin as follows: suppose the inequalities hold for \( z \leq t \). We use induction to establish the bounds for \( s \) in \([t, t + 1]\). So let \( s \) be chosen from the range \([t, t + 1]\). This approach resolves the fact that the recurrence is defined over the reals, and increases the region where the solution is known by segments rather than by points. Another (adequate but less elegant) possibility is to define \( z_n \) to be the \( n^{th} \) iterate \( f(f(\cdots f(z) \cdots)) \), and consider the recurrence on the sequence \( z_n, z_{n-1}, \ldots, z_1, z \). For half credit, solve the above with \( f(z) = z - 1 \).

**Exercise** Are the positivity requirements for the \( a_z \) and \( b_z \) important?

**Exercise** Show, by a simple range transformation, that \( V \equiv \kappa \times T \), so that \( T \leq U \leq \kappa \times T \).

Conclude that changing the inhomogeneous terms \( b_z \) by at most a constant factor can only change the solution by the same constant factor (or by a factor that is even closer to 1).

**Exercise** Does this proof method generalize to establish the analogous bounds for the more complicated equations \( T(n) = \sum_{i=1}^{n} a_i T(f(i)) + b(n) \) or, say, \( T(n) = \sum_{i=1}^{n} a(i, n) T(f(i)) + b(n) \), for nonnegative \( a \)'s and \( b \)'s?

**Exercise** Does this proof method generalize to establish the analogous bounds for the more complicated equation \( T(n) = a_n T(f(n))^2 + b(n) \), with, say \( T(1) = 1 \)?

**Exercise** Consider the recurrences:

\[
\begin{align*}
T(x) &= b_1; & \text{if } x \leq 1, \\
T(n) &= a_n T(f(n)) + b_n; & \text{if } n > 1.
\end{align*}
\]

\[
\begin{align*}
U(x) &\leq b_1; & \text{if } x \leq 1, \\
U(n) &\leq a_n U(f(n)) + b_n; & \text{if } n > 1.
\end{align*}
\]

\[
\begin{align*}
L(x) &\geq b_1; & \text{if } x \leq 1, \\
L(n) &\geq a_n L(f(n)) + b_n; & \text{if } n > 1.
\end{align*}
\]

Show by induction that \( T(n) \leq L(n) \), for positive \( a \)'s, and that \( U(n) \leq T(n) \).

Notice that the requirement for a lower bound \( L \) is just the opposite of what we might expect: the recurrence reads, \( L(x) \geq \text{something} \). There is a perfectly good reason for the direction of
the inequality. Suppose we have an algorithm $A$, and wish to show that the running time for $A$
onumber
on problems of size $n$ must be at least as big as some function $L(n)$ (in, say, the worst case).

Then we want to find a good function $L$. Probably $L(n) = 0$ is a true, but a rather poor lower
bound. We want $L$ to be at least as big as something more informative than zero, which is to say
that $L(x) \geq \text{something}$. For recursive algorithms, the recursive processing can often be used to
quantify the something in terms of the recursive processing of the algorithm: the time to Mergesort
$n$ items is at least twice an underestimate of the running time for Mergesorting $n/2$ items plus an
underestimate of the time to merge the two sorted sets: $L(n) \geq 2L(\frac{n}{2}) + n - 1$. We may conclude
that by changing the inequality to an equality, in this case (which lowers the value of $L(n)$ to be
equal to twice (a lowered estimate for) $L(\frac{n}{2})$ plus $n - 1$) gives the solution $T$ that is indeed a true
lower bound; $L$ is at least as large as $T$.

Similarly, we sometimes formulate upper bounds via formulations that mathematically prevent
our overestimates from being too large, whence $U(n) < \text{something}$. Conclude that by changing the
inequality to an equality, in this case, the solution $T$ will be an upper bound, since $U$ is no larger
than $T$.

2.4.2 *Simplifying the recurrence and solving it everywhere

The purpose of this section is to establish general principles that prove once and for all, that
irritating details of carefully derived recurrence equations are irrelevant. In order to achieve this
goal, we are obliged to face a little more mathematics than ought to be necessary. Luckily, we will
do this just once, and do not have to be in touch with the methods used to prove the point. Even
the specific theorems are not so important, as long as we know that they exist.

On to the details.

When we solved the equation for Mergesort,

\begin{align*}
T(1) &= 1; \\
T(n) &= 2T(\frac{n}{2}) + n, \quad \text{if } n > 1,
\end{align*}

(2.27)

it was convenient to assume that $n$ was a power of 2. The solution was then seen to be $T(n) =
n\log_2 n$, for $n = 2^k$. If $n$ is not a power of 2, then there is a $k$ where $2^{k-1} < n < 2^k$; take
$k = \lceil \log_2 n \rceil$. Since Mergesort, like most algorithms, runs longer for larger data sets, we may use
this $k$ to conclude that $T(2^{k-1}) < T(n) < T(2^k)$. So $(k - 1)2^{k-1} < T(n) < k2^k$, which captures the
value of $T(n)$, up to a small constant factor.

The point of this exercise is to observe that even though we solved the recurrence equation for
the sparse set of values $n = 2^k$, $k = 1, 2, \ldots$, the function $T(n)$ will only double, approximately, if
we overestimate $T(n)$ by evaluating $T$ for the first value at least as large as $n$ in our special sparse
set.

Moreover, this method even applies to the seemingly more complex equation:

\begin{align*}
U(1) &= 1; \\
U(n) &= U(\lfloor \frac{n}{2} \rfloor) + U(\lceil \frac{n}{2} \rceil) + n, \quad \text{if } n > 1,
\end{align*}

@Sacki, Cole 1994

rec/sparse
since it is the same as equation 2.27, when \( n \) is a power of 2. The technical reason that we still have the inequality \( U(2^{k-1}) \leq U(n) \leq U(2^k) \), for \( 2^{k-1} \leq n \leq 2^k \), is that the solution \( U(n) \) is nondecreasing.

**Exercise** Consider the recurrence
\[
\begin{align*}
T(x) &= b(1); & \text{if } x \leq 1, \\
T(n) &= a(n)T(f(n)) + b(n), & \text{if } n > 1.
\end{align*}
\]

Suppose that \( a(n) \), \( b(n) \), and \( f(n) \) are nondecreasing, and assume that, say \( f(n) \leq n - 1 \), which is sufficient to guarantee that a solution exists. Show that \( T \) is nondecreasing.

The method of solving a recurrence on a sparse set of values, and using the solution to bound the solution elsewhere will work if the following are true.

1) The solution is nondecreasing, which is typical of the recurrence in computer science.

2) The solution function \( T(n) \) grows only polynomially in \( n \).

3) The sparse set where we solve for \( T \) grows no faster than exponentially.

Indeed, since the consecutive values in an exponentially growing sequence differ by only a constant factor, the first such value that is at least as large as \( n \) can only be a constant factor larger than \( n \), and \( p(cn) = \Theta(p(n)) \), when \( p \) is a polynomial of fixed degree.

Unfortunately, there are, upon rare occasion, useful algorithms that yield recurrence equations where these growth requirements, as general as they are, do not hold. A natural example is provided by the recurrence:
\[
\begin{align*}
T(n) &= 1, & \text{if } n \leq 2; \\
T(n) &= \sqrt{n}T(\sqrt{n}) + n, & \text{if } n > 2.
\end{align*}
\]

We saw in Section 2.3.4 that the solution (which results from the domain transformation \( n = 2^m \)) is
\[
T(n) = n \log_2 \log_2 n + \frac{n}{2},
\]
when \( n = 2^m \). Thus there is a meaningful solution for \( n = 2, 4, 16, 256, 62836, 2^{32}, 2^{64}, \ldots \). Moreover, the solution grows just a little faster than linear, for these values of \( n \), at least. But how much should \( T (257) \) be? The next larger value where we have solved for \( T \) is 62836, but using \( T(62836) \) as an upper bound for \( T(257) \) would seem unsatisfactory and would give much worse performance, in general, than apparent (but as yet unproven) bound of \( T(n) = \Theta(n \log \log n) \), for all \( n \).

Three questions we must face are:

- How should this recurrence equation be interpreted when \( n \) is not a perfect square?
- How can \( T(n) \) be bounded for values general values of \( n \)?
- A related question is: how to attain tight bounds for recurrences that may not have any solution defined on the integers as the recurrence unwinds?

For example, suppose we had the recurrence
\[
\begin{align*}
T(1) &= 1; \\
T(n) &= \sqrt{n}T(\sqrt{n} - 1) + \sqrt{n}T(\sqrt{n} + 1) + n, & \text{if } n > 1.
\end{align*}
\]
This recurrence has no sparse set where the indices remain integers as the recurrence unwinds toward 1. We need to ask how to interpret it, why would such an equation arise, and how can a satisfactory bound be attained for its solution.

Divide-and-Conquer methods often partition problems into subproblems that cannot be all the same size, for some data sets. Consider Mergesort, for example. It can be applied to sets of odd size; in this case, a set of size $n$ is partitioned into subsets of size $\left\lfloor \frac{n}{2} \right\rfloor$ and $\left\lceil \frac{n}{2} \right\rceil$. Its general recurrence equation reads,

$$
\begin{align*}
T(1) &= 1; \\
T(n) &= T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + n, \quad \text{if } n > 1.
\end{align*}
$$

While the powers of two give a sparse of values where, in this case, the floors and ceilings disappear, and the three requirements for satisfactorily bounding $T$ do in fact hold, we now show that the solution can be bounded in many cases where there is no such suitable set.

As a general rule, we may interpret fractional data sizes by rounding up to the next larger integer. The algorithmic reason for rounding upward is that divide-and-conquer methods, for data or problem sizes that fail to divide exactly, will produce some subproblems having a size equal to this larger value. For equation (2.28), a divide-and-conquer partitioning might also yield $\left\lceil \sqrt{n} \right\rceil$ subproblems, when $n$ is not a perfect square. Of course, such rounding cannot give a smaller answer, for our recurrences where larger problems require longer amounts of time, and all subproblems require positive amounts of time.

Recurrences with ceilings can be formulated with case statements, that give slightly different formulations, depending on value (or divisibility) of $n$. These more complicated systems of recurrence equations are more difficult to solve, although they sometimes can be solved by elementary means. As a technical matter, it is easier to replace $\lceil expression \rceil$ by $expression + 1$, since the latter has no cases. This means that the recurrence equation would be solved on the reals, rather than the integers.

For example, equation (2.29) would read: $T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + n$, if $n > 2$. Increasing the argument $\left\lfloor \frac{n}{2} \right\rfloor$ to $\frac{n}{2} + 1$ will produce a larger value for $T$, since $T(n)$ is increasing as a function of $n$. The system, with initial condition included might read:

$$
\begin{align*}
T(n) &= 1, \quad \text{if } n \leq 2; \\
T(n) &= T(\frac{n}{2}) + T(\frac{n}{2} + 1) + n, \quad \text{if } n > 2.
\end{align*}
$$

The need to specify $T(2)$ as an initial condition comes from the fact that the argument $\frac{n}{2} + 1$ takes 2 into two. Because the recurrence is now on the reals, we might specify $T$ for reals less than or equal to 2. In general, an overestimation procedure that replaces, for example, $\lceil expression \rceil$ by $expression + 1$, typically requires a slightly different initial condition for this very reason. The recurrence stops being useful for values of $n$ that are not expressed in terms of the solution for smaller values of $n$.

While simpler than recurrences with ceilings or floors (that cannot be eliminated), the rounded recurrence equations still suffers from unnecessary complexity caused by the additional 1's. A first step for equation (2.30) might be to replace $T(n) = T(\frac{n}{2}) + T(\frac{n}{2} + 1) + n$ by the overestimate $T(n) = 2T(\frac{n}{2} + 1) + n$. 

©Siewj, Cole 1994  src/sparse
Yet, these new ones still complicate the equation, while we would expect that their presence ought to increasing the answer by a rather modest amount, in most cases.

There are several ways to solve these problems, and to eliminate these difficulties. We can prove that for many circumstances, the extra 1's cannot change the resulting answer by more than a constant factor, so that the easiest remedy, in these cases, is to drop them and to solve the simplified problem without concern about rounding and whether or not the arguments are integers.

The material is presently withheld due to its technical nature.

2.5 Appendix to Chapter 2

- **Geometric series**: \( c + cr + cr^2 + \cdots + cr^n = \frac{c - cr^{n+1}}{1-r} \). This true for ratios \( r < 1 \) and \( r > 1 \). Derivation: put \( s = c + \cdots + cr^n \), and use clever differencing: consider \( s - rs \). Note that the sum = first term - ratio \times \text{last} divided by 1 - ratio. Also, geometric series (for fixed \(|r| \neq 1\)) can be approximated by the largest term, which is either the first or last term. For \(|r| < 1\), the infinite series sums to \( \frac{c}{1-r} \). Consequences: \( 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2 \). Estimating geometric sums on the cheap: \( c + cr + cr^2 + \cdots + cr^n = \Theta(\text{max (first term, last term)}) \).

- **Modified Geometric series**: \( 1cr + 2cr^2 + 3cr^3 + \cdots = \frac{cr}{(1-r)^2} \), for \(|r| < 1\). Derivation: take the infinite (unmodified) geometric series above, differentiate with respect to \( r \), and multiply by \( r \). There is a corresponding form for a finite series.
Consequences: set \( r = \frac{1}{2} \) and \( c = 1 \). Then \( \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots = \frac{1\times\frac{1}{2}}{(1-1/2)^2} = 2 \).
So \( \frac{n}{2} + \frac{2n}{4} + \frac{3n}{8} + \cdots + \frac{(-1+\lg n)n}{n/2} + \frac{(\lg n)n}{n} < 2n \), whereas \( \frac{n}{n} + \frac{2n}{n} + \cdots + \frac{(-1+\lg n)n}{4} + \frac{(\lg n)n}{2} = \Theta(\frac{(\lg n)n}{2}) \).

- **Harmonic series**: \( 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \log n + \Theta(1) \). The sum of the first \( n \) terms differs from \( \log n \) by at most a fixed constant. Derivation: approximate by an integral. Note also that \( \log n \) and \( \log(n+1) \), for example, are very close: their difference is \( o(1) \) (or more precisely, \( \Theta(1/n) \)) so there are many good estimates for such a sum.

- **Other sums**: \( 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \). Derivation: \( 1 + (n) + 2 + (n-1) + \cdots + n + 1 = n(n+1) \). This derivation of adding the series in the forward and the reverse order works for any arithmetic progression, and there is a natural semantic interpretation: the sum of \( n \) terms is \( n \times \) (average value), and the average value is the average of the first and last terms, as well as the average of any comparable pairing.
\( 1^2 + 2^2 + 3^2 + 4^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \). Derivation: induction.
Generic Derivation: telescoping to cancel terms shows that \( (1^k - 0^k) + (2^k - 1^k) + \cdots + (n^k - (n-1)^k) = n^k \). This summation illustrates what we already know: exact differences such as \( a_n = n^k - (n-1)^k \) are easily summed. Suppose we could express \( n^k \) as a suitable sum of the polynomials \( (n^k + 1 - (n-1)^k) \), \( n^k - (n-1)^k \), \( n^k - (n-1)^k \), \( n^k - 1 - (n-1)^k \), \( \ldots \), \( n^k - 1 - (n-1)^k \): if, for example, we knew that \( n^4 = \frac{n^5 - (n-1)^5}{5} + \frac{n^4 - (n-1)^4}{2} + \frac{n^3 - (n-1)^3}{3} - \frac{n^2 - (n-1)^2}{30} \), then we could evaluate \( \sum_{i=1}^n i^4 \) by summing the expression \( \frac{n^5 - (n-1)^5}{5} + \frac{n^4 - (n-1)^4}{2} + \frac{n^3 - (n-1)^3}{3} - \frac{n^2 - (n-1)^2}{30} \), which is easy because each term is an exact difference, so almost all terms will telescope. Moreover, if we can find these representations for \( n^k \), for each \( k \), then we can represent any polynomial.
as a sum of these exact differences. There are many ways to find these representations, but perhaps the easiest is as follows. We may write identity 1 = n - (n - 1), which represents 1 as an exact difference. Similarly, of course is the difference c = c(n - (n - 1)). Now for our first trick. Integrating both sides of the equation 1 = n - (n - 1) from 0 to n gives \( n = \int_0^n ndn - \int_0^n (n-1)dn = n^2/2 - (n-1)^2/2 + 1/2 = n^2 - (n-1)^2/2 + n(n-1)/2 \), where the last term is a rewrite of the constant \( 1 = (n-1)/2 \). Integrating both sides of the equality \( n = \frac{n^2-(n-1)^2}{2} + \frac{n(n-1)}{2} \) gives \( n^2 = \frac{n^3-(n-1)^3}{3} + \frac{n^2-(n-1)^2}{2} + \frac{n(n-1)}{12} \), where we again rewrote the constant to be an exact difference. Multiplying by 2 gives the difference expression for \( n^2 \). Repeating this process twice more gives \( n^4 = \frac{n^5-(n-1)^5}{5} + \frac{n^4-(n-1)^4}{2} + \frac{n^3-(n-1)^3}{3} - \frac{n(n-1)}{30} \), whence telescoping shows that \( 1^4 + 2^4 + \cdots + n^4 = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n(n-1)}{30} \).

Simpler estimate: \( 1^k + 2^k + 3^k + 4^k + \cdots + n^k = \frac{n^{k+1}}{k+1} + O(n^k) \). Derivation: approximate by an integral as described next, or use above reasoning.

- **General sums**: Integral approximations work quite naturally for approximating the sums that result from estimating operation counts, since the terms are not only positive, but are also monotone. Suppose that \( f(x) \) is nondecreasing in \( x \). Then \( \int_0^n f(x)dx \leq \int_0^n f([x]+1)dx = \sum_{i=1}^n f(i) \leq \int_0^{n+1} f(x)dx \). If \( f(x) \) is nonincreasing in \( x \), then \( \int_0^{n+1} f(x)dx \leq \sum_{i=1}^n f(i) \leq \int_0^n f(x)dx \). Proof: draw a picture. The summations can be represented as step functions of boxes lying above or below the curve \( y = f(x) \). For nondecreasing \( f \), \( f([x]) \leq f(x) \leq f([x]+1) \), for \( x \) in \([0,n]\). Other sums can also be approximated by integrals; a very sharp estimation method is provided by the Euler Summation Formula in Knuth, Vol.I.

- **Convergent sums**: Due to range transformations, it is common to get results of the form \( S(n) = \sum_{1 \leq j \leq n} a_j \), where the infinite series \( a_1 + a_2 + \cdots \) converges (i.e. is finite). For such cases, you may conclude \( S(n) = \Theta(1) \). The asymptotic behavior for \( T \) will come from factors in the back-substitution. These sums also appear as lower order terms: \( T(n) = \sum \text{bigstuff}, + \sum_{1 \leq j \leq n} a_j \), whence \( T(n) = \sum \text{bigstuff}, + O(1) \). Remember that the infinite series is convergent if, for example, \( a_j = O(\frac{1}{j^\alpha}) \), for fixed \( \alpha > 1 \) (this is proved by an integral approximation).

- **Stirling's Formula**: Stirling's formula is a very useful (asymptotic) approximation for \( n! \). It states that

\[
n! \approx \sqrt{2\pi}nn^n e^{-n}
\]

Let \( nfac = \sqrt{2\pi}nn^n e^{-n} \) (so that \( n! \approx nfac \)). A more explicit approximation is: for \( n \geq 1 \),

\[
nfac \leq n! \leq nfac \times (1 + \frac{1}{4n})
\]

Moreover, \( n! = nfac \times (1 + \frac{1}{12n} + O(\frac{1}{n^3})) \). Even more asymptotically precise formulations can be given, but in practice, the \( nfac \leq n! \leq nfac \times (1 + \frac{1}{4n}) \) formulation seems to be sufficient. It is worth noting that while these approximations are excellent in the multiplicative sense, we see that \( n! - nfac = \Theta((n-1)!) \); in fact, the additive error is huge no matter how many error terms are included.

---

5Notice that the asymptotic growth operators make sense for functions that go to zero as well as \( \infty \). If \( f(n) = O(g(n)) \) where \( g(n) \to 0 \), then \( f(n) \) goes to zero at least as quickly as \( g(n) \), up to a constant factor.
A common application of these approximations gives the estimate
\[
\binom{n}{n/2} = \frac{n!}{(n/2)!((n/2))!} = \frac{2^{n+1}}{\sqrt{2\pi n}} (1 + O\left(\frac{1}{n}\right)). \quad \text{More loosely,} \quad \binom{n}{n/2} \approx 2^n.
\]

- **Additional remarks:** The derivation of the formula for a modified geometric series illustrates Taylor series applications and tricks such as differentiation. Another such example comes from the Binomial Theorem:

\[
\sum_{0 \leq j \leq n} \binom{n}{j} x^j y^{n-j} = (x + y)^n.
\]

Differentiating with respect to \(x\) and multiplying yields \(x \frac{d}{dx} \sum_{0 \leq j < n} \binom{n}{j} x^j y^{n-j} = \sum_{0 \leq j < n} \binom{n}{j} j x^j y^{n-j}\). Setting \(x + y = 1\) and \(x = \frac{1}{m}\) shows \(\sum_{0 \leq j < n} \binom{n}{j} j (\frac{1}{m})^j (1 - \frac{1}{m})^{n-j} = \frac{n}{m}\). We will see that these steps derive the expected number of items encountered during an insertion (or unsuccessful search) in a hash table (with open addressing) that has \(m\) buckets and \(n\) items.

- **The Mean Value Theorem for differential calculus:** Let \(f\) be continuous on the closed interval \(a \leq x \leq b\), and suppose that \(f\) is differentiable everywhere in the open interval \(a < x < b\). Then \(f(b) - f(a) = (b - a) f'(x_0)\), for some \(x_0\) with \(a < x_0 < b\).

- **References:**
  
  
  
  
  
  