Order And Algorithm Analysis: A Concise Introduction

©2003 Steven Louis Davis
Copyright©2003

Contents

1 Mathematical Preliminaries **1**
1.1 Logs . . . 2
1.2 Derivatives . . . 4
1.3 Limits . . . 6
1.3.1 0-limits of nonnegative non-increasing sequences . . . 6
1.3.2 Limits and Logs . . . 6
1.3.3 Limits and Derivatives . . . 7
1.4 Combinatorics . . . 9
1.5 Graph Theory . . . 11
1.5.1 Trees . . . 12
1.5.2 Representation in Code . . . 13
1.6 Probability . . . 15
1.6.1 Mathematical Expectation . . . 15
1.6.2 Change of Variable . . . 15
1.7 Sums . . . 16
1.8 Induction . . . 18

I Order **21**

2 Order Comparison **23**
2.1 Equivalent Order . . . 30
2.2 Inferior Order . . . 35
2.3 Non-Strict Order . . . 38
2.4 Traditional Notation . . . 40
2.5 Bounding Techniques . . . 42
2.6 Delay and Invariance . . . 44
2.7 Order Hierarchy . . . 45
2.7.1 Polylogs and Powers . . . 46
2.7.2 Iterated Log Subhierarchy . . . 49
2.7.3 Polynomials and Exponentials . . . 51
2.7.4 Exotic Order Comparisons . . . 52
2.7.5 Ackerman's Function . . . 54
2.7.6 Collected Comparisons . . . 56

2.8 Existence of the Laplace Transform 57

II Algorithm Analysis **63**

3 Time Complexity Analysis **65**
3.1 Average Case Time Complexity Analysis 65

4 Greedy Selection **69**
4.1 Single Source Shortest Paths 69
4.1.1 Dijkstra's Algorithm 70
4.2 Minimal Spanning Trees 71
4.2.1 Prim's Algorithm 71
4.2.2 Kruskals's Algorithm 72

5 Dynamic Programming **75**
5.1 Recursively Defined Solutions 75
5.2 Calculating Combinations 75
5.3 Shortest Paths Revisited 80
5.4 Traveling Salesman Problem 81
5.5 Order of Sequenced Matrix Multiplication 83

6 Divide and Conquer **95**
6.1 Constructive Induction 98
6.2 Fast Exponentiation 107
6.3 Sorting Arrays 110
6.3.1 Mergesort 110
6.3.2 Selectionsort 112
6.3.3 Quicksort 112
6.3.4 Heapsort 117
6.4 Simplifying Recurrences 120
6.5 Fast Fourier Transform 122
6.5.1 The Discrete Fourier Transform 122
6.5.2 The Fast DFT Algorithm 124
6.6 Exploiting Associations with Recursion 126
6.6.1 Large Integer Arithmetic 127
6.6.2 Strassens Matrix Multiplication 132

III Appendices **139**

A Mathematical Reference **141**
A.1 Discrete Fourier Transform 142
A.2 Binomial Coefficients 144
A.3 More Sums 151
A.3.1 Polygeometric Sums 151
A.4 Limits of Sums 155

B General Solution of Elementary Recurrences **157**
B.1 Terminal Compositions 157
B.2 General Solution Of Elementary Recurrences 160
B.3 Further Applications 168

C More Fibonacci Algorithms **171**

Contents

Chapter 1

Mathematical Preliminaries

1.1 Logs

Logarithms are of fundamental importance in the expression of functional orders as they arise quite naturally in the study of computer algorithms. Though there are several ways to define logarithms, nothing can be simpler than noting that the log function is just the inverse of the corresponding (with respect to base) exponential function. As a simple aid to remembering this definition it us useful to keep in mind the defining principle that logs are exponents. Consider the logarithmic equation:

$$y = \log_b(x).$$

To every logarithmic equation corresponds an exponential equation in the same base, and by remembering the defining principle above we obtain

$$b^y = x$$

since we already know the base b, and the exponent is the log, which in the logarithmic equation is y. The x takes its place on the right hand side being the only one of the three quantities not already accounted for. For example since $2^5 = 32$, we have $\log_2(32) = 5$, the log being the exponent 5. Armed with this basic understanding we can now derive the basic properties of logs by using the corresponding properties of exponents. First, simplest, and perhaps most important, we have

$$x = b^{\log_b(x)}$$

by simply substituting for y in the exponential equation. It should also be clear that $\log_b(1) = 0$ since the only possible value of y giving $b^y = 1$ is $y = 0$. We now proceed to obtain the following:

$\log_b(xy) = \log_b(x) + \log_b(y)$**:**

$$\begin{aligned} \log_b(xy) &= \log_b(b^{\log_b(x)} b^{\log_b(y)}) \\ &= \log_b(b^{\log_b(x)+\log_b(y)}) \\ &= \log_b(x) + \log_b(y) \end{aligned}$$

$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$**:**

$$\begin{aligned} \log_b\left(\frac{x}{y}\right) &= \log_b\left(\frac{b^{\log_b(x)}}{b^{\log_b(y)}}\right) \\ &= \log_b(b^{\log_b(x)-\log_b(y)}) \\ &= \log_b(x) - \log_b(y) \end{aligned}$$

$\log_b\left(\frac{1}{y}\right) = -\log_b(y)$**:**

$$\begin{aligned} \log_b\left(\frac{1}{y}\right) &= \log_b(1) - \log_b(y) \\ &= 0 - \log_b(y) \end{aligned}$$

$\log_b(x^k) = k\log_b(x)$**:**

$$\begin{aligned} \log_b(x^k) &= \log_b(xx^{k-1}) \\ &= \log_b(x) + \log_b(x^{k-1}) \\ &= \log_b(x) + \log_b(x) + \log_b(x^{k-2}) \\ &\vdots \\ &= \underbrace{\log_b(x) + \log_b(x) + \cdots + \log_b(x)}_{k \text{ factors}^1} \\ &= k\log_b(x) \end{aligned}$$

$a^{\log_b(x)} = x^{\log_b(a)}$**:**

$$\begin{aligned} a^{\log_b(x)} &= b^{\log_b(a^{\log_b(x)})} \\ &= b^{\log_b(x)\log_b(a)} \\ &= b^{\log_b(x^{\log_b(a)})} \\ &= x^{\log_b(a)} \end{aligned}$$

$\log_b(x) = \frac{\log_\beta(x)}{\log_\beta(b)}$**:**

$$\begin{aligned} \log_\beta(x) &= \log_\beta(b^{\log_b(x)}) \\ &= \log_b(x)\log_\beta(b) \end{aligned}$$

Lastly, we call attention to the convention of denoting logarithms taken in the natural base e $(= 2.718\ldots)$ with the special name ln, that is,

$$\ln(a) = \log_e(a).$$

Another convention is to denote the base 2 log by lg, that is,

$$\lg(a) = \log_2(a).$$

[1] Though we have assumed k is an integer, the property holds also for real numbers.

Table 1.1: Common Derivatives

$f(n)$	$f'(n)$	example(s)
n^p	pn^{p-1}	$\frac{d}{dn}[n^3] = 3n^2$
a^n	$a^n \ln(a)$	$\frac{d}{dn}[2^n] = 2^n \ln(2)$, $\frac{d}{dn}[e^n] = e^n$
$\log_b(n)$	$\frac{1}{n \ln(b)}$	$\frac{d}{dn}[\log_2(n)] = \frac{1}{n \ln(2)}$, $\frac{d}{dn}[\ln(n)] = \frac{1}{n}$

1.2 Derivatives

Being exclusively interested in non-decreasing monotonic (non oscillating) functions dramatically reduces the types of functions encountered. For example we have no need of trigonometric functions. The type of behavior exhibited by algorithm time complexity functions is quite restrictive and appears to be confined to logarithmic, polynomial, exponential, and factorial growth as well as mixtures and generalizations of these. As we encounter very few different basic functions, we can easily summarize the most commonly encountered derivatives as collected in Table 1.1.

We most often encounter these basic functions mixed together in products and compositions. Table 1.2 provides the formulae to handle these combinations. The student familiar with elementary calculus will recognize these as the Product and Chain Rules respectively.

Table 1.2: Product and Composition Formulae

$f(n)$	$f'(n)$	examples
$u(n)v(n)$	$u'(n)v(n) + u(n)v'(n)$	$\frac{d}{dn}[n \log_2(n)] = \log_2(n) + \frac{1}{\ln(2)}$, $\frac{d}{dn}[n^2 2^n] = 2n2^n + n^2 2^n \ln(2) = n2^n(2 + n\ln(2))$
$u(v(n))$	$u'(v(n))v'(n)$	$\frac{d}{dn}[2^{n^3}] = 2^{n^3}\ln(2)(3n^2) = 3\ln(2)n^2 2^{n^3}$, $\frac{d}{dn}[(2n+1)^2] = 2(2n+1)(2) = 4(2n+1)$, $\frac{d}{dn}[\sqrt{3n^4+n+2}] = \frac{1}{2}(3n^4+n+2)^{-\frac{1}{2}}(12n^3+1) = \frac{1}{2\sqrt{3n^4+n+2}}$

1.3 Limits

1.3.1 0-limits of nonnegative non-increasing sequences

By far the most common type of limit encountered is of the form

$$\lim_{n\to\infty} s_n,$$

where $(s_n)_{n=1}^{\infty}$ is a non-increasing sequence of nonnegative real numbers. Of these we are most often interested in those with limiting value of zero. For this case we use the characterization that for any positive number $c > 0$, $f(n) < c$ for large enough n.

Example 1.1. $\lim_{n\to\infty} \frac{1}{n} = 0$

For $c > 0$, let $n > \lceil \frac{1}{c} \rceil$. Then $n > \frac{1}{c}$, so that $\frac{1}{n} < c$.

Example 1.2. $\lim_{n\to\infty} \frac{1}{n^2} = 0$

For $c > 0$, let $n > \lceil \frac{1}{\sqrt{c}} \rceil$. Then $n > \frac{1}{\sqrt{c}}$, so that $\frac{1}{n} < \sqrt{c}$, hence $\frac{1}{n^2} < c$.

It should be clear from our characterization of 0-limits applied to these examples that $f(n) \to \infty$[2] exactly for

$$\lim_{n\to\infty} \frac{1}{f(n)} = 0. \tag{1.1}$$

Similarly, $f(n) \to 0$ exactly when

$$\lim_{n\to\infty} \frac{1}{f(n)} = \infty. \tag{1.2}$$

To generalize, we could note that the continuity of the multiplicative inverse says that $f(n) \to a \in (0, \infty)$ exactly for

$$\lim_{n\to\infty} \frac{1}{f(n)} = \frac{1}{a}. \tag{1.3}$$

Now we could say that the first two limits 1.1, and 1.2 are just special cases of 1.3 thereby giving meaning to the conventions $\frac{1}{\infty} = 0$, and $\frac{1}{0} = \infty$. Another appeal to continuity establishes that for $f(n) \to c$,

$$a^{f(n)} \to a^c.$$

1.3.2 Limits and Logs

Owing to its definition as the inverse of the exponential function, log functions are also continuous, which means that for $f(n) \to c$,

$$\log_b(f(n)) \to \log_b(c).$$

[2] $f(n) \to L$ is a convenient shorthand for $lim_{n\to\infty} f(n) = L$

The natural base e can itself be defined by a particular limit,

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n.$$

You should use your calculator to appreciate this by using a large value for n, say $n = 10^9$ in the expression $\left(1+\frac{1}{n}\right)^n$.

1.3.3 Limits and Derivatives

Our goal here is not to relate derivatives to the limits that define them but rather to introduce a tool that makes the derivative incredibly useful in evaluating the kinds of limits we will require for order comparisons. This tool is L'Hopital's rule which, restated for our purposes says that given functions $f(n), g(n)$ where the limit

$$\lim_{n\to\infty}\frac{f'(n)}{g'(n)}$$

exists, then the limit

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}$$

also exists and the two limits are the same.

In the comparison of algorithms, one is almost exclusively interested in limits of the form:

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}$$

which can sometimes be less than amenable to direct evaluation. By using L'Hopitals rule and taking advantage of the sometimes simplifying property of differentiation, these troublesome limits can often be coaxed to submit to analysis.

Example 1.3. $\lim_{n\to\infty}\frac{log_b(n)}{n} = 0$

$$\begin{aligned}\lim_{n\to\infty}\frac{log_b(n)}{n} &\stackrel{L'}{=} \lim_{n\to\infty}\frac{\frac{1}{n\ln(b)}}{1}\\ &= \frac{1}{\ln(b)}\lim_{n\to\infty}\frac{1}{n}\\ &= 0\end{aligned}$$

Example 1.4. $\lim_{n\to\infty} \frac{b^{n^2}}{n^3} = \infty$

$$\begin{aligned}
\lim_{n\to\infty} \frac{b^{n^2}}{n^3} &\overset{L'}{=} \lim_{n\to\infty} \frac{b^{n^2}\ln(2)2n}{3n^2} \\
&= \frac{2\ln(2)}{3} \lim_{n\to\infty} \frac{b^{n^2}}{n} \\
&\overset{L'}{=} \frac{2\ln(2)}{3} \lim_{n\to\infty} \frac{b^{n^2}\ln(2)2n}{1} \\
&= \frac{4\ln^2(2)}{3} \lim_{n\to\infty} nb^{n^2} \\
&= \infty
\end{aligned}$$

Example 1.5. $\lim_{n\to\infty} \frac{n^2}{b^n} = 0$

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^2}{b^n} &\overset{L'}{=} \lim_{n\to\infty} \frac{2n}{b^n\ln(b)} \\
&= \frac{2}{\ln(b)} \lim_{n\to\infty} \frac{n}{b^n} \\
&\overset{L'}{=} \frac{2}{\ln(b)} \lim_{n\to\infty} \frac{1}{b^n\ln(b)} \\
&= \frac{2}{\ln^2(b)} \lim_{n\to\infty} \frac{1}{b^n} \\
&= 0
\end{aligned}$$

1.4 Combinatorics

One often has occasion to choose sequences or subsets from a given finite set of elements. Sequences differ from subsets in that an order is specified for the elements of a sequence and not for the elements of a subset. For a given subset there can be many sequences using that subset's elements in different orders. For example, consider the subset $X = \{2, 3, 7\}$ of the superset $S = \{1, 2, 3, \ldots, n\}$. We can enumerate six distinct sequences using the elements of X:

$$\begin{aligned} s_1 &= (2, 3, 7) \\ s_2 &= (2, 7, 3) \\ s_3 &= (3, 2, 7) \\ s_4 &= (3, 7, 2) \\ s_5 &= (7, 2, 3) \\ s_6 &= (7, 3, 2). \end{aligned}$$

It is a simple matter to count the number of sequences based on the size of the subset of elements from which the sequences are constructed. For a given subset of m elements, any of those m elements may be chosen as the first element of the sequence. For each of these m first choices, any of the remaining $m-1$ elements may be chosen as the second element of the sequence giving $m(m-1)$ possible choices for the first two elements. For any of these there are $m-2$ remaining elements from which to choose the third, $m-3$ for the fourth, and so on. Consequently we see that the number of different sequences, called permutations, that can be chosen from a given set of m elements is $m!$. We are also interested in the total number of m element permutations chosen over the whole superset S. Since we already know that any m element subset contributes $m!$ permutations we need only know the number of different m element subsets of S. Denoting the number of m element subsets, each called a combination, of a set of n elements by $C(n, m)$, and the number of m element permutations by $P(n, m)$, the number of m element permutations is therefore

$$P(n, m) = C(n, m)m!. \tag{1.4}$$

Approaching the problem from another direction, if p is a permutation of m elements of S, then by appending the remaining $n-m$ elements in all possible $(n-m)!$ permutations gives all permutations of S beginning with the permutation p. Since there must be $n!$ permutations of S we have $n! = P(n, m)(n-m)!$ and therefore

$$P(n, m) = \frac{n!}{(n-m)!}. \tag{1.5}$$

By 1.4 we then have

$$C(n, m) = \frac{n!}{(n-m)!m!}. \tag{1.6}$$

In keeping with the more standard notation we henceforth refer to $C(n, m)$ by the symbol $\binom{n}{m}$ and by the name binomial coefficient. These symbols are called binomial coefficients due to the well known Binomial Theorem.

Theorem 1.4.1. $(a + b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$.

Proof. We offer a simple combinatorial explanation for proof. An inductive proof is given in Theorem A.2.3. Considering the product $(a+b)^n$ as a sequence of factors

$$(a + b)(a + b) \dots (a + b),$$

each term of the resulting product has n factors each of which is either a or b depending on which was chosen. Letting k be the number of b factors chosen, the number of a factors must be $n - k$ and the resulting term is $a^{n-k}b^k$. The number of ways this particular term may arise is the number of ways to choose k of the n factors $(a + b)$ as those from which b was chosen for the term, that is $\binom{n}{k}$. Thus all of the terms $a^{n-k}b^k$ contribute the expression $\binom{n}{k}a^{n-k}b^k$ to the simplified product. As any number of b factors between $k = 0$ and $k = n$ could be chosen the simplified product has the form

$$\sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k.$$

□

Several theorems on binomial coefficients are collected in the Mathematical Reference A.2.

1.5 Graph Theory

Elementary graph theory is a descriptive subset of combinatorics particularly well suited to modeling many problems arising in computer science. A graph is basically a network diagram consisting of labeled vertices, and weighted edges between pairs of those vertices. Graphs represent relations on the set of vertices where the edges are used to express related elements. Edges can be directed from one vertex to another indicating non-reflexive relations or undirected for reflexive relations. Transitivity of relations is naturally depicted as a path within a graph. For simplicity we will concern ourselves only with undirected graphs.

More formally, we define a graph G of n *vertices* and m *edges* as a triple,

$$G = (V, E, W),$$

where

$$V = \{1, 2, 3, \ldots, n\}$$

is the set of vertices,

$$E = \{e_1, e_2, e_3, \ldots, e_m\} \subset V \times V$$

is the set of edges, and

$$W : E \to [0, \infty]$$

is the *Weight Function*. It is convenient to employ the notation $|A|$ for the size or cardinality of a set A, thus $n = |V|$ and $m = |E|$. It is natural to represent graphs with simple diagrams wherein the vertices are circles containing the vertex number, edges are lines drawn between pairs of vertices, and the weight function accounts for numeric labels on the edges. Figure 1.1 illustrates this graphic representation.

The advantages of the diagrams are readily appreciated. Many problems call for the extraction of a *subgraph* from a given graph. Given a graph $G = (V, E, W)$, a subgraph

$$G' = (V', E', W'),$$

is a graph where

$$V' \subset V,$$

$$E' \subset E \cap (V' \times V'),$$

and

$$W' = W|_{E' \times [0,\infty]}.$$

Example 1.6. $G_0 = (\{1,2,3,4\}, \{\{1,3\},\{1,4\},\{2,3\}\}, \{(\{1,3\},2.1),(\{1,4\},3),(\{2,3\},1.5)\}$

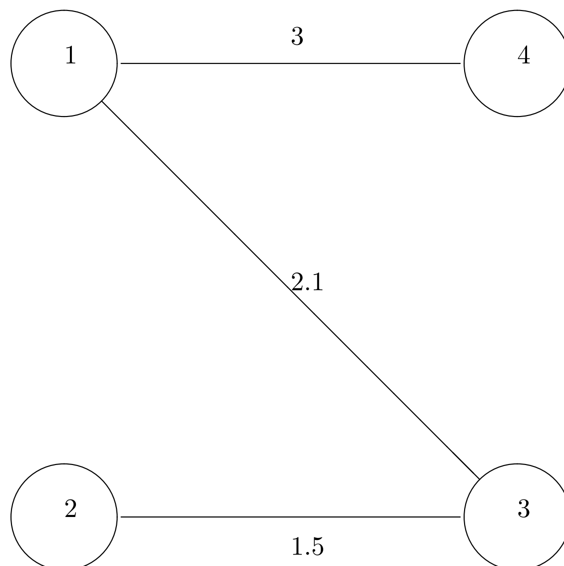

Figure 1.1: Graphic Representation for a Graph.

Here the restriction symbol | has been used to specify those elements of W whose edges lie in E'. Note that every graph contains the empty subgraph. The example contains many subgraphs some of which are listed in Figure 1.2.

The example in the figure shows a connected graph, that is, a graph for which each pair of vertices enjoys a *Path*, or sequence of edges representing the transitive chain relating the vertex pair. Paths are of primary importance in graph theory, often being the objective for a particular type of problem represented by a graph. It should be clear that a connected graph of n vertices must have at least $n-1$ edges. This is a minimal requirement for connectivity. Also, any graph of exactly $n-1$ edges can have no looping paths, a concept more precisely defined later.

1.5.1 Trees

As trees abound in algorithms we make a few fundamental observations concerning them here.

$$G_1' = \{\{1,3,4\},\{\{1,3\},\{1,4\}\}\{(\{1,3\},2.1),(\{1,4\},3)\}\}$$
$$G_2' = \{\{2,3,4\},\{\{2,3\}\},\{(\{2,3\},1.5)\}\}$$
$$G_3' = \{\{1,2\},\emptyset,\emptyset\}$$
$$G_4' = \emptyset$$

Figure 1.2: A Few Subgraphs of G_0

Definition 1.5.1. *A tree is a minimally connected graph.*

As noted above a tree is a connected graph of n vertices and exactly $n-1$ edges. Many important problems require the extraction of a tree from a connected graph. Such subgraphs are normally referred to as *Spanning Trees*. A *rooted tree* is a pair $R = (T, v_0)$ where $T = (V, E, W)$ is a minimally connected graph and $v_0 \in V$. Accordingly, to any n-vertex tree can be associated n rooted trees. The *height* of a rooted tree is the maximum length over all paths in the tree having the root vertex as an endpoint. Thus a tree consisting of a single vertex has height zero. A non-root vertex which is an element of only one edge is called a *leaf*. A non-root vertex is a *child* of the vertex to which it is connected on the path from that vertex to the root vertex. A *balanced tree* is a rooted tree having no paths from the root to leaf vertices of height less than $h-1$ for a tree of height h. A *binary tree* is a rooted tree in which each non-root vertex can be an element of one, two, or three edges, while the root can be an element of one or two edges. For each non-root vertex, of the possible three edges, the edges not leading to the root vertex point to the child vertices. The first such edge, if it exists, is connected to the vertex's *left child* whild the second edge, again if it exists, is connected to the vertex's *right child*. This designation is only of notational convenience and reflects the convention of labeling vertices from left to right in planar representations of trees. A balanced binary tree is *left-balanced* if all of its leaves at height h occur in a contiguous group at the left of the tree, that is as children of an unbroken group of left-most vertices all of whom, with the possible exception of the right-most, have both children. In the exceptional case, the right-most of the group of vertices can have a single left-child. Given an ordered set of keys, $(K, \leq)$, a tree T of elements from K is a heap if for every subtree T' of T, the root of T' is maximal within T'.

1.5.2 Representation in Code

Graphs in general are easily encoded in two-dimensional arrays. Given a graph $G = (V, E, W)$ with $n = |V|$, we define the *Adjacency Matrix* for G as the tabular representation of its weight function W. Thus the adjacency matrix for Figure 1.1 is given in Figure 1.3.

Since each edge in an undirected graph is equivalent to two edges in a directed graph one sees that the adjacency matrix of an undirected graph is necessarily

W	1	2	3	4
1	0	∞	2.1	3
2	∞	0	1.5	∞
3	2.1	1.5	0	∞
4	3	∞	∞	0

Figure 1.3: Adjacency Matrix Representation of a Graph

symmetric, or put another way, unless the adjacency matrix is symmetric the graph is not undirected, having at least one edge between two vertices that is not reversed. Since we will deal here only with undirected graphs we prefer to represent the adjacency matrix with only its upper half, that is, above the main diagonal. The desired representation for the last adjacency matrix is then given in Figure 1.4. Since the diagonal holds no information that cannot be inferred it is also omitted.

W	2	3	4
1	∞	2.1	3
2		1.5	∞
3			∞

Figure 1.4: Preferred Adjacency Matrix Representation For Undirected Graph

Another important encoding is available for left-balanced binary trees. In fact, any left-balanced binary tree can be mapped directly onto an array. If T is a left-balanced binary tree of n elements, then the root is stored in an array A_T at index 0. Given an element t of T stored at a particular location i of the array, locate the left child of $t = A[i]$ at position $2i + 1$ and the right child at position $2n + 2$. Then $A[0..n - 1]$ stores the tree in breadth-first order.

1.6 Probability

1.6.1 Mathematical Expectation

For a discrete random variable X with values $\{X_1, X_2, \ldots, X_i, \ldots\}_{i\in I}$ we define the expectation $E(X)$ as the weighted average

$$E(X) = \sum_{i\in I} X_i P(X = X_i),$$

where $P(X = X_i)$ is the probability that the random variable X assumes the value X_i. For X taking on the finite set of values $1, 2, 3, \ldots, n$, this becomes:

$$E(X) = \sum_{i=1}^{n} iP(X = i).$$

1.6.2 Change of Variable

When it is easier to use a function of a random variable than to use the random variable directly, the following theorem can be used to compute expectations.

Theorem 1.6.1. *If X and Y are random variables and f is a function such that $X = f(Y)$, then $E(X) = E(f(Y))$.*

In the case of discrete random variables X and Y, with sets of values $\{X_i : i \in I\}$, and $\{Y_j : j \in J\}$ respectively, this becomes

$$\begin{aligned}\sum_{i\in I} X_i P(X = X_i) &= E(X) \\ &= E(f(Y)) \\ &= \sum_{j\in J} f(Y_j)P(Y = Y_j)\end{aligned}$$

For Y taking on the finite set of values $1, 2, 3, \ldots, m$, this becomes:

$$\begin{aligned}E(X) &= E(f(Y)) \\ &= \sum_{j\in J} f(Y_j)P(Y = Y_j) \\ &= \sum_{j=1}^{m} f(j)P(Y = j)\end{aligned}$$

1.7 Sums

In our investigations several elementary sums will occur with some regularity.

Theorem 1.7.1. $\sum_{i=1}^{m} i = \frac{1}{2}m(m+1)$

Proof. Let S denote the sum. Then

$$\begin{aligned} 2S &= \sum_{i=1}^{m} i + \sum_{i=1}^{m} (m+1-i) \\ &= \sum_{i=1}^{m} (m+1) \\ &= m(m+1) \end{aligned}$$

□

Lemma 1.7.1. $\sum_{i=1}^{m} ia_i = \sum_{i=1}^{m}\sum_{j=i}^{m} a_j = \sum_{i=1}^{m}\left(\sum_{j=1}^{m} a_j - \sum_{j=1}^{i-1} a_j\right)$

Theorem 1.7.2. $\sum_{i=1}^{m} i^2 = \frac{1}{6}m(m+1)(2m+1)$

Proof. Let S denote the sum. Using Theorem 1.7.1 and Lemma 1.7.1 we have

$$\begin{aligned} S &= \sum_{i=1}^{m}\sum_{j=i}^{m} j \\ &= \sum_{i=1}^{m}\left(\sum_{j=1}^{m} j - \sum_{j=1}^{i-1} j\right) \\ &= \sum_{i=1}^{m}\left(\frac{1}{2}m(m+1) - \frac{1}{2}(i-1)i\right) \\ &= \frac{1}{2}\left[m^2(m+1) - \sum_{i=1}^{m} i^2 + \sum_{i=1}^{m} i\right] \\ &= \frac{1}{2}\left[m^2(m+1) - S + \frac{1}{2}m(m+1)\right] \end{aligned}$$

After some rearrangement we obtain

$$\begin{aligned} 6S &= 2m^2(m+1) + m(m+1) \\ &= m(m+1)(2m+1) \end{aligned}$$

□

Theorem 1.7.3 (Geometric Partial Sum)**.** $\sum_{i=0}^{m} a^i = \frac{a^{m+1}-1}{a-1}$

Proof. Let S denote the sum. Then

$$\begin{aligned}
(a-1)S &= aS - S \\
&= a\sum_{i=0}^{m} a^i - \sum_{i=0}^{m} a^i \\
&= \sum_{i=0}^{m} a^{i+1} - \sum_{i=0}^{m} a^i \\
&= a^{m+1} + \sum_{i=0}^{m-1} a^{i+1} - \sum_{i=0}^{m} a^i \\
&= a^{m+1} + \sum_{j=1}^{m} a^j - \sum_{i=0}^{m} a^i \\
&= a^{m+1} - 1
\end{aligned}$$

□

Theorem 1.7.4 (Polygeometric Sum)**.** $\sum_{i=1}^{m} i2^i = (m-1)2^{m+1} + 2$

Proof. Using the Lemma and Theorem 1.7.3 we have

$$\begin{aligned}
\sum_{i=1}^{m} i2^i &= \sum_{i=1}^{m}\sum_{j=i}^{m} 2^j \\
&= \sum_{i=1}^{m}\left(\sum_{j=0}^{m} 2^j - \sum_{j=0}^{i-1} 2^j\right) \\
&= \sum_{i=1}^{m}\left(2^{m+1} - 1 - [2^i - 1]\right) \\
&= m2^{m+1} - \sum_{i=1}^{m} 2^i \\
&= m2^{m+1} - \left(\sum_{i=0}^{m} 2^i - 1\right) \\
&= m2^{m+1} - \left(2^{m+1} - 1 - 1\right) \\
&= (m-1)2^{m+1} + 2
\end{aligned}$$

□

Higher order sums such as 1.7.4 can be constructed using some tricks from calculus as shown in section A.3.1.

1.8 Induction

Occasionally we rely on a particularly powerful proof technique which can be used to establish properties on an ordered set. It will be sufficient for our purposes to deal with propositions on the set of positive integers though the technique is valid in a much broader context. We state without proof the induction theorem.

Theorem 1.8.1. *Given a property P_k which may be stated for any integer k, If P_{k_0} is true and if the truth of P_k implies the truth of P_{k+1} where $k \geq k_0$, then P_k is true for all $k \geq k_0$.*

This form of the induction theorem uses what is called the weak inductive hypothesis. An alternative but completely equivalent statement using the strong inductive hypothesis is:

Theorem 1.8.2. *Given a property P_k which may be stated for any nonnegative integer k, If P_{k_0} is true and if the assumption that $\forall i \in \{k_0, \ldots, k\}$, P_i is true implies the truth of P_{k+1}, then P_k is true for all $k \geq k_0$.*

Example 1.7. $\sum_{i=1}^{m} i2^i = (m-1)2^{m+1} + 2$
First we note that equality holds for the base case of $m = 1$. Now assume $\sum_{i=1}^{k} i2^i = (k-1)2^{k+1} + 2$ for $k \geq 1$. Then

$$\begin{aligned}
\sum_{i=1}^{k+1} i2^i &= (k+1)2^{k+1} + \sum_{i=1}^{k} i2^i \\
&= (k+1)2^{k+1} + (k-1)2^{k+1} + 2 \\
&= [(k+1) + (k-1)]2^{k+1} + 2 \\
&= 2k2^{k+1} + 2 \\
&= k2^{k+2} + 2 \\
&= [(k+1) - 1]2^{(k+1)+1} + 2
\end{aligned}$$

By Theorem 1.8.1 we have established the sum for any m.

Example 1.8. *Using formula 1.6 as the definition for $\binom{n}{k}$,*

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k} + \binom{n-1}{k-1} & ,0 < k < n \\ 1 & ,k \in \{0, n\} \end{cases}$$

For the base case $n = 1$ the new formula agrees since $\frac{1!}{0!} = \frac{1!}{1!} = 1$, $\binom{0}{1} = 0$, and $\binom{1}{0} = 1$. Now assume that $\frac{n!}{k!(n-k!)} = \binom{n}{k}$ for $n \geq 1$, that is that

$$\frac{n!}{k!(n-k!)} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Then

$$\begin{aligned}\binom{n}{k} + \binom{n}{k-1} &= \frac{n!}{k!(n-k!)}\frac{n-k+1}{n-k+1} + \frac{n!}{(k-1)!(n-k+1!)}\frac{k}{k} \\ &= \frac{n!}{k!(n-k+1)!}[n-k+1+k] \\ &= \frac{(n+1)!}{k!(n+1-k!)} \\ &= \binom{n+1}{k}.\end{aligned}$$

I

Order

Chapter 2

Order Comparison

Two solutions for the problem of Fibonacci number generation are used first, to illustrate the dramatic differences between correct solutions to the same problem, and second, to show how difficult a manual head-to-head fair comparison of running times can be without the aid of the ideas of algorithm analysis.

Consider the problem of generating the nth Fibonacci number given a non-negative integer n, $n \to F_n$, where

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & , n > 1 \\ n & , n = 0, 1 \end{cases} .$$

The following two algorithms, *fib1* and *fib2* (Figure 2.1), are both correct, although quite different, solutions for this problem.

```
int fib1(int n){
if(n<2)
   return n;
else {
   int x=0,y=1;
   for(int i=1;i<n;i++){
      y=x+y;
      x=y-x;}
return y;}}

int fib2(int n){
if(n<2)
   return n;
else
   return fib1(n-1)+fib1(n-2);}
```

Figure 2.1: Fibonacci Algorithms

To attempt a comparison of the two algorithms we introduce the notion of the number X of instructions executed by the algorithm when run with a particular input n. We emphasize the dependence on both the algorithm and the language used to implement the algorithm. Thus for the first algorithm we have

$$X_{[fib1,C]}(n) = 4 + (n-1) \cdot 2 = 2n + 2, \quad (n > 1).$$

Here we have counted the variable declarations and the return each as one instruction each and the body of the loop as two instructions. This might lead to some differences of opinion as to how instructions should be counted. Perhaps the initialization within the loop declaration and both the loop test and loop variable increment should have been counted as well giving the following:

$$X_{[fib1,C]}(n) = 5 + (n-1) \cdot 4 = 4n - 1, \quad (n > 1).$$

This apparent ambiguity may lead us to use a finer language for algorithm expression. Using the gcc compiler to produce assembly code for a Sun SPARC, we obtain the assembly code in Figure 2.2 from fib1.c.
After some inspection of actual instructions executed we now come up with something like

$$X_{[fib1,SPARC]}(n) = 28 + (n-1) \cdot 16.$$

If we are to compare the two algorithms we need to find the corresponding quantity $X_{[fib2,SPARC]}(n)$. After compilation of fib2.c we obtain the assembly code in Figure 2.3.
The recursive character of the routine makes it much more difficult to count steps. Counting yields

$$X_{[fib2,SPARC]}(n) = 11 + (C(n) - 1) \cdot 20 = 20C(n) - 9,$$

where $C(n)$ is the number of function calls. We will satisfy ourselves with a lower bound for $C(n)$ using induction. For $n = 0, 1$ we have $C(n) = 1$, $2^{\frac{0-1}{2}} = \frac{1}{\sqrt{2}}$, and $2^{\frac{1-1}{2}} = 1$, hence $C(n) > 2^{\frac{n-1}{2}}$ in both cases. Invoking the strong inductive hypothesis for $k < n$ we have

$$\begin{aligned}
C(n) &> C(n-1) + C(n-2) \\
&> 2^{\frac{(n-1)-1}{2}} + 2^{\frac{(n-2)-1}{2}} \\
&= 2^{\frac{n-2}{2}} + 2^{\frac{n-3}{2}} \\
&> 2^{\frac{n-3}{2}} + 2^{\frac{n-3}{2}} \\
&= 2 \cdot 2^{\frac{n-3}{2}} \\
&= 2^{1+\frac{n-3}{2}} \\
&= 2^{\frac{n-1}{2}}
\end{aligned}$$

```
   .file "fib1.c"
gcc2_compiled.:
.section ".text"
   .align 4
   .global f1
   .type  f1,#function
   .proc 04
f1:
   !#PROLOGUE# 0
   save %sp,-128,%sp
   !#PROLOGUE# 1
   st %i0,[%fp+68]
   ld [%fp+68],%o0
   cmp %o0,1
   bg .LL2
   nop
   ld [%fp+68],%o0
   mov %o0,%i0
   b .LL1
   nop
   b .LL3
   nop
.LL2:
   st %g0,[%fp-20]
   mov 1,%o0
   st %o0,[%fp-24]
   mov 1,%o0
   st %o0,[%fp-28]
.LL4:
   ld [%fp-28],%o0
   ld [%fp+68],%o1
   cmp %o0,%o1
   bl .LL7
   nop
   b .LL5
   nop
.LL7:
   ld [%fp-24],%o0
   ld [%fp-20],%o1
   add %o0,%o1,%o0
   st %o0,[%fp-24]
   ld [%fp-24],%o0
   ld [%fp-20],%o1
   sub %o0,%o1,%o0
   st %o0,[%fp-20]
.LL6:
   ld [%fp-28],%o0
   add %o0,1,%o1
   st %o1,[%fp-28]
   b .LL4
   nop
.LL5:
   ld [%fp-24],%o0
   mov %o0,%i0
   b .LL1
   nop
.LL3:
.LL1:
   ret
   restore
.LLfe1:
   .size  f1,.LLfe1-f1
   .ident
"GCC: (GNU) 2.8.1"
```

Figure 2.2: Assembly Code for *Fib1*

```
    .file "fib2.c"
gcc2_compiled.:
.section ".text"
    .align 4
    .global f2
    .type  f2,#function
    .proc 04
f2:
    !#PROLOGUE# 0
    save %sp,-112,%sp
    !#PROLOGUE# 1
    st %i0,[%fp+68]
    ld [%fp+68],%o0
    cmp %o0,1
    bg .LL2
    nop
    ld [%fp+68],%o0
    mov %o0,%i0
    b .LL1
    nop
    b .LL3
    nop
.LL2:
    ld [%fp+68],%o0
    add %o0,-1,%o1
    mov %o1,%o0
    call f1,0
    nop
    mov %o0,%l0
    ld [%fp+68],%o0
    add %o0,-2,%o1
    mov %o1,%o0
    call f1,0
    nop
    mov %o0,%o1
    add %l0,%o1,%o0
    mov %o0,%i0
    b .LL1
    nop
.LL3:
.LL1:
    ret
    restore
.LLfe1:
    .size  f2,.LLfe1-f2
    .ident
"GCC: (GNU) 2.8.1"
```

Figure 2.3: Assembly Code for *Fib2*

Now that

$$X_{[fib2,SPARC]}(n) > 20 \cdot 2^{\frac{n-1}{2}} - 9$$

is established, the two algorithms can be compared. To find the time $T_{\mathcal{A}}(n)$, required for a particular algorithm $\mathcal{A}$, we simply multiply the executed instruction count $X_{[\mathcal{A},machine]}(n)$ by the number r of machine cycles per instruction and by the inverse of s, the processor speed,

$$T_{\mathcal{A}}(n) = X_{[\mathcal{A},machine]}(n)(r/s).$$

Assuming the SPARC processor in use runs at 500MHz and that machine instructions require 2 cycles, we obtain

$$\begin{aligned} T_{fib1}(n) &= X_{[fib1,SPARC]}(n)(2)(\frac{1}{500000000}) \\ &= \frac{4n-1}{250000000} \\ T_{fib2}(n) &= X_{[fib2,SPARC]}(n)(2)(\frac{1}{500000000}) \\ &> \frac{20(\sqrt{2})^{n-1}-9}{250000000} \end{aligned}$$

We now choose a modest value for n, say $n = 99$, giving:

$$\begin{aligned} T_{fib1}(n) &= \frac{4 \cdot 99 - 1}{250000000} \\ &= 1.58 \times 10^{-6} \text{ (sec.)} \\ T_{fib2}(n) &> \frac{20(\sqrt{2})^{98}-9}{250000000} \\ &= 1.13 \times 10^{16} \text{ (sec.)} \\ &= 357 \text{ million years.} \end{aligned}$$

We begin to appreciate the vast difference between multiple correct solutions for a given problem.

For our next problem consider integer exponentiation; that is, given a base b and a non-negative integer exponent n, find an algorithm for computing b^n. We again supply two correct solutions, *exp1*, and *exp2* (Figure 2.4) which will have vastly differing character in terms of running time.
The student new to *exp2* should note that it computes 2^{15} as shown in Figure 2.5, using recursion to move from larger to smaller powers. This algorithm will be presented again later in the text.

Now we count steps for both algorithms. For *exp1* we shall settle on

$$X_{exp1,C}(n) = 3n + 2, \tag{2.1}$$

```
float exp1(float b,int n){
float p=1.0;
while(n>0){
   p=p*b;
   n--;}
return p;}

float exp2(float b,int n){
if(n==0)
   return 1.0;
float p=exp2(b,n/2);
p=p*p;
if(n%2)
   p=p*b;
return p;}
```

Figure 2.4: Integer Exponentiation Algorithms

$$
\begin{aligned}
2^{15} &= 2 \cdot 2^{14} \\
&= 2 \cdot (2^7)^2 \\
&= 2 \cdot (2 \cdot 2^6)^2 \\
&= 2 \cdot (2 \cdot (2^3)^2)^2 \\
&= 2 \cdot (2 \cdot (2 \cdot 2^2)^2)^2
\end{aligned}
$$

Figure 2.5: Fast Integer Exponentiation

while for *exp2* we have

$$X_{exp2,C}(n) = 5 + C(n),$$

where again we will not quibble over the number 5 (some may have counted 4 and others 6), and $C(n)$ again represents the number of recursive calls. To find $C(n)$ we will do what some mathematicians call following the rabbit and write

$$\begin{aligned} C(n) &= 5 + C\left(\frac{n}{2}\right) \\ &= 5 + 5 + C\left(\frac{\frac{n}{2}}{2}\right) \\ &= 5 + 5 + 5 + C\left(\frac{\frac{\frac{n}{2}}{2}}{2}\right) \\ &\vdots \\ &= 5k + C\left(\frac{n}{2^k}\right) \end{aligned}$$

We now reason that eventually $n/2^k$ will reach 1 at which point we are finished. Assuming the value $C(1) = 6$, the time for the call to the $n = 0$ case plus the test for $n = 0$, we have

$$C(n) = 5k + 6.$$

However as $n/2^k = 1$ we have $k = \log_2(n)$ hence

$$X_{exp2,C}(n) = 5 + 5k + 6 = 5\log_2(n) + 11. \tag{2.2}$$

How do the numbers of steps differ chaning from the computation b^n to b^{2n}? For *exp1*, one as $X_{exp1,C}(2n) = 3(2n) + 2 = 6n + 2$, which for large n is approximately double $X_{exp1,C}(n)$. On the other hand

$$X_{exp2,C}(2n) = 5\log_2(2n) + 11 = 5\log_2(n) + 12$$

which is only one more step than $X_{exp2,C}(n)$. Clearly *exp2* has a dramatic advantage. This can be seen by graphically comparing the running times for *exp1* and *exp2*.

2.1 Equivalent Order

From our analysis of fibonacci algorithms we have seen that the counts of instructions executed depends a great deal on the language used to implement the algorithm. We have also seen that making an instruction count is quite tedious and time consuming. If we are to compare algorithms we would prefer some system that does not need to take language detail into account; some measure should be characteristic of the algorithm itself irrespective of implementation language. If we compare the instruction counts for the same algorithm and different languages there is an obvious similarity between them. For *fib1*, and *fib2* we obtained

$$X_{[fib1,C]}(n) = 2n + 2$$
$$X_{[fib1,SPARC]}(n) = 4n - 1$$

and,

$$X_{[fib2,SPARC]}(n) = 20(\sqrt{2})^n + 11.$$

We might generalize and say that no matter what the language, the executed instruction counts for *fib1* and *fib2* will always be of the form:

$$X_{[fib1,-]}(n) = c_1 n + c_2$$
$$X_{[fib2,-]}(n) = c_3 b^n + c_4$$

The difference in language seems only to account for changes in the constants in the general formulation. We therefore agree to disregard such constant differences. Since the time is obtained from the executed instruction count by multiplying by more constants, we are contented that the algorithms are properly characterized by these generalized instruction counts for purposes of comparison. We have thus decided to formulate our notion of time complexity in such a way as to be able to say that $f(n) = 4n + 3$ and $g(n) = 13n + 11$ are somehow the same. And for $f(n) = 2(\sqrt{2})^n + 1$ and $g(n) = 20(\sqrt{2})^n + 11$, we declare that f and g are equivalent.

Definition 2.1.1. *Two nonnegative functions f and g have the same order (written $f \simeq g$) if the sequence $\left(\frac{f(n)}{g(n)}\right)_{n=1}^{\infty}$ of quotients has finite non-zero superior and inferior subsequential limits; that is if*

$$\left[\liminf_{n\to\infty} \frac{f(n)}{g(n)}, \limsup_{n\to\infty} \frac{f(n)}{g(n)}\right] \subset (0, \infty).$$

If the limit of the quotient exists (which will usually be the case in our studies) then a simpler definition suffices:

Definition 2.1.2. *Two nonnegative functions f and g have the same order (written $f \simeq g$) if the limit of the quotient of f and g as $n \to \infty$ is a positive*

real number; that is,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0, \infty).$$

Since the inversion of a positive real number is still a positive real number, we see that $f \simeq g$ if and only if $g \simeq f$. When we examine our instruction counts, this definition appears to do exactly what we need.

Example 2.1. $4n - 1 \simeq 2n + 2$

$$\begin{aligned}\lim_{n\to\infty} \frac{4n-1}{2n+2} &= \lim_{n\to\infty} \frac{\frac{4n-1}{n}}{\frac{2n+2}{n}} \\ &= \lim_{n\to\infty} \frac{4 - \frac{1}{n}}{2 + \frac{2}{n}} \\ &= 2 \in (0, \infty)\end{aligned}$$

Example 2.2. $2n + 2 \simeq 4n - 1$

$$\begin{aligned}\lim_{n\to\infty} \frac{2n+2}{4n-1} &= \lim_{n\to\infty} \frac{\frac{2n+2}{n}}{\frac{4n-1}{n}} \\ &= \lim_{n\to\infty} \frac{2 + \frac{2}{n}}{4 - \frac{1}{n}} \\ &= \frac{1}{2} \in (0, \infty)\end{aligned}$$

Example 2.3. $2(\sqrt{2})^n + 1 \simeq 20(\sqrt{2})^n + 11$

$$\begin{aligned}\lim_{n\to\infty} \frac{2(\sqrt{2})^n + 1}{20(\sqrt{2})^n + 11} &= \lim_{n\to\infty} \frac{\frac{2(\sqrt{2})^n+1}{(\sqrt{2})^n}}{\frac{20(\sqrt{2})^n+11}{(\sqrt{2})^n}} \\ &= \lim_{n\to\infty} \frac{2 + \frac{1}{(\sqrt{2})^n}}{20 + \frac{11}{(\sqrt{2})^n}} \\ &= \frac{1}{10} \in (0, \infty)\end{aligned}$$

Example 2.4. $2(\sqrt{2})^n + 1 \not\simeq 2n + 2$

$$\begin{aligned}
\lim_{n\to\infty} \frac{2(\sqrt{2})^n + 1}{2n+2} &= \lim_{n\to\infty} \frac{\frac{2(\sqrt{2})^n+1}{n}}{\frac{2n+2}{n}} \\
&= \lim_{n\to\infty} \frac{\frac{2(\sqrt{2})^n}{n} + \frac{1}{n}}{2 + \frac{2}{n}} \\
&= \frac{1}{2} \lim_{n\to\infty} \frac{(\sqrt{2})^n}{n} \\
&\stackrel{L'}{=} \frac{1}{2} \lim_{n\to\infty} \frac{(\sqrt{2})^n \ln(2)}{1} \\
&= \frac{\ln 2}{2} \lim_{n\to\infty} (\sqrt{2})^n \\
&= \infty \notin (0, \infty)
\end{aligned}$$

Example 2.5. $2n + 2 \not\simeq 2(\sqrt{2})^n + 1$

$$\begin{aligned}
\lim_{n\to\infty} \frac{2n+2}{2(\sqrt{2})^n + 1} &= \lim_{n\to\infty} \frac{\frac{2n+2}{n}}{\frac{2(\sqrt{2})^n+1}{n}} \\
&= \lim_{n\to\infty} \frac{2 + \frac{2}{n}}{\frac{2(\sqrt{2})^n}{n} + \frac{1}{n}} \\
&= 2 \lim_{n\to\infty} \frac{n}{(\sqrt{2})^n} \\
&\stackrel{L'}{=} 2 \lim_{n\to\infty} \frac{1}{(\sqrt{2})^n \ln(2)} \\
&= \frac{2}{\ln 2} \lim_{n\to\infty} \frac{1}{(\sqrt{2})^n} \\
&= 0 \notin (0, \infty)
\end{aligned}$$

Example 2.6. $3n^2 + 1 \simeq \frac{1}{100}n^2 + n$

$$\begin{aligned}
\lim_{n\to\infty} \frac{3n^2 + 1}{\frac{1}{100}n^2 + n} &= \lim_{n\to\infty} \frac{3 + \frac{1}{n^2}}{\frac{1}{100} + \frac{1}{n}} \\
&= 300 \in (0, \infty)
\end{aligned}$$

The next example involving logs may seem somewhat surprising.

Example 2.7. $\ln(x^2+1) \simeq \ln(x)$ *Using L'Hopital's rule we have*

$$\begin{aligned}
\lim_{n\to\infty} \frac{\ln(x^2+1)}{\ln(x)} &\overset{L'}{=} \lim_{n\to\infty} \frac{\frac{2x}{x^2+1}}{\frac{1}{x}} \\
&= \lim_{n\to\infty} \frac{2x^2}{x^2+1} \\
&= \lim_{n\to\infty} \frac{2}{1+\frac{1}{x^2}} \\
&= 2
\end{aligned}$$

It should be clear that for any function f, $f \simeq f$ since

$$\lim_{n\to\infty} \frac{f(n)}{f(n)} = 1 \in (0,\infty).$$

It should also be clear that for constants $c_1, c_2, c_3, c_4 > 0$, that

$$c_1 n^2 + c_2 f(n) \simeq c_3 n^2 + c_4 g(n)$$

as long as both $f(n) \leq n^2$ and $g(n) \leq n^2$, for then

$$\begin{aligned}
0 &\leq \lim_{n\to\infty} \frac{c_1 n^2 + c_2 f(n)}{c_3 n^2 + c_4 g(n)} \\
&= \lim_{n\to\infty} \frac{c_1 + c_2 \frac{f(n)}{n^2}}{c_3 + c_4 \frac{g(n)}{n^2}} \\
&\leq \frac{c_1 + c_2}{c_3} \in (0,\infty)
\end{aligned}$$

In particular, for $c_3 = 1$ and $g = 0$ we have

$$c_1 n^2 + c_2 f(n) \simeq n^2.$$

Thus n^2 is the simplest representative of functions of its order. This whole process could of course have been done with any function $h(n)$ rather than n^2 which tells us that our new notion of order is taken from the most significant part of the expression. That is, if $h(n) = cf(n) + g(n)$ where $g \leq f$, then the order of h is f and we would write $h \simeq f$. It is also true that equivalence of is transitive.

Theorem 2.1.1. *If $f \simeq g$ and $g \simeq h$, then $f \simeq h$.*

Proof. Assuming the limits exist, let l_1, l_2 be the limits of $\frac{f(n)}{g(n)}$ and $\frac{g(n)}{h(n)}$ respec-

tively. Then $l_1, l_2 \in (0, \infty)$ and

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{h(n)} &= \lim_{n\to\infty} \frac{f(n)}{h(n)} \frac{g(n)}{g(n)} \\
&= \lim_{n\to\infty} \frac{f(n)}{g(n)} \frac{g(n)}{h(n)} \\
&= \lim_{n\to\infty} \frac{f(n)}{g(n)} \lim_{n\to\infty} \frac{g(n)}{h(n)} \\
&= l_1 l_2 \in (0, \infty)
\end{aligned}$$

For the general case, if the limit suprema L_1, L_2 are both finite, then

$$\begin{aligned}
\limsup_{n\to\infty} \frac{f(n)}{h(n)} &= \limsup_{n\to\infty} \frac{f(n)}{h(n)} \frac{g(n)}{g(n)} \\
&= \limsup_{n\to\infty} \frac{f(n)}{g(n)} \frac{g(n)}{h(n)} \\
&\leq \limsup_{n\to\infty} \frac{f(n)}{g(n)} \limsup_{n\to\infty} \frac{g(n)}{h(n)} \\
&= L_1 L_2 < \infty
\end{aligned}$$

Similarly, if the limit infima l_1, l_2 are both non-zero, then

$$\begin{aligned}
\liminf_{n\to\infty} \frac{f(n)}{h(n)} &\geq \liminf_{n\to\infty} \frac{f(n)}{g(n)} \liminf_{n\to\infty} \frac{g(n)}{h(n)} \\
&= l_1 l_2 > 0
\end{aligned}$$

□

Finally, order is preserved by multiplication.

Theorem 2.1.2. $\frac{f}{g} \simeq h$ *if and only if* $f \simeq gh$*.*

2.2 Inferior Order

Definition 2.2.1. *For nonnegative functions f and g, f is of inferior order to g (written $f \prec g$) if the limit of the quotient of f and g as $n \to \infty$ is zero; that is,*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

This works out nicely for our comparison of the integer exponentiation algorithms, since comparison of 2.1 and 2.2 will now show

$$X_{exp2,C}(n) \prec X_{exp1,C}(n).$$

Example 2.8. $5\log_2(n) + 11 \prec 3n + 2$

$$\begin{aligned}\lim_{n\to\infty} \frac{5\log_2(n) + 11}{3n+2} &\overset{L'}{=} \lim_{n\to\infty} \frac{\frac{5}{n\ln(2)}}{3} \\ &= \frac{5}{3\ln(2)} \lim_{n\to\infty} \frac{1}{n} \\ &= 0\end{aligned}$$

Further examples show that our new notion of order respects polynomial order.

Example 2.9. $8n + 1 \prec n^2 + 2$

$$\begin{aligned}\lim_{n\to\infty} \frac{8n+1}{n^2+2} &= \lim_{n\to\infty} \frac{\frac{8}{n} + \frac{1}{n^2}}{1 + \frac{2}{n^2}} \\ &= \frac{0}{1} \\ &= 0\end{aligned}$$

More generally,

Theorem 2.2.1. *For $i < j, n^i \prec n^j$.*

Proof.

$$\begin{aligned}\lim_{n\to\infty} \frac{n^i}{n^j} &= \lim_{n\to\infty} \frac{1}{n^{j-i}} \\ &= 0\end{aligned}$$

□

It is just as easily shown that a change of exponential base gives rise to a change in order.

Theorem 2.2.2. *For $0 < a < b, a^n \prec b^n$.*

Proof. Since $\frac{a}{b} < 1$,

$$\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty} \left(\frac{a}{b}\right)^n$$
$$= 0$$

□

The same cannot however be said about the log functions as might be expected.

Theorem 2.2.3. *For* $0 < a < b, \log_a(n) \simeq \log_b(n)$.

Proof.

$$\lim_{n\to\infty} \frac{\log_a(n)}{\log_b(n)} = \lim_{n\to\infty} \frac{\frac{\log_b(n)}{\log_b(a)}}{\log_b(n)}$$
$$= \frac{1}{\log_b(a)} \lim_{n\to\infty} 1$$
$$= \frac{1}{\log_b(a)} \in (0, \infty)$$

□

In terms of order all logs are the same. For this reason analytical results concerning log orders are usually written without the base. There are two ways to compose polynomial and log functions. The only type of interest is a power of a log, $\log_b^i(n)$ Functions of this form are called polylogs.

Theorem 2.2.4. *For* $b > 1, 1 < i < j, \log_b^i(n) \prec \log_b^j(n)$.

Proof.

$$\lim_{n\to\infty} \frac{\log_b^i(n)}{\log_b^j(n)} = \lim_{n\to\infty} \frac{1}{(\log_b(n))^{j-i}}$$
$$= 0$$

□

It is clear that for any function f, $f \not\prec f$, since

$$\lim_{n\to\infty} \frac{f(n)}{f(n)} = 1 \neq 0.$$

Like $\simeq$, the relation $\prec$ is transitive.

Theorem 2.2.5. *If* $f \prec g$ *and* $g \prec h$, *then* $f \prec h$.

Proof.

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{h(n)} &= \lim_{n\to\infty} \frac{f(n)}{h(n)} \frac{g(n)}{g(n)} \\
&= \lim_{n\to\infty} \frac{f(n)}{g(n)} \frac{g(n)}{h(n)} \\
&= \lim_{n\to\infty} \frac{f(n)}{g(n)} \lim_{n\to\infty} \frac{g(n)}{h(n)} \\
&= 0
\end{aligned}$$

□

2.3 Non-Strict Order

It is possible that during evaluation of a limit, one can only establish

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

In this case the limit supremum could be a positive real number or zero, admitting the two possibilities $f \simeq g$ or $f \prec g$. In such cases it is natural to write $f \preceq g$. For this non-strict notion of order, it is clear that for any function $f \preceq f$. It is also clear that $\preceq$ is a transitive relation.

Working with the non-strict order is a useful technique for establishing equivalent order.

Theorem 2.3.1. *$f \simeq g$ if and only if $f \preceq g$ and $g \preceq f$.*

While it makes sense to compare the orders of any two nonnegative functions, ordinary pointwise comparison is not generally possible. That is to say, though it may be tempting to define the relations $f < g$ and $f > g$ to mean $f(n) < g(n)$ for all n and $f(n) > g(n)$ for all n, respectively, we can see that any crossing of the functions makes such global pointwise comparisons impossible. For example, given the functions $f(n) = n + 2$ and $g(n) = \frac{1}{5}n(n+2)$, it doesn't even make sense to say that $f < g$ or $f > g$ pointwise without first restricting the domains of comparison since $g(n) \leq f(n)$ for $n \in (1, 5)$ but $f(n) \leq g(n)$ for $n \in (5, \infty)$. ($f(n) = g(n)$ for $n \in \{0, 5\}$). On the other hand, the comparison $f \prec g$ is easily established.

Example 2.10. *For $f(n) = n + 2$ and $g(n) = \frac{1}{5}n^2 + \frac{2}{5}n$, $f \prec g$*

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n+2}{\frac{1}{5}n^2 + \frac{2}{5}n} \\
&= \lim_{n\to\infty} \frac{\frac{n+2}{n^2}}{\frac{\frac{1}{5}n^2 + \frac{2}{5}n}{n^2}} \\
&= \lim_{n\to\infty} \frac{\frac{1}{n} + \frac{2}{n^2}}{\frac{1}{5} + \frac{2}{5n}} \\
&= 0
\end{aligned}$$

When two functions do compare pointwise without restriction, that is, if $f(n) < g(n)$ or $f(n) \leq g(n)$ for all n, then $f \preceq g$. Moreover, even if the inequality holds only for all sufficiently large values of n, the same order relation still follows. It is for this reason that this type of order is called asymptotic; the pointwise behavior is only assured for large n. One can prove slightly more; since the multiplication a positive constant should not effect the order we have:

Lemma 2.3.1. *Let $c \in (0, \infty), n_0 \in \mathcal{N}$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. Then $f \preceq g$.*

Table 2.1: Comparisons Compared

pointwise comparison	order comparison
$n < 2n$	$n \preceq 2n$
$n < n^2 + 1$	$n \prec n^2 + 1$
$n \leq cn + 4, c > 1$	$n \simeq cn + 4$

Proof.

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} \leq c$$

.

□

It is not enough for $f < g$ pointwise to have $f \prec g$ as one can easily verify by noting that $\frac{1}{2}f(n) < f(n)$ for all n but $\frac{1}{2}f \simeq f$. Table 2.1 illustrates the situation further. In all three examples it is correct to say $f \preceq g$.

Lemma 2.3.1 is easily amplified to form a characterization which serves in most texts as the definition for our notion of $f \preceq g$.

Theorem 2.3.2. *$f \preceq g$ if and only if $\exists c \in (0, \infty), \exists n_0 \in \mathcal{N}$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.*

2.4 Traditional Notation

The notation used in this book is not conventional. It is much more common to define order via order classes, that is, by defining sets of all functions of the same order. In so doing there are usually five different classes defined. The most basic is the set defined for a given function f as

$$O(f) = \{g : \exists c \in (0, \infty), \exists N \geq 0, [n \geq N \Rightarrow g(n) \leq cf(n)]\}.$$

The comparison of functions using $O(f)$ corresponds to our non-strict order comparison; that is, to write $g \preceq f$ is equivalent to writing $g \in O(f)$.

Theorem 2.4.1. *$g \preceq f$ if and only if $g \in O(f)$.*

Proof. Let $g \in O(f)$. Then $\exists c \in (0, \infty), N \geq 0$ such that $g(n) \leq cf(n)$ for $n \geq N$. Then $\limsup_{n\to\infty} \frac{g(n)}{f(n)} \leq c$, giving $g \preceq f$. The argument is readily reversed. □

While $O(f)$ provides a notion of upper bound, reversing the inequality in the definition provides a corresponding notion of lower bound. The order class is defined as

$$\Omega(f) = \{g : \exists c \in (0, \infty), \exists N \geq 0, [n \geq N \Rightarrow g(n) \geq cf(n)]\}.$$

Intersecting these two sets provides the notion of equivalent order; that is, for

$$\Theta(f) = O(f) \bigcap \Omega(f),$$

$g \simeq f$ if and only if $g \in \Theta(f)$ (or equivalently $f \in \Theta(g)$ or $\Theta(f) = \Theta(g)$). A modification of the quantifier is needed to define the notion of inferior order. One obtains two additional order classes defined as

$$o(f) = \{g : \forall c \in (0, \infty), \exists N \geq 0, [n \geq N \Rightarrow g(n) \leq cf(n)]\},$$

and

$$\omega(f) = \{g : \forall c \in (0, \infty), \exists N \geq 0, [n \geq N \Rightarrow g(n) \geq cf(n)]\}.$$

Then $g \prec f$ is expressed as $g \in o(f)$ and $g \in \omega(f)$ means $f \prec g$.

Theorem 2.4.2. *$g \prec f$ if and only if $g \in o(f)$.*

Proof. One need only note that $g \in o(f)$ if and only if $\lim\limits_{n\to\infty} \frac{g(n)}{f(n)} = 0$. □

Unfortunately the most prevalent usage of these order classes incorporates a deliberate abuse of mathematical notation for the sake of convenience. While it is straight forward to express equivalent order by writing $f \in \Theta(g)$, it more often expressed by writing $f = \Theta(g)$ (motivated from the spoken "f is order g" which is a natural albeit lazy verbalization of the mathematically precise notation).

It is the authors contention that this surfeit of definitions of order classes early in the study of algorithm analysis serves mostly to confuse students with an otherwise easily grasped concept.

2.5 Bounding Techniques

It is interesting to note that lavish application of coarse bounds can be quite a useful technique in establishing order relationships. This is due in no small part to the fact that any interval bounded away from zero and ∞ (or any non-zero real number in the case the limit exists) is sufficient to establish equivalent order. The actual numbers involved in the limits are of no importance as long as they are non-zero and finite. We will now make use of bounding arguments to establish order relationships that might otherwise appear difficult to prove. In most cases we will establish coarse pointwise upper bounds valid only for sufficiently large n.

Example 2.11. $3n^5 + n^3 + 4n^2 + 1 \simeq n^5$

Since $n^5 < 3n^5 + n^3 + 4n^2 + 1$ *we have* $n^5 \preceq 3n^5 + n^3 + 4n^2 + 1$. *Also,*

$$\begin{aligned} 3n^5 + n^3 + 4n^2 + 1 &\leq 3n^5 + n^5 + 4n^5 + n^5, \qquad \text{(for } n \geq 1\text{)}, \\ &= 9n^5 \end{aligned}$$

thus $3n^5 + n^3 + 4n^2 + 1 \simeq n^5$ *by Theorems 2.3.1 and 2.3.1.*

Example 2.12. $\sqrt{4n^2 + 1} \simeq n$

First,

$$\begin{aligned} \sqrt{4n^2 + 1} &\leq \sqrt{4n^2 + 5n^2}, \qquad \text{(for } 5n^2 \geq 1, \text{ or } n \geq \tfrac{1}{\sqrt{5}}\text{)} \\ &= 3n \end{aligned}$$

By Theorem 2.3.1, $\sqrt{4n^2 + 1} \preceq n$. *Note also that* $n = \sqrt{n^2} < \sqrt{4n^2 + 1}$, *giving* $n \preceq \sqrt{4n^2 + 1}$ *hence* $\sqrt{4n^2 + 1} \simeq n$ *by Theorem 2.3.1.*

Alternatively, one could have bounded $\sqrt{4n^2 + 1}$ by $\sqrt{5}n$ for $n \geq 1$.

Example 2.13. $\log_b(n^2 + 3n + 1) \preceq \log_b(n)$

For $n > 1$,

$$\begin{aligned} \log_b(n^2 + 3n + 1) &\leq \log_b(n^2 + 3n^2 + n^2) \\ &= \log_b(5n^2) \\ &= 2\log_b(n) + \log_b(5) \end{aligned}$$

In the next example a sum will be bounded below by throwing away half of its terms. As the bound obtained will be of the same order as the upper bound, the functions order will be established. The function will occur a few more times in the sequel.

Example 2.14. $\log_b(n!) \simeq n\log_b(n)$

$$\begin{aligned}
\log_b(n!) &= \sum_{i=1}^{n} \log_b(i) \\
&\leq \sum_{i=1}^{n} \log_b(n) \\
&= n\log_b(n) \quad ,\textit{and} \\
\log_b(n!) &\geq \sum_{i=\frac{n}{2}}^{n} \log_b(i) \\
&\geq \sum_{i=\frac{n}{2}}^{n} \log_b(\frac{n}{2}) \\
&= \frac{n}{2}\log_b(\frac{n}{2})
\end{aligned}$$

therefore,

$$\begin{aligned}
1 \leq \lim_{n\to\infty} \frac{n\log_b(n)}{\log_b(n!)} &\leq \lim_{n\to\infty} \frac{n\log_b(n)}{\frac{n}{2}\log_b(\frac{n}{2})} \\
&= 2\lim_{n\to\infty} \frac{\log_b(n)}{\log_b(n) - \log_b(2))} \\
&= 2\lim_{n\to\infty} \frac{1}{1 - \frac{\log_b(2)}{\log_b(n)}} \\
&= 2
\end{aligned}$$

Thus

$$\lim_{n\to\infty} \frac{n\log_b(n)}{\log_b(n!)} \in [1,2] \subset (0,\infty).$$

Note that there is a compelling reason to throw away half of the terms above, for otherwise, one obtains a lower bound of $n \cdot 0$, leaving a large range of uncertainty for the order of $n\log_b(n)$.

2.6 Delay and Invariance

It is possible to think of using the values of a given function f for higher values of n in an attempt to get a higher order function; say $g(n) = f(n + 100)$. From one point of view these seem to be the same function and the possibility of a difference in order seems silly. In fact for most of the functions considered thus far there is no difference.

Theorem 2.6.1. *Let $f \in \{log(n), n^k, a^n\}$, $i > 0$, and $g(n) = f(n + i)$. Then $g \simeq f$.*

Proof.

$$\lim_{n\to\infty} \frac{log(n)}{log(n+i)} = \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{n+i}} = \lim_{n\to\infty} \frac{n+i}{n} = 1 \in (0, \infty)$$

$$\lim_{n\to\infty} \frac{n^k}{(n+i)^k} = \left[\lim_{n\to\infty} \frac{n}{n+i}\right]^k = 1 \in (0, \infty)$$

$$\lim_{n\to\infty} \frac{a^n}{a^{n+i}} = \lim_{n\to\infty} \frac{1}{a^i} = \frac{1}{a^i} \in (0, \infty)$$

□

It is however simple to find examples of functions for which such delays make a difference in order.

Example 2.15. $n! \prec (n + 1)!$

$$\lim_{n\to\infty} \frac{n!}{(n+1)!} = \lim_{n\to\infty} \frac{1}{n+1} = 0$$

Example 2.16. $n^n \prec (n + 1)^{n+1}$

$$\begin{aligned}\lim_{n\to\infty} \frac{n^n}{(n+1)^{n+1}} &= \lim_{n\to\infty} \left[\frac{1}{n+1}\left(\frac{n}{n+1}\right)^n\right] \\ &= \left[\lim_{n\to\infty} \frac{1}{n+1}\right]\left[\lim_{n\to\infty} \left(\frac{n}{n+1}\right)^n\right] \\ &= 0\left(\frac{1}{e}\right)\end{aligned}$$

2.7 Order Hierarchy

With tools in hand we now proceed to establish the relationships among the many functions we are likely to encounter. We begin with the simpler functions and proceed to refine a hierarchy by adding functions with increasingly difficult comparisons. For reasons which will later become clear we will from this point forward deal only with monotonic nondecreasing nonnegative functions. It certainly seems reasonable that algorithm time complexities should yield only such functions (when was the last time that you ran a program and it transported you back in time).

It should be observed that the bounded functions, by virtue of the fact that they do not grow at all, belong at the bottom of the order hierarchy. Can we select a simplest representative to stand for all such functions? That is if f is any bounded nonnegative nondecreasing (and nonzero[1]) function, for what simple function g can we write $f \simeq g$? Let B be an upper bound for the values of f. Then

$$0 < \lim_{n\to\infty} \frac{f(n)}{1} < B,$$

therefore $f \simeq 1$. To justify our claim that these belong at the bottom of the order hierarchy, consider any nonnegative unbounded function f. Then we must have

$$\lim_{n\to\infty} \frac{1}{f(n)} = 0,$$

otherwise f would be bounded. Thus we establish $1 \prec f$.

We have already shown that the exponential, polynomial, and log subhierarchies obey (except in the degenerate case for logs) a preservation of order suggested by the powers and bases involved. Our preliminary examples (ref what) suggest $log_b(n) \prec n^p \prec a^n$. We now establish these in general.

[1]Zero functions have no place as time complexity functions. No operation, not even a null instruction executes in zero time.

2.7.1 Polylogs and Powers

We have already seen from Theorem 2.2.4 that the polylog functions form an order subhierarchy. This entire subhierarchy is bounded by the polynomial subhierarchy.

Theorem 2.7.1. $\log_b^k(n) \prec n^p$, *for* $k \geq 1, b > 1, p > 0$

Proof. If k is an integer, then

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_b^k(n)}{n^p} &\overset{L'}{=} \lim_{n\to\infty} \frac{k\log_b^{k-1}(n)\frac{1}{n\ln(a)}}{pn^{p-1}} \\
&= \frac{k}{p\ln(a)} \lim_{n\to\infty} \frac{\log_b^{k-1}(n)}{n^p} \\
&\overset{L'}{=} \frac{k}{p\ln(a)} \lim_{n\to\infty} \frac{(k-1)\log_b^{k-2}(n)\frac{1}{n\ln(a)}}{pn^{p-1}} \\
&= \frac{k(k-1)}{p^2\ln^2(a)} \lim_{n\to\infty} \frac{\log_b^{k-2}(n)}{n^p} \\
&\vdots \\
&= \frac{k(k-1)(k-2)\dots(3)(2)}{p^{k-1}\ln^{k-1}(a)} \lim_{n\to\infty} \frac{\log_b(n)}{n^p} \\
&\overset{L'}{=} \frac{k(k-1)(k-2)\dots(3)(2)}{p^{k-1}\ln^{k-1}(a)} \lim_{n\to\infty} \frac{\frac{1}{n\ln(a)}}{pn^{p-1}} \\
&= \frac{k!}{p^k\ln^k(a)} \lim_{n\to\infty} \frac{1}{n^p} \\
&= 0.
\end{aligned}$$

If k is not an integer, let $l = \lceil k \rceil$. Then

$$\begin{aligned}
log_b^k(n) &\prec log_b^l(n) \\
&\prec n^p.
\end{aligned}$$

□

Misleading Experiments

The asymptotic behavior of an order comparison is not always evident from simple graphic representations. As an example we consider how the functions $\log_2^3(n)$ and $n^{\frac{1}{2}}$ compare. Perhaps simple numeric comparison will yield some insight:

The dominance of the log function over the square root seems apparent. In terms of order one might naturally conclude that $n^{\frac{1}{2}} \preceq \log_2^3(n)$ and therefore expect that the limit

$$\lim_{n\to\infty} \frac{n^{\frac{1}{2}}}{\log_2^3(n)}$$

Table 2.2: compared values for $\log_2^3(n)$ and $n^{\frac{1}{2}}$

n	$\log_2(n)$	$\log_2^3(n)$	$n^{\frac{1}{2}}$
2^2	2	8	2
2^4	4	64	4
2^6	6	216	8
2^8	8	512	16
2^{10}	10	1000	32

be zero. Using successive applications of L'hopitals rule for limits of quotients gives:

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^{\frac{1}{2}}}{\log_2^3(n)} &= \lim_{n\to\infty} \frac{\frac{1}{2}n^{-\frac{1}{2}}}{3\log_2^2(n)\frac{1}{n\ln(2)}} \\
&= \frac{1}{6\ln(2)} \lim_{n\to\infty} \frac{n^{\frac{1}{2}}}{\log_2^2(n)} \\
&= \frac{1}{6\ln(2)} \lim_{n\to\infty} \frac{\frac{1}{2}n^{-\frac{1}{2}}}{2\log_2(n)\frac{1}{n\ln(2)}} \\
&= \frac{1}{24\ln^2(2)} \lim_{n\to\infty} \frac{n^{\frac{1}{2}}}{\log_2(n)} \\
&= \frac{1}{24\ln^2(2)} \lim_{n\to\infty} \frac{\frac{1}{2}n^{-\frac{1}{2}}}{\frac{1}{n\ln(2)}} \\
&= \frac{1}{48\ln^3(2)} \lim_{n\to\infty} n^{\frac{1}{2}} \\
&= \infty
\end{aligned}$$

Certainly this is not evident from extrapolation of the data points considered above. In fact, the asymptotic relationship between the two functions is opposite of that indicated by the preliminary graph. Look at the relative values of these functions for much larger values of n:
Somewhere between $n = 2^{29}$ and $n = 2^{30}$ the log expression has been overtaken by the root. This is an example of asymptotic behavior that takes a very large problem size to be realized.

Table 2.3: compared values for $\log_2^3(n)$ and $n^{\frac{1}{2}}$

n	$\log_2(n)$	$\log_2^3(n)$	$n^{\frac{1}{2}}$
2^{16}	16	4096	256
2^{20}	20	8000	1024
2^{24}	24	13824	4096
2^{28}	28	21952	16384
2^{29}	29	24389	23170
2^{30}	30	27000	32768
2^{40}	40	64000	1048576

2.7.2 Iterated Log Subhierarchy

We next introduce a subhierarchy below the polylog hierarchy consisting of compositions of logs. Denote the m-fold application of a function $f(n)$ by $f^{[m]}(n)$, that is,

$$f^{[m]}(n) = f(f(f(...f(n)...))) \quad \text{(for } m-1 \text{ compositions of } f\text{)}$$

For example, if $f(n) = n^2 + 1$, then

$$\begin{aligned}
f^{[1]}(n) &= f(n) \\
&= n^2 + 1, \\
f^{[2]}(n) &= f(f(n)) \\
&= f(n^2 + 1) \\
&= (n^2 + 1)^2 + 1 \\
&= n^4 + 2n^2 + 2 \\
f^{[3]}(n) &= f(f(f(n))) \\
&= f(f^[2](n)) \\
&= f((n^2 + 1)^2 + 1) \\
&= ((n^2 + 1)^2 + 1)^2 + 1 \\
&= n^8 + 4n^6 + 8n^4 + 8n^2 + 5
\end{aligned}$$

More precisely, define $f^{[m]}$ for $m > 0$ by

Definition 2.7.1.

$$f^{[m]}(n) = \begin{cases} f(n) & ,m = 1 \\ f(f^{[m-1]}(n)) & ,m > 1 \end{cases}$$

Functional composition represented by $f^{[m]}$ can be used to descend the order hierarchy below the *log* functions.

Theorem 2.7.2. $\log_b^{[k+i]}(n) \prec \log_b^{[k]}(n)$*, for* $k, i \geq 1, b > 1$

Proof. It suffices to consider $i = 1$ and apply transitivity.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_b^{[k+1]}(n)}{\log_b^{[k]}(n)} &\stackrel{L'}{=} \lim_{n\to\infty} \frac{\frac{1}{\log_b^{[k]}(n)} \frac{d}{dn}\left[\log_b^{[k]}(n)\right]}{\frac{d}{dn}\left[\log_b^{[k]}(n)\right]} \\
&= \lim_{n\to\infty} \frac{1}{\log_b^{[k]}(n)} \\
&= 0.
\end{aligned}$$

□

Furthermore, the entire iterated log subhierarchy is dominated by by the polylog hierarchy in the simplest way, that is, any iterated log is dominated by any log.

Theorem 2.7.3. $\log_b^{[i]}(n) \prec \log_b^p(n)$, *for* $i, b > 1$.

Proof. Using the last theorem it suffices to demonstrate that $\log_b^{[2]}(n) \prec \log_b(n)$. In fact,

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_b^{[2]}(n)}{\log_b(n)} &\overset{L'}{=} \lim_{n\to\infty} \frac{\frac{1}{n\log_b(n)\ln^2(b)}}{\frac{1}{n\ln(b)}} \\
&= \lim_{n\to\infty} \frac{1}{\log_b(n)\ln(b)} \\
&= 0
\end{aligned}$$

□

Taking things to extremes we define the terminal log function, $\log^*$ function as follows:

Definition 2.7.2. $\log_b^*(n) = \min\{k \geq 0 : \log_b^{[k]}(n) \leq 1\}$

It should be clear that for any k that $\log_b^*(n) \leq \log_b^{[k]}(n)$ eventually. Thus

$$\log_b^*(n) \leq \log_b^{[k+1]}(n) \prec \log_b^{[k]}(n),$$

demonstrating that $\log_b^*(n) \prec \log_b^{[k]}(n)$ for any k.

2.7.3 Polynomials and Exponentials

The relationship for polynomials and exponentials is established much the same way as that for polylogs and polynomials.

Theorem 2.7.4. *$n^p \prec a^n$, for $p > 0, b > 1$.*

Proof. If p is an integer, then

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^p}{b^n} &\stackrel{L'}{=} \lim_{n\to\infty} \frac{pn^{p-1}}{b^n \ln(b)} \\
&= \frac{p}{\ln(b)} \lim_{n\to\infty} \frac{n^{p-1}}{b^n} \\
&\stackrel{L'}{=} \frac{p}{\ln(b)} \lim_{n\to\infty} \frac{(p-1)n^{p-2}}{b^n \ln(b)} \\
&= \frac{p(p-1)}{\ln^2(b)} \lim_{n\to\infty} \frac{n^{p-2}}{b^n} \\
&\vdots \\
&= \frac{p(p-1)(p-2)\dots(3)(2)}{\ln^{p-1}(b)} \lim_{n\to\infty} \frac{n}{b^n} \\
&= \frac{p!}{\ln^p(b)} \lim_{n\to\infty} \frac{1}{b^n} \\
&= 0
\end{aligned}$$

If p is not an integer, let $q = \lceil k \rceil$. Then $n^p \prec n^q \prec b^n$. □

Finally we put a collective cap on all of the exponentials.

Theorem 2.7.5. *$a^n \prec n^n$, for $a > 0$.*

Proof.

$$\begin{aligned}
\lim_{n\to\infty} \frac{a^n}{n^n} &= \lim_{n\to\infty} \left(\frac{a}{n}\right)^n \\
&= 0
\end{aligned}$$

since for $n > A = \lceil a \rceil + 1$, one has $\frac{a}{n} < \frac{a}{A} < 1$ and $\left(\frac{a}{A}\right)^n \to 0$. □

2.7.4 Exotic Order Comparisons

Here we collect, mostly for curiosity's sake, several order comparisons most of which are not too likely to arrise in practice but none the less do provide a greater appreciation for the span of the order hierarchy.
First is a comparison demonstrating an upper bound for all exponentials.

Theorem 2.7.6. $a^n \prec n!$.

Proof. Let $N = \lceil a \rceil, r = \frac{a}{N+1}$. Then $r < 1$ and

$$\begin{aligned}\lim_{n\to\infty} \frac{a^n}{n!} &= \frac{a^N}{N!} \lim_{n\to\infty} \frac{a^{n-N}}{(N+1)(N+2)\dots(n-1)n} \\ &\leq \frac{a^N}{N!} \lim_{n\to\infty} r^{n-N} \\ &= \frac{a^N}{N!r^N} \lim_{n\to\infty} r^n \\ &= 0\end{aligned}$$

□

Here are two more for very high orders.

Theorem 2.7.7. $n! \prec \log^n(n)$

Proof. By Example 2.14 and Theorem 2.7.3, $\lim_{n\to\infty} \left(\frac{\log^{[2]}(n)}{\log(n)} - \frac{\log(n!)}{n\log(n)} \right) = c < 0$ thus $\lim_{n\to\infty} n\log^{[2]}(n) - \log(n!) = \lim_{n\to\infty} n\log(n) \lim_{n\to\infty} \left(\frac{\log^{[2]}(n)}{\log(n)} - \frac{\log(n!)}{n\log(n)} \right) = -\infty$. Then

$$\begin{aligned}\lim_{n\to\infty} \frac{\log^n(n)}{n!} &= \lim_{n\to\infty} a^{\log_a\left(\frac{\log^n(n)}{n!}\right)} \\ &= \lim_{n\to\infty} a^{\log_a(\log_a(n)) - \log_a(n!)} \\ &= \lim_{n\to\infty} a^{n\log_a^{[2]}(n) - \log_a(n!)} \\ &= 0\end{aligned}$$

□

Theorem 2.7.8. *$log^n(n) \prec n^n$.*

Proof. Exercise. [Examine the proof of Theorem 2.7.7.] □

We now show that there is quite a bit of room between the polynomials and the exponentials

Theorem 2.7.9. $n^p \prec \lceil \log \rceil!(n)$.

Proof. Let $b = a^p, k = \lceil \log_a(n) \rceil$. Since $b^k \prec k!$ by Theorem 2.7.6 we have $a^{p\lceil \log_a(n)\rceil} \prec \lceil \log_a(n) \rceil!(n)$, but $n^p = a^{\log_a(n^p)} = a^{p\log_a(n)} \leq a^{p\lceil \log_a(n)\rceil}$. □

Theorem 2.7.10. $\lceil\log\rceil!(n) \prec \log^{\log(n)}(n)$.

Proof. Exercise. [Let $k = \log(n)$ and use transitivity of order.] $\square$

Theorem 2.7.11. $\log^{\log(n)}(n) \prec n^{\log(n)}$.

Proof.

$$\begin{aligned}\lim_{n\to\infty} \frac{\log_a^{\log_a(n)}(n)}{n^{\log_a(n)}} &= \lim_{n\to\infty} a^{\log_a\left(\frac{\log_a(n)}{n}\right)^{\log_a(n)}} \\ &= \lim_{n\to\infty} a^{\log_a(n)(\log_a^{[2]} - \log_a(n))} \\ &= 0\end{aligned}$$

$\square$

Theorem 2.7.12. *For $a \geq b > 1, n^{log_b(n)} \prec a^n$.*

Proof. Using Theorem 2.2.2 it is enough to demonstrate that $n^{log_b(n)} \prec b^n$. By Theorem 2.7.1 we have

$$\begin{aligned}\lim_{n\to\infty} \frac{n^{\log_b(n)}}{b^n} &= \lim_{n\to\infty} \frac{b^{\log_b\left(n^{\log_b(n)}\right)}}{b^n} \\ &= \lim_{n\to\infty} b^{\log_b^2(n)-n} \\ &= 0\end{aligned}$$

$\square$

2.7.5 Ackerman's Function

Functional composition can also be used to climb the order hierarchy of additive, multiplicative, and exponential functions and beyond. The general idea is to start with an additive first order function and use the composition process to obtain a multiplicative function of first order. Then apply the composition process to this multiplicative function to obtain an exponential function. The same process can be applied further to give whatever lies beyond the exponential function, and there's no reason to stop there.

Define a sequence of functions beginning with $f_0(n) = n + 2$. Then

$$f_0^{[m-1]}(n) = n + 2(m - 1),$$

and in particular,

$$f_0^{[m-1]}(2) = 2m.$$

Next define $f_1(n) = 2n$, so that

$$f_1^{[m-1]}(n) = n2^{m-1},$$

and

$$f_1^{[m-1]}(2) = 2^m.$$

Then for $f_2(n) = 2^n$, $f_2^{[m-1]}(2)$ is a cascade of m nested exponentiations beginning with 2, and this is defined to be f_3. One may proceed indefinitely in this manner to define a sequence of functions $f_0, f_1, f_2, f_3, f_4...$, where $f_m(n) = f_{m-1}^{[n-1]}(2)$. Now define Ackermans function (a lesser relative actually) by

$$a(n) = f_n(n).$$

To get some idea of the growth rate of $a(n)$, consider the first few values for n:

$$
\begin{aligned}
a(0) &= f_0(0) = 2, \\
a(1) &= f_1(1) = 2, \\
a(2) &= f_2(2) = 2^2 = 4, \\
a(3) &= f_3(3) = 2^{2^2} = 16, \\
a(4) &= f_4(4) = f_3^{[3]}(2) \\
&= f_3(f_3(f_3(2))) \\
&= f_3(f_3(2^2)) \\
&= f_3(f_3(4)) \\
&= f_3(2^{2^{2^2}}) \\
&= f_3(65536) \\
&= 2^{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}} \quad \text{(64K levels of exponentiation)} \\
&>> 10^{100000}
\end{aligned}
$$

If written without an exponent, assuming you can write 10 zeros per inch, this number is a 1 followed by more than a tenth of a mile of zeros. Would you care to estimate $a(5)$? It is generally conceded that $a(4)$ is greater than the number of particles in the universe.

 ©2003 Steven Louis Davis

Table 2.4: Order Hierarchy

function	description
$a(n)$	Ackerman's simple cousin
n^n	
$\log^n(n)$	
$n!$	factorial
$a^n \quad (a > 1)$	exponential
$n^{\log(n)}$	
$\log^{\log(n)}(n)$	
$\lceil\log\rceil!(n)$	factorial log
$n^p \quad (p > 0)$	polynomial
$\log^p(n) \quad (p > 0)$	polylog
$\log^{[i]}(n)$	iterated log composition
$\log^*(n)$	terminal log
1	constant

2.7.6 Collected Comparisons

Table 2.4 is used to summarize several order comparisons.
Concerning the table the following remarks should be made. In the cases of polynomial and polylog functions, these labels would perhaps better be called colloquialisms as more precise labels would be Power and Powerlog. Unlike the table, the hierarchy is in fact infinite; between any two f, g with $f \prec g$ there is a function h with $f \prec h \prec g$.

2.8 Existence of the Laplace Transform

Perhaps the most recognizable appearance of order theory in elementary mathematics occurs in the existence criteria for the Laplace Transform. Given a continuous function $f(t)$ defined on $[0, \infty)$ the Laplace Transform of f is defined as the function

$$F(s) = \int_0^\infty e^{-st} f(t)\, \mathrm{d}t$$

for $s > 0$. A sufficient condition for the existence of the integral is that $f(t) \preceq e^{at}$ for $a < s$.

Theorem 2.8.1. *If f is continuous on $[0, \infty)$ and $f(t) \preceq e^{at}$ for $a < s$, then $e^{-st} f(t)$ is integrable on $[0, \infty)$ and*

$$\int_0^\infty e^{-st} f(t)\, \mathrm{d}t = \lim_{n\to\infty} \int_0^n e^{-st} f(t)\, \mathrm{d}t.$$

Proof. First, the definite integral is defined owing to the continuity and therefore boundedness of $e^{-st} f(t)$ on any bounded interval. Choose $c > 0$ and $t_0 \in [0, \infty)$ such that $|f(t)| \leq ce^{at}$ for $t \geq t_0$. Let $g(t) = e^{-st} f(t)$, and $g_n(t) = g(t) I_{[0,n]}(t)$, where $I_{[0,n]}(t)$ is the indicator function for the set $[0, n]$. Then g_n is integrable on $[0, \infty)$ and $g_n \to g$ everywhere. The sequence is cauchy in $L_1[0, \infty)$, for if $n > m \geq t_0$, we have

$$\begin{aligned} \|g_n - g_m\|_1 &= \int_m^n e^{-st} |f(t)|\, \mathrm{d}t \\ &\leq c \int_m^n e^{t(a-s)}\, \mathrm{d}t \\ &= \frac{c}{a-s} e^{t(a-s)} \Big|_m^n \\ &= \frac{c}{a-s} \left[e^{n(a-s)} - e^{m(a-s)} \right] \longrightarrow 0, \quad \text{as } m, n \to \infty. \end{aligned}$$

By Chebychev's and Reisz's theorems, a subsequence of (g_n) must converge almost everywhere to the L_1 limit, and since $g_n \to g$ everywhere, g must be the limit, that is, g is integrable. Moreover,

$$\begin{aligned} \int_0^\infty e^{-st} f(t)\, \mathrm{d}t &= \lim_{n\to\infty} \int_0^\infty g_n(t)\, \mathrm{d}t \\ &= \lim_{n\to\infty} \int_0^n e^{-st} f(t)\, \mathrm{d}t \end{aligned}$$

□

Since the assumption $f(t) \preceq e^{at}$ for $a < s$ implies $f(t) \prec e^{st}$ it might seem natural to suspect that the condition $f(t) \prec e^{st}$ is sufficient for the existence

of the transform. This however is not the case as the function $f(t) = e^{st}h(t)$ demonstrates, where

$$h(t) = \begin{cases} 1 & ,0 \leq t < 1 \\ \frac{1}{t} & ,1 \leq t. \end{cases}$$

Chapter Exercises

Order Comparison

E 2.1. *Use a language L other than assembly or C to code the fib1 algorithm and determine $X_{fib1,L}(n)$.*

E 2.2. *Compare both of the exponentiation algorithms, exp1 and exp2 on the same graph.*

E 2.3. *Find at least three reasons that timing algorithm execution with a stop-watch is not a fair measure of its time complexity.*

E 2.4. *Can time complexity analysis be left to a program? That is, can a program be written which takes algorithms as inputs and whose output is the time complexity function of the input algorithm?*

E 2.5. *Find another problem having at least two solutions with different time complexity functions.*

Equivalent Order

For each of the following pairs of functions f, g, determine whether $f \simeq g$ or $f \not\simeq g$.

E 2.6. $f(n) = (3n+1)^3, g(n) = n^3$

E 2.7. $f(n) = n^5 + 2n^2 + 1, g(n) = n^4$

E 2.8. $f(n) = \log_2(n), g(n) = n$

E 2.9. $f(n) = \log_2(1 + 3^n), g(n) = n$

E 2.10. $f(n) = \log_2^2(n), g(n) = \log_2(n)$

E 2.11. $f(n) = \ln(3n^4 + 5n^2 + 2), g(n) = \ln(n^2) + \ln(n)$

E 2.12. $f(n) = 2^n, g(n) = 5^n$

E 2.13. $f(n) = e^{n^2+2n+1}, g(n) = e^{n^2+1}$

E 2.14. $f(n) = \sqrt{2n+1}, g(n) = 2\sqrt{n}$

E 2.15. $f(n) = 2x\sin(x) + 3, g(n) = x\cos(3x) + 1$

E 2.16. *Prove Theorem 2.1.2.*

Inferior Order

For each of the following pairs of functions f, g, demonstrate that $f \prec g$.

E 2.17. $f(n) = 3n, g(n) = n^2 + 3$

E 2.18. $f(n) = n^2 + 4n + 3, g(n) = n^5$

E 2.19. $f(n) = (n+2)^3, g(n) = (n+2)^4$

E 2.20. $f(n) = \sqrt{n}, g(n) = n$

E 2.21. $f(n) = \frac{100}{n}, g(n) = \frac{n}{100}$

E 2.22. $f(n) = n \log_2(n), g(n) = n^2$

E 2.23. $f(n) = \log_2(n^7), g(n) = n$

E 2.24. $f(n) = n^2 \log_2(n^3), g(n) = n^3$

E 2.25. $f(n) = n2^n, g(n) = 3^n$

E 2.26. $f(n) = n^2 2^n, g(n) = 3^n$

Non-Strict Order

Establish the indicated relationship.

E 2.27. $n^2 + 2n \simeq n^2$

E 2.28. $n^3 + 3\sqrt{n^4 + 2n^3 + n^2 + 3} \preceq n^3$

E 2.29. $2n^2 + 4n \log_2(n) \preceq n^2$

E 2.30. $\sqrt[3]{n^3 + 3n^2 + 3n} \preceq n$

E 2.31. $(\sqrt{2n} + 1)^4 \preceq n^2$

E 2.32. $\log_2\left(\frac{(2n)!}{(n-1)!}\right) \succeq n \log_2(n)$

Polylogs and Powers

For the functions $f(n) = \log_2^2(n), g(n) = n^{\frac{1}{3}}$ do the following:

E 2.33. *Graph both functions together on the same graph.*

E 2.34. *Show that $f(n) \prec g(n)$.*

E 2.35. *Find the approximate crossing point of f and g.*

E 2.36. *Find another pair of functions with this kind of noncharacteristic early behavior.*

Iterated Logs

E 2.37. *Given that* $n = 1, 2, 4, 8, 16, 32, ...$ *are convenient values to use in plotting the* $\log_2(n)$ *function, determine convenient values to use when plotting the function* $f(n) = \log_2^{\frac{1}{2}}(n-1)$.

E 2.38. *How big must* n *be for* $\log_2^*(n) = 5$*?*

Polynomials and Exponentials

E 2.39. *Find a function* $f(n)$ *such that* $n^p \prec f(n) \prec n^{p+1}$.

E 2.40. *Find a function* $f(n)$ *such that* $a^n \prec f(n) \prec b^n$ *for any* $b > a$.

E 2.41. *Prove Theorem 2.7.8.*

E 2.42. *Prove Theorem 2.7.10.*

E 2.43. *Find a function that grows more slowly than* $\log_2^*(n)$.

 ©2003 Steven Louis Davis

II

Algorithm Analysis

Chapter 3

Time Complexity Analysis

As applied to algorithms, order analysis is concerned with time and memory requirements that are characteristic of the algorithm. Functions which characterize these requirements are called time or memory complexity functions. Their analysis is called time or memory complexity analysis. More formally, given an algorithm A which solves instances of a problem, a time complexity function for A is a non-negative function $T_A(n)$ associating the time required for A to solve a problem instance of size n. Memory complexity is similarly defined. Though it is traditional to characterize problem instances in terms of size alone, this does not always yield a well defined notion of complexity. Those algorithms behaving differently for different problem instances of the same size must be analyzed under restrictions capable of differentiating between those cases yielding different results. To clarify the situation, consider the problem of array searching. It is clear that the size of the array is not the only consideration in determining the amount of time required to find a particular element. The element itself is of primary importance since it could be found early or late in the search. We refer to such problems as *data sensitive*. For such problems the time complexity depends not only on the size of the problem instance but also on the distribution of data within the instance. If the distribution D is known the time complexity $T_D(n)$ is the desired measure of the algorithm. Because such information is usually unavailable we use three abstract distributions: those distributions yielding the *best*, *worst*, and *average* behavior of the algorithm. For data sensitive analysis we will use the symbols $B(n), W(n)$, and $A(n)$ to denote best, worst, and average case analysis. When there is no data sensitivity we simply use $T(n)$.

3.1 Average Case Time Complexity Analysis

For data sensitive problems the time complexity $T_(n)$ can be considered to be a random variable governed by the data distribution of the problem instance. Accordingly, its average value is simply the expectation over its range of values.

$$E(T) = \sum_{i=B(n)}^{W(n)} iP(T = i) \tag{3.1}$$

We now analyze the linear and binary search algorithms. It should be noted here that no complete analysis can be given here unless all possible array distributions were to be considered. We will satisfy ourselves with uniformly distributed keys, that is, that in an array of n keys the probability that a key k of the array is in any particular position p is $P(k = a[p]) = \frac{1}{n}$.

Example 3.1 (Average Case Analysis of Linear Search)**.**

For linear search we characterize the possible running times as the possible array positions in which the search key may be found. Thus the best case is 1, the worst n, and the average is computed, assuming the uniform distribution, from 3.1 as:

$$\begin{aligned}
A(n) &= E(T) \\
&= \sum_{i=B(n)}^{W(n)} iP(T = i) \\
&= \sum_{i=1}^{n} i\frac{1}{n} \\
&= \frac{1}{n}\sum_{i=1}^{n} i \\
&= \frac{1}{n}\frac{n(n+1)}{2} \\
&= \frac{n+1}{2}
\end{aligned}$$

Example 3.2 (Average Case Analysis of Binary Search)**.**

For binary search the situation seems much the same except that the worst case is now $\log_2(n)$ and the probability of finding a given key at time i is $P(T = i) = \frac{2^{i-1}}{n}$. Then

```
int bin_search(int a[],int left,int right,int key){
if(left>right)
   return 0;
int midpoint=(left+right)/2;
if(key<a[midpoint])
   return bin_search(left,midpoint-1);
else if(key>a[midpoint])
   return bin_search(midpoint+1,right);
else
   return midpoint;
}
```

Figure 3.1: Binary Search Algorithm

$$
\begin{aligned}
A(n) &= E(T) \\
&= \sum_{i=B(n)}^{W(n)} iP(T=i) \\
&= \sum_{i=1}^{\log_2(n)} i\frac{2^{i-1}}{n} \\
&= \frac{1}{n}\sum_{i=1}^{\log_2(n)} i2^{i-1} \\
&= \frac{1}{2n}\sum_{i=1}^{\log_2(n)} i2^{i} \\
&= \frac{1}{2n}\left[(\log_2(n)-1)2^{\log_2(n)+1}+2\right] \\
&= \frac{1}{2n}[(\log_2(n)-1)2n+2] \\
&= \frac{1}{n}[(\log_2(n)-1)n+1] \\
&= \log_2(n)-1+\frac{1}{n}
\end{aligned}
$$

Note how easy it would be to make the mistake of using $P(T=i)=\frac{1}{n}$ for the uniform distribution as before. The algorithm itself changes the characterization of the the probability since there are many possible midpoints that could be examined at a given iteration. Using the incorrect probability would yield

$$\begin{aligned} A(n) &= E(T) \\ &= \sum_{i=B(n)}^{W(n)} iP(T=i) \\ &= \sum_{i=1}^{\log_2(n)} i\frac{1}{n} \\ &= \frac{1}{n}\sum_{i=1}^{\log_2(n)} i \\ &= \frac{1}{n}\frac{\log_2(n)(\log_2(n)+1)}{2} \end{aligned}$$

Thus $A(n) \simeq \frac{\log^2(n)}{n}$. Comparing orders we see immediately that $\lim_{n\to\infty} A(n) = 0$ which would suggest that the best first step for finding a key is to throw ones array into a giant heap of data before beginning the search!

Chapter 4

Greedy Selection

One of the simplest types of algorithm iteratively assembles a solution from a set of components with associated values, taking a single component in each iteration with the choice based on optimal component value. Often the optimal value will be a maximal value prompting the term 'greedy' associated with the algorithm type.

4.1 Single Source Shortest Paths

We first consider the Single Source Shortest Paths problem and investigate a greedy solution attributed to Dijkstra and important in several applications, notably that of computing routing information for routed networks. The problem, given a connected weighted graph of n vertices with one vertex v_0 identified as the source vertex, is to find a spanning tree with the property that each path from a vertex to the source has minimal length for that vertex. Attacking this problem with an exhaustive enumeration of all possible spanning trees is costly and voraciously consumes processor cycles even for modest numbers of vertices. The worst case for n vertices is a completely connected graph of $\binom{n}{2}$ edges. For each of the $n-1$ non source vertices $\{v_i : i = 1, 2, \ldots, n-1\}$ we must enumerate all paths between v_0 and v_i using the remaining $n-2$ vertices. Since any subset of these $n-2$ vertices can contribute paths, and every permutation of such a subset yields a different path we have:

$$\begin{aligned} W(n) &= (n-1)\sum_{k=0}^{n-2}\binom{n-2}{k}k! \\ &= (n-1)(n-2)!\sum_{k=0}^{n-2}\frac{1}{n-2-k} \\ &= (n-1)!\sum_{j=0}^{n-2}\frac{1}{j}, \qquad\qquad (j = n-2-k) \end{aligned}$$

Using the fact that $\sum_{j=0}^{n-2} \frac{1}{j!} \in [1, e)$, we have $W(n) \simeq (n-1)!$.

4.1.1 Dijkstra's Algorithm

Dijkstra's algorithm (Figure 4.1) grows a tree of internal vertices, initialized with a single vertex called the source, and iteratively incorporates from the set of external vertices that vertex whose path to the source vertex, using only internal vertices, is minimal. This minimal choice which is optimal for the problem at hand is the greedy selection. When all vertices have become internal, the resulting spanning tree has the property that for each non source vertex, its path to the source has minimal weight. This is not necessarily a spanning tree of minimal weight.

```
void dijkstra(int n,int edges[][],int source){
int dist[n+1],int link[n+1];
_initialize(n,edges,source,dist,link);
for(int i=1;i<=n;i++){
   next_internal=_min(n,edges,source,dist);
   _update(n,edges,dist,link,next_internal);}}

int _initialize(int n,int e[][],int s,int d[],int l[]){
for(int i=1;i<=n;i++){
   d[i]=e[s][i];
   l[i]=s;}}

int _min(int n,int e[][],int s,int d[]){
int min=HUGE_VAL,mindex=-1;
for(int i=1;i<=n;i++)
   if((i!=s)&&d[i]&&d[i]<min){
      mindex=i;
      min=d[i];}
return mindex;}

void _update(int n,int e[][],int d[],int l[],int ni){
for(int i=1;i<=n;i++){
   if((i!=ni)&&d[i]){
      int tmp=d[ni]+e[ni][i];
      if(tmp<d[i]){
         d[i]=tmp;
         l[i]=ni;}}}
d[ni]=0;}
```

Figure 4.1: Dijkstra's Algorithm

Quick inspection shows that Dijkstra's algorithm is of order n^2. In fact, initialization is an order n function, as are the minimum computation and update

algorithms. The main algorithm calls the initialization function once and then calls both the minimum and update functions n times giving

$$T(n) \simeq n + n(2n) \simeq n^2.$$

That such a simple algorithm is nearly at the opposite end of the order hierarchy from the exhaustive algorithm gives one an appreciation for the time spent going into the investigation of alternative solutions.

4.2 Minimal Spanning Trees

4.2.1 Prim's Algorithm

Were one interested in producing spanning trees of minimal total weight, a simple modification of Dijkstra's algorithm solves the new problem. As previously alluded, these structures are called Minimal Spanning Trees. An exhaustive enumeration of all possible trees on n vertices may be obtained by enumerating all of the $n-1$ element subsets of the edge set which in the worst case is complete having size $|E| = \binom{n}{2}$. Each of these subsets can be checked in $n-1$ steps to verify whether or not it is a spanning tree for the graph and among those candidate trees one of minimal weight chosen. Using Theorem A.2.5 the time for such an enumeration $W_e(n)$ is bounded below by:

$$\begin{aligned} W_e(n) &= \binom{\binom{n}{2}}{n-1}(n-1) \\ &= \binom{\frac{n(n-1)}{2}}{n-1}(n-1) \\ &\geq \left(\frac{n}{2}\right)^{n-1}(n-1) \\ &= 2\left(\frac{n}{2}\right)^{n-1}\left(\frac{n}{2}-\frac{1}{2}\right) \\ &\simeq \left(\frac{n}{2}\right)^{n} \end{aligned}$$

Using Theorem A.2.9 we have

$$\begin{aligned} \lim_{n\to\infty} \frac{\frac{n!}{\sqrt{n}}}{\left(\frac{n}{2}\right)^n} &= \lim_{n\to\infty} \frac{n!}{\sqrt{n}\left(\frac{n}{2}\right)^n} \\ &= 0, \end{aligned}$$

thus $W_e(n) \succ \frac{n!}{\sqrt{n}}$. Since $\frac{1}{\sqrt{n}} > \frac{1}{n}$, we have $W_e(n) \succ (n-1)!$.

The following algorithm attributed to Prim also grows a tree of internal vertices, initialized with an arbitrary initial vertex, and iteratively incorporates

from the set of external vertices that vertex whose distance to the closest internal vertex is minimal. When all vertices have become internal, the resulting spanning tree has has minimal weight. By using the new update function of Figure 4.2, the meaning of $d[i]$ is changed from distance-to-source to distance-to-closest-internal-vertex.

```
void _update(int n,int e[][],int d[],int l[],int ni){
for(int i=1;i<=n;i++){
   if((i!=ni)&&d[i]){
      if(e[ni][i]<d[i]){
         d[i]=e[ni][i];
         l[i]=ni;}}}
d[ni]=0;}
```

Figure 4.2: Prim's Update

As the control of the algorithm is not changed the order remains the same, n^2.

4.2.2 Kruskals's Algorithm

We will only make brief mention of Kruskal's algorithm as the algorithm is much easier to understand abstractly rather than in detail, the data structures used requiring a different kind of time complexity analysis than the types which we consider in this text. While Prim's Algorithm grows a single tree, another approach is to start with many individual trees and by merging greedily, come up with a single minimal spanning tree. The greedy choice is of an edge of minimal weight used to merge two formerly distinct trees. This approach together with the judicious choice of structures for managing the trees effectively reduces the selection of MST's to the sorting of the edge set in order of increasing edge weight. Given n vertices, the number m of edges for a connected graph exhibits the following bounds.

$$n - 1 \leq m \leq \binom{n}{2},$$

or simply,

$$n - 1 \leq m \leq \frac{n(n-1)}{2}.$$

As will later be seen, optimal sorting based on comparison of keys proceeds in $m\log(m)$ time yielding bounds for Kruskal's time complexity for n vertices,

$$n\log(n) \leq T(n) \leq n^2\log(n).$$

Chapter Exercises

E 4.1. *Design and analyze a greedy algorithm for counting out specific amounts of standard coins. By adding a new non-standard coin show that the greedy algorithm can make a less than optimal choice.*

E 4.2. *Use Dijkstra's algorithm to find a spanning tree for the graph represented by the adjacency matrix. Express your answer as a graph.*

W	*2*	*3*	*4*	*5*	*6*
1	*53*	*57*	*11*	*22*	*35*
2		*63*	*5*	*48*	*55*
3			*61*	*50*	*52*
4				*7*	*43*
5					*28*

E 4.3. *Use Prim's algorithm to find the minimal spanning tree for the graph:*

w	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	*8*	*15*	*9*	*5*	*8*	*13*	*9*
2		*7*	*12*	*21*	*7*	*9*	*6*
3			*4*	*13*	*5*	*5*	*2*
4				*8*	*5*	*6*	*3*
5					*7*	*9*	*14*
6						*1*	*4*
7							*11*

E 4.4. *Use Kruskal's algorithm to find a minimal spanning tree for the same graph.*

E 4.5. *Determine the cost of the exhaustive approach to finding a minimal spanning tree.*

E 4.6. *Compare in detail the orders of Prim's and Kruskal's algorithms with respect to the degree of connectivity of the graph.*

©2003 Steven Louis Davis

Chapter 5

Dynamic Programming

5.1 Recursively Defined Solutions

Both Dynamic Programming algorithms and Divide-and-Conquer algorithms are based on on the application of a recursive mathematical relation between problems of different sizes sharing sufficient structure that all problem sizes can be acted on by common code. The given problem is either built up from several generations of smaller problems or alternatively is successively decomposed into smaller problems. Though these two alternatives described seem identical they are not in fact. The construction of larger and larger problems starting from smallest problems will determine which problems of intermediate size are constructed, and the process of successive decomposition of problems into smaller sizes may determine different sets of problems of intermediate size. If the decomposition produces replication of smaller problems then the process is inherently inefficient. The two approaches are naturally referred to as bottom-up and top-down applications of the recursive relations relating the problem sizes. More commonly, algorithms based on these two approaches are called Dynamic Programming and Divide-and-Conquer algorithms respectively. An important indicator of which application is appropriate is apparent for optimization problems: If the decomposition of an optimal solution for a problem of given size necessarily yields optimal solutions for distinct subordinate smaller problems, then a bottom-up strategy should work. This is sometimes referred to as a principle of optimality.

5.2 Calculating Combinations

Combinations $C(n, k)$ have already occurred several times in our analyses and it is natural to turn to the problem of their computation. Surprisingly the reader is likely already familiar with a dynamic programming solution for their computation. To extract a possibly dormant memory we turn to the simple technique of rapid expansion of integer powers of binomials $(a + b)^n$. Manual

 ©2003 Steven Louis Davis

labor for a few values of n produces the results in Figure 5.1.

$$
\begin{gathered}
1 \\
a+b \\
a^2+2ab+b^2 \\
a^3+3a^b+3ab^2+b^3 \\
a^4+4a^3b+6a^2b^2+4ab^3+b^4
\end{gathered}
$$

Figure 5.1: Binomial Expansions

It is a simple matter to remember to cascade the powers for a^i beginning with n and descending to 0 and also to do the reverse with the powers for b^j. Eliminating all but the coefficients from the expansions yields the simplified diagram of Figure 5.2 which is immediately recognized as Pascal's triangle.

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}
$$

Figure 5.2: Pascal's Triangle

Noting that the outer coefficients are always ones it is apparent that any inner coefficient can be obtained by adding the two coefficients immediately above its position provided of course that they have already been computed. A trivial shifting of the triangle as shown in Figure 5.3 to fit into an ordinary array immediately suggests an algorithmic solution.
Calling the array C and indexing from 0, now any interior element of the triangle can be obtained using the relation:

$$
C(n,k) = \begin{cases} 1, & k \in \{0,n\}, \\ C(n-1,k)+C(n-1,k-1), & 0<k<n. \end{cases} \tag{5.1}
$$

Provided that one begins with the simplest entries for $n=0,1$ and progresses to higher values for n, the necessary entries will always be available for computation of entries in the next line. This is precisely the character of a dynamic

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1

Figure 5.3: Shifted Pascal's Triangle

programming solution. The obvious realization of the algorithm is given in Figure 5.4.

```
int bc1(int n,int k){
if(k==0||k==n)
   return 1;
int C[n+1][n+1];
C[1][0]=C[0][1]=1;
for(int i=2;i<=n;i++)
   for(int j=0;j<=i;j++)
      if(j==0||j==n)
         C[i][j]=1;
      else
         C[i][j]=C[i-1][j]+C[i-1][j-1];
return C[n][k];}
```

Figure 5.4: Binomial Coefficient Algorithm

The time required here can be given as

$$T_{bc1}(n) = 1 + \sum_{i=2}^{n}(i+1)$$

since the inner loop size depends on the outer loop index. Using sum 1.7.1 we obtain

$$\begin{aligned}T_{bc1}(n) &= 1 + \sum_{i=2}^{n}(i+1) \\ &= 1 + \sum_{i=2}^{n} i + \sum_{i=2}^{n} 1 \\ &= \sum_{i=1}^{n} i + n - 1 \\ &= \frac{n(n+1)}{2} + n - 1,\end{aligned}$$

so that

$$T_{bc1}(n) \simeq n^2.$$

One may notice that the computation of $bc1(n,k)$ only requires a rectangular swath of Pascal's triangle delimited by the four entries $(0,0),(k,k),(n,k),(n-k,0)$, giving the algorithm:

```
int bc1a(int n,int k){
int B[n];
if(k==0||k==n)
   return 1;
B[0]=B[1]=1;
for(int i=2;i<=n;i++){
   if(i<=k)
      B[i]=1;
   for(int j=min(i-1,k);j>=max(1,i-n+k);j--)
      B[j]=B[j]+B[j-1];}
return B[k];}
```

Figure 5.5: Minimal Binomial Coefficient Algorithm

This version also uses a single vector to hold only the last row of the computed array. From this version we refine our measure for $T_{bc1}(n)$ leaving n and k independent as

$$T_{bc1}(n,k) \simeq nk.$$

Then we obtain best and worst cases for extreme values of $k=1, k=n$ as $B_{bc1}(n,k) \simeq n$, and $W_{bc1}(n,k) \simeq n^2$ respectively. Note that if only small values of k will be used that this algorithm justifies its extra minimum and maximum computations quite nicely.

In order to gain appreciation for the dynamic algorithm we would like another algorithm against which to compare. Being that there is no obvious counterpart to the exhaustive enumerations used for the graph theoretic problems

we turn to the only other easy avenue, that of applying the recursive definition 5.1 in a top-down manner obtaining the recursive algorithm in Figure 5.6.

```
int bc2(int n,int k){
if(n<2)
   return 1;
else
   return bc2(n-1,k)+bc2(n-1,k-1);}
```

Figure 5.6: Recursive Binomial Coefficients Algorithm

A simple analysis shows the recursive algorithm to be very poorly behaved. In fact, the algorithm's time is characterized precisely by the magnitude of the coefficient being computed:

$$T_{bc2}(n,k) \simeq \binom{n}{k}.$$

Since the largest values are obtained for $n = \lceil 2k \rceil$, we have ,assuming $n = 2k$,

$$\begin{aligned} W_{bc2}(n,k) &= T(2k,k) \\ &= \binom{2k}{k} \\ &= \frac{(2k)!}{(k!)^2} \\ &= \left(\frac{2k}{k}\right)\left(\frac{2k-1}{k-1}\right)\left(\frac{2k-2}{k-2}\right)\cdots\left(\frac{k+1}{1}\right). \end{aligned}$$

By Theorem A.2.6 we have

$$W_{bc2}(n,k) \succeq 2^k = (\sqrt{2})^n.$$

As $W_{bc1} \simeq n^2 \prec (\sqrt{2})^n$, we retire from the analysis somewhat pleased with the dynamic algorithm.

The reader may have one other algorithm in mind motivated by the formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Since $n!$ can certainly be computed in n time, the three factorials can also be computed and divided in n time giving an algorithm superior to the dynamic algorithm. There is however one draw back to this approach. Owing to the relatively large possible values of $n!, k!$, and $(n-k)!$, one obtains an algorithm where these required intermediate values may overflow the capability of the processor even when $\binom{n}{k}$ is representable on that same processor. If unbounded precision is available this of course poses no difficulty.

5.3 Shortest Paths Revisited

The Shortest Paths Problem simply stated is to find for each pair of vertices in a given graph, a path between them of minimal weight. The general Shortest Paths problem can be thought of as requiring solutions of the Single Source problem using each vertex in turn as the source. Quite naturally, for a graph of n vertices this could be achieved in n^3 time by using Dijkstra's n^2 order algorithm for each of the n source vertices. We next give an alternative algorithm again producing a solution of order n^3 but using a dynamic programming approach. Let $G = (V, E, W)$ be a weighted undirected graph of n vertices. Consider the shortest path between fixed vertices i, j having allowed for the use of any of the vertices in a given set $A \subset V$ as intermediate vertices. Denoting the length of this shortest path by $D(i, j, A)$, we see that the length of the shortest path in G between i, and j is given by $D(i, j, V)$. It is also plain that $D(i, j, \emptyset) = W_{i,j}$. We thus have a description of path lengths between vertices which can describe both paths of no intermediate vertices as well as shortest paths which can have as many as $n-2$ intermediate vertices. Now imagine that for a given set size $k < n$, one has available all shortest path lengths $D(i, j, A)$ between pairs of vertices i, j and considering any subset $A \subset V$ of size k as possible intermediates. It is then a simple matter to derive, for each pair i, j of vertices and new intermediate vertex $k \in V \setminus (A \cup \{i, j\})$ the length of any shortest path between i and j considering all paths within $A \cup \{k\}$. If the inclusion of k yields a path shorter than $D(i, j, A)$, then the length of this new path must satisfy

$$D(i, k, A) + D(k, j, A) < D(i, j, A),$$

and in this case we record

$$D(i, j, A \cup \{k\}) = D(i, k, A) + D(k, j, A).$$

To further simplify matters we will consider only the sets $A_k = \{1, 2, \ldots, k\}$ together with $A_0 = \emptyset$. We now have a simple way to enumerate a particular sequence of sets $A_0, A_1, \ldots, A_n = V$ and computing from each collection $\{D(i, j, A_k)\}_{i,j}$ the collection $\{D(i, j, A_{k+1})\}_{i,j}$. Initializing the collection $\{D(i, j, A_0)\}_{i,j}$ from $(W_{i,j})$, the final collection $\{D(i, j, A_n)\}_{i,j}$ yields all desired shortest path lengths. This description outlines Floyd's Shortest Paths algorithm which is given in figure 5.7.

The results stored in D, P consist of the shortest path lengths together with the information necessary to reconstruct the actual paths. The interpretation of P_{ij} is as the intermediate vertex required to improve the shortest path at the kth stage of the algorithm. A recursive algorithm can then be used to reconstruct the actual set of intermediate vertices in proper order as shown in Figure 5.8. The path is the in-order yield of a binary tree encoded in P.

Since Floyd's algorithm consists of three nested loops of size n we conclude that $T(n) \simeq n^3$. Since the recursive path recovery algorithm never produces more than n vertices, its order n does not increase the order of the main algorithm when the costs of both are considered together.

```
void floyd(int n,int D[][],int P[][]){
for(int k=1;k<=n;k++)
   for(int i=1;i<=n;i++
      for(int j=1;j<=n;j++){
         int l=D[i][k]+D[k][j];
         if(l<D[i][j]){
            D[i][j]=l;
P[i][j]=k;}}}
```

Figure 5.7: Floyd's Algorithm

```
void path(int i,int j){
if(P[i][j]){
path(i,P[i][j]);
cout<<P[i][j]<<endl;
path(i,P[i][j]);}}
```

Figure 5.8: Path Retrieval Algorithm

5.4 Traveling Salesman Problem

Within a connected graph we may encounter circuits, that is paths which can be endlessly traversed. Any circuit connecting all the vertices of a graph is called a Hamiltonian circuit or a tour. Given a graph with at least one tour an obvious tour of interest is one of minimal total edge weight. The identification of such a tour is the goal of the traveling salesman problem. More formally, if $H(G)$ is the set of Hamiltonian circuits for a given graph $G = (V, E, W)$ and $H(G) \neq \emptyset$, we seek any tour $h \in H(G)$ such that $\sum_{e \in h} W(e)$ is minimal.

An exhaustive enumeration in a completely connected graph (the worst case) is quite expensive. Noting that every vertex is on every tour we fix one vertex as the source of enumeration. if $|V| = n$ there are $n-1$ vertices which could be the first (non-source) vertex visited on the tour. For each of these there are $n-2$ remaining vertices which could be next on the tour, $n-3$ for the next vertex and so on. The total number of tours to enumerate is thus $(n-1)!$.

Having arbitrarily fixed vertex i_0 as a source of enumeration we will base a somewhat less expensive enumeration on the definition of the quantity $D(i, A)$ defined for any vertex $i \in V \setminus \{i_0\}$ and $A \subset V \setminus \{i, i_0\}$ to be the length of any shortest path between vertices i, i_0 which passes through every vertex in A. The quantities $D(i, \emptyset)$ are plainly the weights of the edges $\{i, i_0\}$ and are initialized from the weight function of the graph. The only exception to the definition is that when $A = V \setminus \{i_0\}$, we allow $i = i_0$. Then

$$D(i_0, A) = D(i_0, \{1, 2, 3, \ldots, i_0 - 1, i_0 + 1, \ldots, n\}) \tag{5.2}$$

gives the length of any tour of minimal length.

If one has computed all quantities $\{D(i,A)\}_{|A|=k}$ it is then a simple matter to obtain the quantities $\{D(i,A)\}_{|A|=k+1}$. Given A with $|A|=k+1$ we can choose any of the $k+1$ vertices $j \in A$ to form the first edge in the minimal path described by $D(i,A)$. This leaves the k vertices of $A \setminus \{j\}$ which must still be used when traversing any residual minimal path from j to i_0. Since this residual path must be minimal its length is properly described as $D(i, A \setminus \{j\})$. By choosing the vertex j minimizing the path from i directly to j and then continuing along a minimal residual path to i_0 we obtain the relation

$$D(i,A) = \min_{j \in A}\{W_{i,j} + D(j, A \setminus \{j\})\}. \tag{5.3}$$

Using 5.3 we now can obtain $\{D(i,A)\}_{|A|=1}$ from $\{D(i,A)\}_{|A|=0}$, $\{D(i,A)\}_{|A|=2}$ from $\{D(i,A)\}_{|A|=1}$, and penultimately $\{D(i,A)\}_{|A|=n-2}$ from $\{D(i,A)\}_{|A|=n-3}$, adhering to the original definition of $D(i,A)$ that does not allow $i = i_0$. Finally a single calculation using 5.3 with $i = i_0$ is used to find 5.2. Figure 5.9 summarizes the algorithm.

```
for i=1 to n
   D(i,{})=W[i,i0]
for k=1 to n-1
   for each subset A of V\{1} with |A|=k
      for i in V\(A U {1})
         D(i,A)=min {W[i,j]+D(j,A\{j}) : j in A}.
D(1,{2,3,...,n})=min {W[i,j]+D(j,A\{j}) : 1<=j<=n,j<>1}
```

Figure 5.9: Traveling Salesman Algorithm

Minimal paths are reconstructed by augmenting the algorithm with an additional storage structure $P(i,A)$ whose value is the first value of j establishing the minimum $D(i,A)$ in the inner most loop. If this is the sequence $j_1, j_2, \ldots, j_m$ where m is the length of a minimal tour, then these would be stored in P as

$$\begin{array}{ll} j_1 = P(i_0, A_1) & A_1 = V \setminus \{i_0\} \\ j_2 = P(j_1, A_2) & A_2 = A_1 \setminus \{j_1\} = V \setminus \{i_0, j_1\} \\ j_3 = P(j_2, A_3) & A_3 = A_2 \setminus \{j_2\} = V \setminus \{i_0, j_1, j_2\} \end{array}$$

To analyze the algorithm we again count loop iterations. The initializing loop and the final loop are clearly inferior additions to the main nested loop which is where we concentrate. The outer most loop runs k for $n-1$ iterations but the code inside the loop uses the value of k at each iteration and so we will sum on k. The next loop must choose all k element subsets of the $n-1$ element set $V \setminus \{i_0\}$ which is easily obtained as $\binom{n-1}{k}$. The next loop runs through all elements of the $n-k-1$ element set $V \setminus (A \cup \{i_0\})$. Finally, the inermost loop is the minimum computation ranging over the k elements of the set A. We therefore have

$$T(n) = \sum_{k=1}^{n-1} \binom{n-1}{k}(n-k-1)k.$$

With a little massaging this sum is easily worked out. First, using Theorem A.2.1,

$$(n-k-1)\binom{n-1}{k} = (n-1-k)\binom{n-1}{k} = (n-1)\binom{n-2}{k}.$$

Thus $T(n)$ becomes

$$T(n) = (n-1)\sum_{k=1}^{n-1} k\binom{n-2}{k}.$$

Next we reindex the sum, letting $i = k-1$, to obtain

$$T(n) = (n-1)\sum_{i=0}^{n-2} i\binom{n-2}{i}.$$

Now by Theorem A.2.4,

$$\sum_{i=0}^{n-2} i\binom{n-2}{i} = (n-2)2^{n-3}$$

and we have

$$T(n) = (n-1)(n-2)2^{n-3} \simeq n^2 2^n$$

.

It is interesting to note that having devoted so much effort to the algorithm we have obtained a rather expensive solution. It is, however, better than the $(n-1)!$ order exhaustive approach but not by much. In fact, the new approach is still an exhaustive enumeration of a kind, albeit a smarter one. It so happens that no algorithm has yet improved on $n^2 2^n$ order.

5.5 Order of Sequenced Matrix Multiplication

Consider the problem of multiplying a sequence of matrices $(M_i)_{i=1}^n$ with $M_i \in \mathcal{M}_{d_{i-1}d_i}$ for positive integer dimensions $d_0, d_1, \ldots, d_n$. Owing to the associativity of matrix multiplication there are many possible orders of associations available to compute the product $P = \prod_{i=1}^n M_i$. Consider the product:

 ©2003 Steven Louis Davis

$$P = M_1 M_2 M_3 M_4 M_5$$

$$= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

We compute the product in several different orders of association and count the

number of scalar multiplications for each. First a left most association is used.

$$P = [[[M_1M_2]M_3]M_4]M_5$$

$$= \left[\left[\left[\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}\right]\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}\right]\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix}\right]\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$= \left[\left[\begin{pmatrix} 22 & 28 \\ 49 & 64 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}\right]\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix}\right]\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \text{,12 multiplications}$$

$$= \left[\begin{pmatrix} 162 & 212 & 262 & 312 \\ 369 & 482 & 595 & 708 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix}\right]\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \text{,32 multiplications}$$

$$= \begin{pmatrix} 4292 & 5240 \\ 9746 & 11900 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad \text{,16 multiplications}$$

$$= \begin{pmatrix} 20012 & 29544 \\ 45446 & 67092 \end{pmatrix} \quad \text{,8 multiplications}$$

We have required 68 scalar multiplications in total. Next computing with right-most associativity we have:

$$P = M_1[M_2[M_3[M_4M_5]]]$$

$$= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \left[\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \left[\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \left[\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \right] \right] \right]$$

$$= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \left[\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \left[\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 7 & 10 \\ 15 & 22 \\ 23 & 34 \\ 31 & 46 \end{pmatrix} \right] \right] \quad \text{,16 multiplications}$$

$$= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \left[\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \begin{pmatrix} 230 & 184 \\ 534 & 788 \end{pmatrix} \right] \quad \text{,16 multiplications}$$

$$= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 1298 & 1760 \\ 2826 & 3704 \\ 4354 & 5648 \end{pmatrix} \quad \text{,12 multiplications}$$

$$= \begin{pmatrix} 20012 & 26112 \\ 45446 & 59448 \end{pmatrix} \quad \text{,12 multiplications}$$

Here there is a total of 56 scalar multiplications, somewhat better than before. Finally we will use the order:

$$P = [M_1 M_2][[M_3 M_4] M_5]$$

$$= \left[\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \right] \left[\left[\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} \right] \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \right]$$

$$= \begin{pmatrix} 22 & 28 \\ 49 & 64 \end{pmatrix} \left[\left[\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} \right] \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \right] \quad \text{,12 multiplications}$$

$$= \begin{pmatrix} 22 & 28 \\ 49 & 64 \end{pmatrix} \left[\begin{pmatrix} 50 & 60 \\ 114 & 140 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \right] \quad \text{,16 multiplications}$$

$$= \begin{pmatrix} 22 & 28 \\ 49 & 64 \end{pmatrix} \begin{pmatrix} 230 & 340 \\ 534 & 788 \end{pmatrix} \quad \text{,8 multiplications}$$

$$= \begin{pmatrix} 20012 & 29544 \\ 45446 & 67092 \end{pmatrix} \quad \text{,8 multiplications}$$

This order totals 44 scalar multiplications, the best yet. If an efficient algorithm can be devised to determine the best order of association then much time can be saved in the processor doing scalar multiplications.

We first examine an exhaustive enumeration of all possible orders of association. If $N(n)$ is the total number of different associations then considering just the left and right-most associations shows that $N(n) > 2N(n-1)$, since each of these associations accounts for one of the original matrices and then must count the number of possible associations for the remaining sequence of $n-1$ matrices. Using the fact that $N(1) = 1$, we have

$$N(n) > 2N(n-1) > 2^2 N(n-2) > \cdots > 2^k N(n-k) = 2^{n-1},$$

for $n-k=1$. To formulate the dynamic approach we will consider that index j

characterizing the last product performed for a given association. For example, in the left-most association the last product performed is

$$\left(\prod_{i=1}^{n-1} M_i\right) M_n,$$

associated with the index $j = n - 1$, while in the right-most it is given by

$$M_1 \left(\prod_{i=2}^{n} M_i\right)$$

where $j = 1$. In general we are focusing on the last product,

$$\left(\prod_{i=1}^{j} M_i\right)\left(\prod_{i=j+1}^{n} M_i\right).$$

There are $n-1$ such final products from which to select one with a minimal number of scalar multiplications, each subordinate product having several possible orderings for its products. Since $\left(\prod_{i=1}^{j} M_i\right) \in \mathcal{M}_{d_0 d_j}$ and $\left(\prod_{i=j+1}^{n} M_i\right) \in \mathcal{M}_{d_j d_n}$, the number of scalar multiplications for this final product is $d_0 d_j d_n$. Denoting the total number of number of scalar multiplications for the entire sequence by $m(1, n)$ we have

$$m(1, n) = \min_{1 \leq j < n} \{d_0 d_j d_n + m(1, j) + m(j + 1, n)\}.$$

We see that the quantity $m(1, n)$ requires similar quantities $m(1, j)$, and $m(j + 1, n)$. Noting that these describe associations for sequences of length less than n, we have stumbled on a recursive relation that can be used in a dynamic algorithm:

$$m(i_1, i_2) = \min_{i_1 \leq j < i_2} \{d_{i_1-1} d_j d_{i_2} + m(i_1, j) + m(j + 1, i_2)\}. \quad (5.4)$$

The smallest possible sequences contain only one matrix and can thus be trivially initialized as $m(i, i) = 0$ as no scalar multiplications are required. All quantities $m(i, j) = 0$ can be compiled and stored in a table from which the desired association is subsequently retrieved. Figure 5.10 gives the algorithm for compiling the association information together with algorithms for computing the minimum 5.4 and for retrieving the actual association from the compiled association table.

We now turn to the analysis. As the retrieval algorithm is only called enough times to print out the n matrices with a given association, it is an order n algorithm. The *FindMin* algorithm has order $col - row = diag - 1$ and is used within *BuildA*'s nested loops giving a time complexity of (using d, r in place of $diag, row$ respectively)

$$
\begin{aligned}
T(n) &= \sum_{d=2}^{n-1} \sum_{r=1}^{n-d+1} (d-1) \\
&= \sum_{d=2}^{n-1} (d-1) \sum_{r=1}^{n-d+1} 1 \\
&= \sum_{d=2}^{n-1} (d-1)(n-d+1) \\
&= \sum_{d=2}^{n-1} (-d^2 + [n+2]d - [n+1]) \\
&= -\sum_{d=2}^{n-1} d^2 + (n+2) \sum_{d=2}^{n-1} d - \sum_{d=2}^{n-1} (n+1) \\
&= 1 - \sum_{d=1}^{n-1} d^2 + (n+2) \left(\sum_{d=1}^{n-1} d - 1 \right) - \sum_{d=2}^{n-1} (n+1) \\
&= 1 - \frac{(n-1)(n)(2[n-1]+1)}{6} + (n+2) \left(\frac{(n-1)(n)}{2} - 1 \right) - (n+1)(n-2) \\
&= 1 - \frac{(n-1)(n)(2n-1)}{6} + \frac{(n+2)(n-1)(n)}{2} - (n+2) - (n+1)(n-2) \\
&= \frac{1}{6} n^3 + bn^2 + cn + d \\
&\simeq n^3
\end{aligned}
$$

```
int m[n+1][n+1];
int a[n+1][n+1];

void FindMin(int row,int col){
m[row][col]=MAXINT;
for(int j=row;j<col;j++){
   int x=d[row-1]*d[j]*d[col]+m[row][j]+m[j+1][col];
   if(x<m[row][col]){
      m[row][col]=x;
      a[row][col]=j;
  }
 }
}

void BuildA(int n,int d[]){
for(int diag=2;diag<n;diag++)
   for(row=1;row<=n-diag+1;row++){
      col=row+diag-1;
      FindMin(row,col);
  }
FindMin(1,n);
}

void RetrieveAssociation(int i,int j){
// initial call with i=1,j=n
if(i==j)
   cout<<"M"<<i;
else{
   int k=a[i][j];
   cout<<"(";
   retrieve_order(i,k);
   retrieve_order(k+1,j);
   cout<<")";
   }
}
```

Figure 5.10: Order of Sequenced Matrix Multiplication Algorithm

Chapter Exercises

E 5.1. *Given a hypothetical machine architecture where the largest representable integer is* 999*, find* n *and* k *such that* $B(n,k)$ *can be computed with the dynamic algorithm but not with the factorial algorithm.*

E 5.2. *Given the matrix* D_3 *of values for the* $k = 3$ *iteration of Floyd's algorithm, find the values for the* $k = 4$ *iteration [transform* D_3 *into* D_4*]*

D_3	2	3	4	5	6
1	34	13	29	30	38
2		47	18	31	54
3			16	43	51
4				42	17
5					62

E 5.3. *Given the shortest paths encoded by Floyd's algorithm in the following path matrix, find the shortest path between vertices 1 and 6.*

P	2	3	4	5	6	7	8	9
1	8	7	8	0	8	0	7	7
2		4	0	0	0	8	4	0
3			0	9	4	0	0	0
4				2	2	8	0	0
5					2	9	9	0
6						8	4	2
7							0	0
8								0

E 5.4. *For a Traveling Salesperson's Problem of* 4 *vertices, list the order in which the quantities* $D(i, A)$ *are generated by the dynamic algorithm.*

E 5.5. *Use the dynamic algorithm to find the optimal tour for the graph:*

w	2	3	4
1	7	4	6
2		2	6
3			3

E 5.6. *Given the optimal matrix multiplication order encoded in the following matrix, use the order recovery algorithm to print out the optimal matrix associations.*

P	2	3	4	5	6	7	8
1	1	2	3	2	5	5	7
2		2	2	2	5	2	7
3			3	3	5	3	7
4				4	5	4	7
5					5	5	7
6						6	7
7							7

E 5.7. *Use the dynamic algorithm to find the length of an optimal tour for the graph: [you may find the large table convenient for representing* $D(i, A)$.

w	2	3	4	5
1	44	51	26	10
2		23	54	3
3			6	33
4				36

$D(i, A)$	{2}	{3}	{4}	{5}	{2,3}	{2,4}	{2,5}	{3,4}	{3,5}	{4,5}	{2,3,4}	{2,3,5}	{2,4,5}	{3,4,5}	{2,3,4,5}
1															
2															
3															
4															
5															

E 5.8. *Use induction to show that the recursive binary coefficients algorithm computes* $2\binom{n}{k} - 1$ *terms.*

E 5.9. *Analyze the recursive algorithm for computing the order of chained matrix multiplication.*

 ©2003 Steven Louis Davis

Chapter 6

Divide and Conquer

We now turn to the flip side of the application of recursively defined solutions in a top-down fashion. Whereas dynamic algorithms begin with the smallest elements of the problem space and successively assemble larger and larger solutions culminating in the solution of a whole problem instance, divide and conquer algorithms make their first application to the whole problem instance, successively subdividing the problem into smaller instances until trivialy solvable smallest case instances are reached. The algorithms direct the efficient partitioning of problems into smaller problems which do not overlap. To see how this process might go wrong one can consider the top-down application of the recursive principle 5.4 used in the solution of the sequenced matrix multiplication problem:

$$m(i_1, i_2) = \min_{i_1 \leq j < i_2} \{d_{i_1-1} d_j d_{i_2} + m(i_1, j) + m(j+1, i_2)\}.$$

Applying 5.4 using a top-down strategy yields a new algorithm for finding the minimum number of scalar multiplications. (compare with *BuildA* and *FindMin* from Figure 5.10).

For simplicity we have settled with finding the minimum number of scalar multiplications and not bothered with storing information needed for optimal association retrieval. Of course, the natural formulation for the time complexity of a recursively defined algorithm is a recursively defined time complexity function. For the algorithm of Figure 6.1 we have

$$T(n) = \begin{cases} \sum_{j=1}^{n-1} (T(j) + T(n-j)) & , n > 1 \\ 1 & , n = 1. \end{cases}$$

or more simply,

$$T(n) = \begin{cases} 2 \sum_{j=1}^{n-1} T(j) & , n > 1 \\ 1 & , n = 1. \end{cases} \tag{6.1}$$

Again, for brevity we will simplify the analysis by discarding most terms of the sum (all except the last) and settling for a lower bound. We shall later return

```
int FindMin(int i1,int i2,int d[]){
if(i1==i2)
   return 0;
else{
   min=MAXINT;
   for(int j=i1;j<i2;j++){
      int x=d[i1-1]*d[j]*d[i2]+FindMin(i1,j)+FindMin(j+1,i2);
      if(x<min)
         min=x;
      }
   return min;
   }
}
```

Figure 6.1: Top-Down application of equation 5.4

to the full analysis to introduce a technique for simplifying such sums. For the time being we have

$$\begin{aligned} T(n) &> 2T(n-1) \\ &> 2^2T(n-2) \\ &> 2^3T(n-3) \\ &\vdots \\ &= 2^kT(n-k) \end{aligned}$$

When $n-k=1$ we obtain

$$T(n) = 2^k \cdot 1 = 2^{n-1}.$$

This does not compare favorably with the order n^3 of the dynamic algorithm. Examination of the algorithm shows that the top-down application of 5.4 does not partition the problems smaller instances in a non-overlapping way. Most smaller problem instances are solved several times during the computation, a situation which is amply reflected in its comparative inefficiency.

For a positive motivational problem we turn to the searching of sorted arrays. That is, given an array $a[\ldots]$ of size n and a search key k presumed to be an element of a, return the position of k in a. We employ the binary search technique already given in Figure 3.1. In the worst case the maximal number of recursive calls will be made and the key found only when the last subarray being searched has size 1 as detected when $left = right$. The recursive characterization of the time complexity is given by 6.2.

$$W(n) = \begin{cases} 1 + W\left(\frac{n}{2}\right) & , n > 1 \\ 1 & , n = 1 \end{cases} \tag{6.2}$$

6.1 Constructive Induction

The recursively defined time complexity functions are simply referred to as recurrences. We employ a technique to solve recurrences of a particularly simple form, that is, recurrences having only one recursive term. These we refer to as elementary recurrences. The technique is essentially one of repeated substitution. With enough substitutions a pattern can usually be recognized that expresses the function without any recursion. This process of obtaining solutions for recurrences will be referred to as constructive induction.
For the recurrence in 6.2 we proceed as follows:

$$\begin{aligned} W(n) &= 1 + W\left(\frac{n}{2}\right) \\ &= 1 + \left[1 + W\left(\frac{\frac{n}{2}}{2}\right)\right] \\ &= 2 + W\left(\frac{n}{2^2}\right) \\ &= 2 + \left[1 + W\left(\frac{\frac{n}{2^2}}{2}\right)\right] \\ &= 3 + W\left(\frac{n}{2^3}\right) \end{aligned}$$

Continuing this process we establish the formula

$$W(n) = k + W\left(\frac{n}{2^k}\right), \tag{6.3}$$

which correctly gives the result for any number k of substitutions using the recursive part of 6.2 . Reasoning that the argument of W in 6.3 will eventually reach the base value 1, we have

$$\begin{aligned} W(n) &= k + W(1), \quad \frac{n}{2^k} = 1 \\ &= k + 1 \end{aligned} \tag{6.4}$$

Using $\frac{n}{2^k} = 1$, we can solve for k giving

$$W(n) = \log_2(n) + 1.$$

For another simple example consider the Towers of Hanoi problem stated as follows: Given a set of n discs of diameters $1, 2, \ldots, n$ initially placed one atop the other in a single stack of decreasing diameters, move the stack one disc at a time to recreate the original stack in a new location while never placing a disc on top of a smaller disc and by never having more than three stacks in total. By formulating a solution for moving all n discs recursively in terms of moving the top disc and then making a recursive call to move the remaining $n-1$ discs, we obtain the algorithm in Figure 6.2 where we have labeled three stacks $1, 2, 3$.

Assuming the stack resides initially on stack position 1 and is to end at stack position 3, the initial call would be $toh(n,1,3)$. The algorithm just prints out the moves that should be made.

```
void toh(int n,int src,ind dest){
if(n>1){
   int tmp=6-src-dest;
   toh(n-1,src,tmp);
   cout<<"move one disk from stack "<<src<<" to stack "<<dest<<endl;
   toh(n-1,tmp,dest);
   }
}
```

Figure 6.2: Recursive Towers of Hanoi Solution

The corresponding recurrence for the time complexity is given by

$$T(n) = \begin{cases} 1 + 2T(n-1) & ,\ n > 1 \\ 1 & ,\ n = 1 \end{cases} \tag{6.5}$$

Constructive induction yields:

$$\begin{aligned} T(n) &= 1 + 2T(n-1) \\ &= 1 + 2[1 + 2T([n-1]-1)] \\ &= 1 + 2 + 2^2T(n-2) \\ &= 1 + 2 + 2^2[1 + 2T([n-2]-1)] \\ &= 1 + 2 + 2^2 + 2^3T(n-3) \\ &\vdots \\ &= 1 + 2 + 2^2 + \cdots + 2^{k-1} + 2^kT(n-k) \\ &= 1 + 2 + 2^2 + \cdots + 2^{k-1} + 2^kT(1) \quad ,n-k=1 \\ &= 1 + 2 + 2^2 + \cdots + 2^{k-1} + 2^k \\ &= \sum_{i=0}^{k} 2^i \\ &= 2^{k+1} - 1 \\ &= 2^n - 1 \end{aligned}$$

It is sometimes necessary to perform simplifying operations on recurrences to render an elementary recurrence. We now return to the analysis of the top-down solution 6.1 for the Sequenced Matrix Multiplication problem:

$$T(n) = \begin{cases} 2\sum_{j=1}^{n-1} T(j) & , n > 1 \\ 1 & , n = 1. \end{cases}$$

This is decidedly not an elementary recurrence having many recursive terms. We can however quickly obtain an elementary recurrence after considering the difference $T(n) - T(n-1)$. Since

$$\begin{aligned} T(n) - T(n-1) &= 2\sum_{j=1}^{n-1} T(j) - 2\sum_{j=1}^{[n-1]-1} T(j) \\ &= 2T(n-1), \end{aligned}$$

we have $T(n) = 3T(n-1)$. Now we obtain

$$\begin{aligned} T(n) &= 3T(n-1) \\ &= 3[3T([n-1]-1)] \\ &= 3^2 T(n-2) \\ &\vdots \\ &= 3^k T(n-k) \\ &= 3^k T(1) \quad ,n-k=1 \\ &= 3^k \cdot 1 \\ &= 3^{n-1} \end{aligned}$$

It should be noted that sums can easily masquerade as recurrences.

Theorem 6.1.1. *For* $f(n) = a_n + f(n-1), f(1) = a_1$ *we have* $f(n) = \sum_{i=1}^{n} a_i$.

As with any analysis technique there is no substitute for practice. Table 6.1 summarizes results for several elementary recurrences which the student is encouraged to solve before looking at the solutions worked out in the following examples.

Table 6.1: Elementary Recurrences

	$f(n)^*$	solution**	order
1	$1+f(\frac{n}{2})$	$\log_2(n)+1$	$\log(n)$
2	$\frac{1}{n}+f(n-1)$	$\sum_{i=1}^{n}\frac{1}{i}$	$\log(n)$
3	$\log_2(n)+f(\frac{n}{2})$	$\frac{1}{2}\log_2(n)(\log_2(n)+1)+1$	$\log^2(n)$
4	$1+2f(\frac{n}{2})$	$2n-1$	n
5	$n+f(\frac{n}{2})$	$2n-1$	n
6	$1+f(n-1)$	n	n
7	$log_2(n)+2f(\frac{n}{2})$	$3n-\log_2(n)-2$	n
8	$n+2f(\frac{n}{2})$	$n(\log_2(n)+1)$	$n\log(n)$
9	$\log_2(n)+f(n-1)$	$\log_2(n!)+1$	$n\log(n)$
10	$n\log_2(n)+2f(\frac{n}{2})$	$n(\frac{1}{2}\log_2(n)(\log_2(n)+1)+1)$	$n\log^2(n)$
11	$n+f(n-1)$	$\frac{1}{2}n(n+1)$	n^2
12	$n^2+2f(\frac{n}{2})$	$n(2n-1)$	n^2
13	$n^2+f(n-1)$	$\frac{1}{6}n(n+1)(2n+1)$	n^3
14	$1+2f(n-1)$	2^n-1	2^n
15	$n+2f(n-1)$	$2^{n+1}-n-2$	2^n

With the exception recurrences 2,6,9,11, and 13, which are simple sums, we proceed to establish the results of the table in the following examples. In all of the examples the the function satisfying the given recurrence also satisfies $f(1)=1$.

$^*f(1)=1$

**equality holds only for n in the lattice $\mathcal{N}_\beta$ of of inverse compositions,

$$\mathcal{N}_\beta=\{1,\beta^{[-1]}(1),\beta^{[-2]}(1),\ldots,\},$$

where $\beta(n)\in\{\log(n),\frac{n}{2},n-1,\frac{n}{c},n-c\}$.

Example 6.1. *For* $f(n) = 1 + f(\frac{n}{2}), f(n) \simeq \log(n)$.

$$\begin{aligned} f(n) &= 1 + f\left(\frac{n}{2}\right) \\ &= 2 + f\left(\frac{n}{2^2}\right) \\ &\vdots \\ &= k + f\left(\frac{n}{2^k}\right) \end{aligned}$$

For $\frac{n}{2^k} = 1$ *we have* $f(n) = k + 1 = \log_2(n) + 1$.

Example 6.2. *For* $f(n) = \frac{1}{n} + f(n-1), f(n) \simeq \log(n)$.
The solution $f(n) = \sum_{i=1}^{n} \frac{1}{i}$ *is immediate. Comparing the sum with the definite integrals of* $\frac{1}{x}$ *we have the upper bound*

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i} < 1 + \int_{1}^{n} \frac{1}{x}\,\mathrm{d}x = 1 + \ln(n),$$

and the lower bound

$$\sum_{i=1}^{n} \frac{1}{i} > \int_{1}^{n+1} \frac{1}{x}\,\mathrm{d}x = \ln(n+1),$$

which by Theorem 2.6.1 has the same order as $\ln(n)$.

Example 6.3. *For* $f(n) = \log_2(n) + f(\frac{n}{2}), f(n) \simeq \log^2(n)$.

$$\begin{aligned} f(n) &= \log_2(n) + f\left(\frac{n}{2}\right) \\ &= \log_2(n) + \log_2\left(\frac{n}{2}\right) + f\left(\frac{n}{2^2}\right) \\ &= 2\log_2(n) - 1 + f\left(\frac{n}{2^2}\right) \\ &= 3\log_2(n) - 1 - 2 + f\left(\frac{n}{2^3}\right) \\ &\vdots \\ &= k\log_2(n) - \sum_{i=1}^{k-1} + f\left(\frac{n}{2^k}\right) \\ &= k\log_2(n) - \frac{(k-1)k}{2} + f\left(\frac{n}{2^k}\right) \end{aligned}$$

For $\frac{n}{2^k} = 1$ *we have* $k = \log_2(n)$ *and*

$$f(n) = log_2^2(n) - \frac{1}{2}(log_2^2(n) - log_2(n)) + 1.$$

Example 6.4. *For* $f(n) = 1 + 2f(\frac{n}{2}), f(n) \simeq n.$

$$\begin{aligned} f(n) &= 1 + 2f\left(\frac{n}{2}\right) \\ &= 1 + 2\left(1 + 2f\left(\frac{n}{2^2}\right)\right) \\ &= 1 + 2 + 2^2 f\left(\frac{n}{2^2}\right) \\ &= 1 + 2 + 2^2 + 2^3 f\left(\frac{n}{2^3}\right) \\ &\vdots \\ &= \sum_{i=0}^{k-1} 2^i + 2^k f\left(\frac{n}{2^k}\right) \end{aligned}$$

For $\frac{n}{2^k} = 1$ *we have* $n = 2k$ *and*

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

Example 6.5. *For* $f(n) = n + f(\frac{n}{2}), f(n) \simeq n.$

$$\begin{aligned} f(n) &= n + f\left(\frac{n}{2}\right) \\ &= n + \frac{n}{2} + f\left(\frac{n}{2^2}\right) \\ &= n + \frac{n}{2} + \frac{n}{2^2} + f\left(\frac{n}{2^3}\right) \\ &\vdots \\ &= n\sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i + f\left(\frac{n}{2^k}\right) \end{aligned}$$

For $\frac{n}{2^k} = 1$ *we have* $n = 2k$ *and*

$$f(n) = 1 + n\sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i = 1 + n\left(\frac{\frac{1}{2^k} - 1}{\frac{1}{2} - 1}\right).$$

Example 6.6. *For $f(n) = \log_2(n) + 2f(\frac{n}{2}), f(n) \simeq n$.*

$$\begin{aligned} f(n) &= \log_2(n) + 2f\left(\frac{n}{2}\right) \\ &= \log_2(n) + 2\log_2\left(\frac{n}{2}\right) + 2^2 f\left(\frac{n}{2^2}\right) \\ &= \log_2(n) + 2\log_2\left(\frac{n}{2}\right) + 2^2\log_2\left(\frac{n}{2^2}\right) + 2^3 f\left(\frac{n}{2^3}\right) \\ &\vdots \\ &= \sum_{i=0}^{k-1} 2^i \log_2\left(\frac{n}{2^i}\right) + 2^k f\left(\frac{n}{2^k}\right) \end{aligned}$$

For $\frac{n}{2^k} = 1$ we have $k = \log_2(n)$ and

$$\begin{aligned} f(n) &= 2^k + \sum_{i=0}^{k-1} 2^i \log_2\left(\frac{n}{2^i}\right) \\ &= n + \log_2(n)\sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{k-1} i2^i \\ &= n + \log_2(n)(2^k - 1) - ((k-2)2^k + 2) \\ &= n + (n-1)\log_2(n) - (\log_2(n) - 2)n - 2 \\ &= 3n - \log_2(n) - 2 \end{aligned}$$

Example 6.7. *For $f(n) = n + 2f(\frac{n}{2}), f(n) \simeq n\log(n)$.*

$$\begin{aligned} f(n) &= n + 2f\left(\frac{n}{2}\right) \\ &= n + 2\left(\frac{n}{2} + 2f\left(\frac{n}{2^2}\right)\right) \\ &= 2n + 2^2\left(\frac{n}{2^2} + 2f\left(\frac{n}{2^3}\right)\right) \\ &= 3n + 2^3 f\left(\frac{n}{2^3}\right) \\ &\vdots \\ &= kn + 2^k f\left(\frac{n}{2^k}\right) \end{aligned}$$

For $\frac{n}{2^k} = 1$ we have $n = 2k$, $k = \log_2(n)$ and

$$f(n) = n(\log_2(n) + 1).$$

Example 6.8. *For* $f(n) = n\log_2(n) + 2f(\frac{n}{2}), f(n) \simeq n\log^2(n)$.

$$\begin{aligned}
f(n) &= n\log_2(n) + 2f\left(\frac{n}{2}\right) \\
&= n\log_2(n) + 2\left[\frac{n}{2}\log_2\left(\frac{n}{2}\right) + 2f\left(\frac{n}{2^2}\right)\right] \\
&= n\left[\log_2(n) + \log_2\left(\frac{n}{2}\right)\right] + 2^2\left[\frac{n}{2^2}\log_2\left(\frac{n}{2^2}\right) + 2f\left(\frac{n}{2^3}\right)\right] \\
&= n\left[\log_2(n) + \log_2\left(\frac{n}{2}\right) + \log_2\left(\frac{n}{2^2}\right)\right] + 2^3 f\left(\frac{n}{2^3}\right) \\
&\vdots \\
&= n\sum_{i=0}^{k-1}\log_2\left(\frac{n}{2^i}\right) + 2^k f\left(\frac{n}{2^k}\right)
\end{aligned}$$

For $\frac{n}{2^k} = 1$ *we have* $n = 2k$, $k = \log_2(n)$ *and*

$$\begin{aligned}
f(n) &= n\left[1 + \sum_{i=0}^{k-1}\log_2\left(\frac{n}{2^i}\right)\right] \\
&= n\left[1 + k\log_2(n) - \sum_{i=0}^{k-1} i\right] \\
&= n\left[1 + \log_2^2(n) - \frac{(k-1)k}{2}\right] \\
&= n\left[\frac{1}{2}\log_2(n)(\log_2(n)+1) + 1\right]
\end{aligned}$$

Example 6.9. *For* $f(n) = n^2 + 2f(\frac{n}{2}), f(n) \simeq n^2$.

$$\begin{aligned}
f(n) &= n^2 + 2f\left(\frac{n}{2}\right) \\
&= n^2 + 2\left[\left(\frac{n}{2}\right)^2 + 2f\left(\frac{n}{2^2}\right)\right] \\
&= n^2(1 + \frac{1}{2}) + 2^2\left[\left(\frac{n}{2^2}\right)^2 + 2f\left(\frac{n}{2^3}\right)\right] \\
&\vdots \\
&= n^2\sum_{i=0}^{k-1}\frac{1}{2^i} + 2^k f\left(\frac{n}{2^k}\right)
\end{aligned}$$

For $\frac{n}{2^k} = 1$ *we have* $n = 2k$, $k = \log_2(n)$ *and*

$$\begin{aligned} f(n) &= n^2 \sum_{i=0}^{k-1} \frac{1}{2^i} + 2^k \\ &= n^2 \left(\frac{\frac{1}{2^k} - 1}{\frac{1}{2} - 1} \right) + n \\ &= n(2n-1) \end{aligned}$$

Example 6.10. *For* $f(n) = 1 + 2f(n-1), f(n) \simeq 2^n$.

$$\begin{aligned} f(n) &= 1 + 2f(n-1) \\ &= 1 + 2(1 + 2f(n-2)) \\ &= 1 + 2(1 + 2(1 + 2f(n-3))) \\ &= 1 + 2 + 2^2 + 2^3 f(n-3) \\ &\vdots \\ &= \sum_{i=0}^{k-1} 2^i + 2^k f(n-k) \end{aligned}$$

For $n - k = 1$ *we have* $k = n - 1$ *and* $f(n) = \sum_{i=0}^{k} 2^i = 2^n - 1$.

Example 6.11. *For* $f(n) = n + 2f(n-1), f(n) \simeq 2^n$.

$$\begin{aligned} f(n) &= n + 2f(n-1) \\ &= n + 2(n - 1 + 2f(n-2)) \\ &= n + 2(n - 1 + 2(n - 2 + 2f(n-3))) \\ &= n + 2(n - 1 + 2(n - 2 + 2(n - 3 + 2f(n-4)))) \\ &= n(1 + 2 + 2^2 + 2^3) - (2 + 2^3 + 3 \cdot 2^3) + 2^4 f(n-4) \\ &\vdots \\ &= n \sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{k-1} i2^i + 2^k f(n-k) \end{aligned}$$

For $n - k = 1$ *we have* $k = n - 1$ *and using Theorem 1.7.4,* $f(n) = n(2^k - 1) - ((k-2)2^k + 2) + 2^k = 2^{n+1} - n - 2$.

6.2 Fast Exponentiation

Another strikingly simple use of the Divide and Conquer technique yields a fast exponentiation algorithm. Based on the fact that every power is either a square or is one factor away from a square we can compute powers by doing mostly squaring with an occasional additional single factor multiplication. To illustrate, consider the computation of 3^{19}. Using a simple iterative approach would accumulate the result at a cost of 19 multiplications. On the other hand we can factor the computation as follows

$$\begin{aligned} 3^{19} &= 3(3^{18}) \\ &= 3((3^9)^2) \\ &= 3((3(3^8)^2) \\ &= 3((3((3^4)^2)^2) \\ &= 3((3(((3^2)^2)^2)^2) \end{aligned}$$

Here there are only 6 multiplications. This technique, shown in Figure 6.3 is easily coded. Though illustrated here only for integers, the technique can be applied to almost any type of multiplication including matrices.

```
int fast_exp(int x, int n){
if(n==0)
   return 1.0;
else if(n%2)
   return x*fast_exp(x,n-1);
else
   return(square(fast_exp(x,n/2));
}
```

Figure 6.3: Fast Exponentiation Algorithm

In the worst case there is an additional factor multiplication interleaved with each squaring giving a time complexity of

$$W(n) = \begin{cases} 1 + W(\frac{n}{2}) & ,n > 1 \\ 1 & ,n = 1 \end{cases}$$

Recognizing this as recurrence #1 from table 6.1 the solution is log time.

Aware of fast exponentiation, one can easily make a critical design mistake in its application in embedded problems. Consider the problem of polynomial evaluation, that is, given a polynomial

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n,$$

compute its value for a given value of x. A naive approach evaluating term by term appears to offer obvious support for the fast exponentiation algorithm. Comparing the two algorithms in Figure 6.4,

```
float npe_se(int n,float a[],float x){
float r=a[0];
for(int i=1;i<=n;i++)
   r=r+a[i]*slow_exp(x,n);
return r;
}

float npe_fe(int n,float a[],float x){
float r=a[0];
for(int i=1;i<=n;i++)
   r=r+a[i]*fast_exp(x,n);
return r;
}
```

Figure 6.4: Termwise Polynomial Evaluation Algorithms

one has

$$T_{npe_se}(n) \simeq \sum_{i=1}^{n} i \simeq n^2,$$

while

$$T_{npe_fe}(n) \simeq \sum_{i=1}^{n} \log(i) \simeq n \log(n),$$

giving preference to *npe_fe*. There is however a better organization of the the solution which removes the need for exponentiation altogether. Consider the following representation for our polynomial,

$$\begin{aligned} p(x) &= a_0 + x(a_1 + a_2 x + a_3 x^2 + \cdots + a_n x^{n-1}) \\ &= a_0 + x(a_1 + x(a_2 + a_3 x + \cdots + a_n x^{n-2})) \\ &= a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x a_n) \dots)) \end{aligned}$$

This representation is easily coded giving the algorithm of Figure 6.5 which is clearly of order n besting either of the two algorithms relying on exponentiation.

```
   float pe3(int n,float a[],float x){
float r=a[n];
for(int i=n;i>0;i--)
r=x*r+a[i-1]
   return r;
   }
```

Figure 6.5: Termwise Polynomial Evaluation Algorithms

6.3 Sorting Arrays

Here we examine the high performance comparison based sort techniques. Some of the elementary sorts which will not be examined include insertion, bubble, and shell sorting. All of these perform in n^2 time and work in place, that is without requiring extra memory. It is possible to improve on the order for sorting but not by much. We begin by stating without proof an important theorem concerning the behavior of sorts based on atomic comparison of keys. Specialized sorts such as Radixsort which break the keys down for special comparisons are outside the scope of the theorem.

Theorem 6.3.1. *No sort based on the atomic comparison of keys can perform in better than* $n\log(n)$ *time.*

As justification we observe that given n keys the sorted list of keys could be any one of the $n!$ permutations. Since any comparison can only reduce by half the remaining number of comparisons a lower bound on the number of comparisons is the number of times $n!$ can be halved before reducing the number of keys to a single element. This number is thus $\log_2(n!)$ which we have seen in Example 2.14 with order $\simeq n\log(n)$.

Since $n\log(n)$ time is nearly linear for many practical values of n these sorts do constitute a significant improvement in practical terms.

6.3.1 Mergesort

Merge sorting is the only technique which will require extra memory, and its analysis will be the only one that includes both time and memory complexity. The technique is based on a simple method for combining two sorted arrays into a single sorted array. If confronted by two piles of keys, each stack being already sorted, one forms a third stack by examining the two keys on top of the stacks and choosing the key of lesser order to be the next key in the combined stack. This is referred to as a merging operation and described formally by the algorithm in Figure 6.6.

The first loop does the actual merging and the remaining loops perform cleanup on which ever stack is not exhausted in the *merge* loop. If the array $C[]$ has size n then the loops all together account for moving all of these elements. The order of the *merge* operation is thus n. We now obtain the sorting algorithm by dividing the array into two subarrays, sorting those subarrays with recursive calls and merging the sorted results. The algorithm is given in Figure 6.7.
Observing n array element allocations, a size n initialization loop, two loops for splitting the n elements of the whole array, one *merge* operation and two recursive calls for half size subarrays, the time complexity of the *mergesort* algorithm is given by the elementary recurrence

$$T(n) = \begin{cases} n + 2T(\frac{n}{2}) & ,n > 1 \\ 1 & ,n = 1 \end{cases}.$$

```
void merge(int A[],int a,int B[],int b,int C[]){
int i=0,j=0,k=0;
while(i<a&&j<b)
   if(A[i]<B[j])
      C[k++]=A[i++];
   else
      C[k++]=B[j++];
while(i<a)
   C[k++]=A[i++];
while(j<b)
   C[k++]=B[j++];
}
```

Figure 6.6: Merge Algorithm

```
void mergesort(int C[],int n){
if(n>1){
   int h=n/2;
   int A[h],B[n-h];
   for(int i=0;i<h;i++)
      A[i]=C[i];
   for(int i=0;i<n-h;i++)
      B[i]=C[i+h];
   mergesort(A,h);
   mergesort(B,n-h);
   merge(A,h,B,n-h,C);
   }
}
```

Figure 6.7: Mergesort

which is quickly recognized as recurrence #8 from table 6.1 having order $n\log(n)$. It is clear that *mergesort* uses at least as much extra memory as the original array to store the copies made by the preliminary loops. Since *mergesort* immediately stores the array elements in two half size arrays and then calls itself on one at a time (after whose completion the memory is released), the memory complexity of the *mergesort* algorithm is given by the elementary recurrence

$$M(n) = \begin{cases} n + M(\frac{n}{2}) & ,n > 1 \\ 1 & ,n = 1 \end{cases}.$$

which is recurrence #5 with order n.

6.3.2 Selectionsort

We continue with one of the elementary n^2 order sorts, the *selection* sort from which two of the high performance sorts can, in a manner of speaking, be derived. *Selectionsort* is not a Divide and Conquer Algorithm and is being presented only as a basis for deriving the Quicksort and Heapsort algorithms. The *selectionsort* strategy basically stated is to find a maximal element a_i for the whole array, exchange it with a_n, and continue to repeat this process to find maximal elements for the arrays $a[1..n-1], a[1..n-2], \ldots, a[1..2]$, thereby depositing the maxima in positions $a_{n-1}, a_{n-2}, \ldots, a_2$. In this way each element finds its correct position as one of the maxima found. The *selectionsort* algorithm is given in Figure 6.8.

```
void selectionsort(int a[],int n){
for(int i=n;i>1;i--)
   exchange(find_max(a,i),a[i]);
}
```

Figure 6.8: Selectionsort

Assuming the obvious i order sequential scan algorithm for finding the maximum of i elements, the order of *selectionsort* is given by

$$\begin{aligned} T(n) &= \sum_{i=2}^{n} i \\ &= -1 + \sum_{i=1}^{n} i \\ &= -1 + \frac{1}{2}n(n+1) \\ &\simeq n^2 \end{aligned}$$

6.3.3 Quicksort

Perhaps the most fascinating sorting technique due both to its ingenuity and performance is the *Quicksort* algorithm attributed to C. Hoare. This approach turns selection sorting inside out; whereas selection sort makes passes over ever smaller subarrays identifying a maximal element which then is placed at the end of the subarray, *Quicksort* takes an arbitrary element and identifies its correct final position, a process called partitioning, which by its nature divides the array into two subarrays on either side of the correctly positioned element (henceforth called the pivot element). If the array elements are randomly distributed then the positioned element lands on average in the middle of the array yielding two subarrays of half the original size which can then be sorted. For example, the array:

41	87	16	39	25	44	23	72	28

could after partitioning using the last element yielding the array:

16	25	23	28	41	87	39	44	72

The subarrays $(16, 25, 23)$ and $(41, 87, 39, 44, 72)$ can then be sorted using recursive calls. The actual distribution of elements on either side of the pivot element depends on the partitioning strategy. Other possible partitionings for the chosen pivot include:

25	23	16	28	44	87	72	41	39

and

16	25	23	28	87	44	41	72	39

.

Given a working partition algorithm the overall technique is trivialy coded as shown in Figure 6.9 where the initial call to sort an n-element array $a[0..n-1]$ would be $quicksort(a, 0, n-1)$.

```
void quicksort(int a[],int left,int right){
if(left<right){
   int p=partition(a,left,right);
   quicksort(a,left,p-1);
   quicksort(a,p+1,right);
   }
}
```

Figure 6.9: Hoare's Quicksort Algorithm

If not coded with care the partitioning algorithm can require successive data shifting at the cost of efficiency of noticeable order difference. The partition algorithm given in Figure 6.10 employs only a single pass over the array being partitioned by using exchanges only. Its order is easily seen to be n. This partition algorithm also selects the right-most element as the pivot.

Quicksort's behavior is quite dependent on the distribution of the keys being sorted. To illustrate we consider two particular arrays and the time complexity analyses they suggest. First we examine an array where the distribution of keys is such that each pivot element is placed at the midpoint of its subarray by partitioning. For example the following array has the stated property.

63	59	51	84	92	57	98	16	32	46	33	25	19	31	48

.

```
int partition(int a[],int left,int right){
int j=-1;
for(int i=left;i<right;i++)
   if(a[i]<a[right])
      exchange(a[++j],a[i]);
exchange(a[++j],a[right]);
}
```

Figure 6.10: Partitioning Algorithm

For such distributions the time complexity is given by the elementary recurrence

$$T(n) = \begin{cases} n + 2T(\frac{n}{2}) & ,n > 1 \\ 1 & ,n = 1 \end{cases}$$

which is elementary recurrence #8 of order $n\log(n)$. The skeptical reader may have suspected that some contrivance has been employed to achieve the above distribution of elements.

We next consider sorted arrays of n elements. In this case the pivot element is already correctly placed resulting in an $n-1$ element left subarray and an empty right subarray. The same behavior holds true for each subarray to be sorted subsequently. The time complexity is then given by the elementary recurrence

$$T(n) = \begin{cases} n + T(n-1) & ,n > 1 \\ 1 & ,n = 1 \end{cases}$$

which is elementary recurrence #11 of order n^2. We will next show that these do in fact characterize the best and worst cases for *Quicksort*'s performance. Given that *Quicksort* achieves $n\log(n)$ for real examples (where the pivot elements equally divide the subarrays), Theorem 6.3.1 allows us to conclude that its best case time is

$$B_{qs}(n) \simeq n\log(n).$$

We deal with the worst case similarly, by way of a theorem bounding worst case time.

Theorem 6.3.2. $W_{qs}(n) \preceq n^2$.

Proof. Use induction on n to establish the inequality $W(n) \leq \frac{n(n-1)}{2}$. First establish the base case for $n = 0$ for which $W(0) = 0$, and of course $0(0-1)/2 = 0$. Now assume the (strong) inductive hypothesis, that $W(k) \leq \frac{k(k-1)}{2}$ for all $k \leq n$, and proceed to establish the inequality for $k = n+1$. For some $p \in \{1, 2, 3, ..., n+1\}$,

$$W(n+1) = n + W(p-1) + W(n+1-p)$$

.

Since $p-1 \leq n$ and $n+1-p \leq n$, both W terms on the right may be rewritten using the inductive hypothesis:

$$\begin{aligned} W(n+1) &\leq n + \frac{(p-1)(p-1)}{2} + \frac{(n+1-p)(n-p)}{2} \\ &= n + \frac{p^2 - 3p + 2 + n^2 - 2np + p^2 + n - p}{2} \end{aligned}$$

For the desired inequality we must have:

$$\begin{aligned} 2n + p^2 - 3p + 2 + n^2 - 2np + p^2 + n - p \leq n^2 + n, &\quad \text{or} \\ 2n + 2p^2 - 4p + 2 - 2np \leq 0, &\quad \text{or} \\ p^2 - 2p + 1 \leq np - n, &\quad \text{or} \\ (p-1)^2 \leq n(p-1) & \end{aligned}$$

which is certainly true for $p = 1$. For $p > 1$ we may divide the inequality by $(p-1)$ to obtain $p - 1 \leq n$. □

Having established that $W_{qs}(n)$ is bounded above by order n^2, and having demonstrated an example (the sorted distribution) where this behavior is achieved we have established that

$$W_{qs}(n) \simeq n^2.$$

We now turn to the average case analysis. Before proceeding we will need a theorem relating the orders of similarly defined recurrences. To facilitate the discussion it will be necessary to introduce a general form for elementary recurrences. These are the functions f characterized completely for nonnegative nondecreasing functions g, h by

$$f(n) = \begin{cases} g(n) + h(n) f(\beta(n)), & n > 1, \\ 1, & n = 1, \end{cases} \tag{6.6}$$

where $\beta(n)$ is one of the simple functions such as $n-1$ or $\frac{n}{2}$.

The theorem basically states that for two recurrences differing only by the function g, equivalent order of the functions f is determined on the basis of equivalent orders of the functions g. We do most of the work in a preliminary lemma.

Lemma 6.3.1. *If $f_i(n) = g_i(n) + h(n) f_i(\beta(n))$ where $g_1 \preceq g_2$ then $f_1 \preceq f_2$.*

Proof. By Theorem 2.3.2 we may choose n_0 and $c > 0$ such that $g_1(n) \leq c g_2(n)$ for $n \geq n_0$. Clearly $g_i \simeq G_i$ where $G_i = 0$ on $[0, n_0]$ and agrees with g_i for $n > n_0$. We prove the result for the corresponding restrictions $F_i = f_i|_{\{n_0, n_0+1, \ldots\}}$ by induction.

First, $F_1(n_0) = G_1(n_0) = 0 = G_2(n_0) = F_2(n_0)$, thus $F_1(n_0) \leq cF_2(n_0)$ is established as a base case. Assuming $F_1(n) \leq cF_2(n)$ for $n \geq n_0$ we have

$$\begin{aligned} F_1(\beta^{-1}(n)) &= G_1(\beta^{-1}(n)) + h(\beta^{-1}(n))F_1(n) \\ &\leq cG_2(\beta^{-1}(n)) + h(\beta^{-1}(n))F_1(n) \\ &\leq cG_2(\beta^{-1}(n)) + ch(\beta^{-1}(n))F_2(n) \\ &= cF_2(\beta^{-1}(n)) \end{aligned}$$

As this holds for all n, another apeal to Theorem 2.3.2 establishes the desired result. □

Theorem 6.3.3. *If $f_i(n) = g_i(n) + h(n)f_i(\beta(n))$ where $g_1 \simeq g_2$ then $f_1 \simeq f_2$.*

Proof. Since $g_1 \simeq g_2$ we have $g_1 \preceq g_2$ and $g_2 \preceq g_1$, therefore by the lemma, $f_1 \preceq f_2$, and $f_2 \preceq f_1$, that is, $f_1 \simeq f_2$. □

Theorem 6.3.4. *$A_{qs}(n) \simeq nlog(n)$.*

Proof. For *quicksort*'s average case we will use Theorem 1.6.1 and average all possible running times given by the possible values of the final pivot position resulting from any partition. Thus we require the expected value of the function

$$T(n) = n + T(p-1) + T(n-p)$$

of the random variable p taking values in $\{1, 2, ..., n\}$:

$$\begin{aligned} A(n) &= E[n + A(p-1) + A(n-p)] \\ &= \frac{[n + A(0) + A(n-1)] + [n + A(1) + A(n-2)] + [n + A(2) + A(n-3)] + ... + [n + A(n-1) + A}{n} \\ &= n + \frac{2}{n}\sum_{i=0}^{n-1} A(i) \end{aligned}$$

Then,

$$\begin{aligned} nA(n) &= n^2 + 2\sum_{i=0}^{n-1} A(i), \\ (n+1)A(n+1) &= (n+1)^2 + 2\sum_{i=0}^{n} A(i), \\ (n+1)A(n+1) - nA(n) &= (n+1)^2 - n^2 + 2A(n) \end{aligned}$$

Rearranging a bit yields:

$$\begin{aligned} (n+1)A(n+1) &= 2n + 1 + (n+2)A(n), \\ \frac{A(n+1)}{n+2} &= \frac{2n+1}{(n+1)(n+2)} + \frac{A(n)}{n+1} \end{aligned}$$

Now we define $g(n) = \frac{A(n+1)}{n+2}$ and note the similarity between the resulting recurrence for g:

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

and the elementary recurrence f (#2 from Table 6.1). In fact, since

$$\lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{2n+1}{(n+1)(n+2)}} = \frac{1}{2},$$

f and g have the same order by Theorem 6.3.3; that is, $\frac{A(n+1)}{n+2} \simeq \log(n)$. By Theorem 2.6 we have $\frac{A(n)}{n+1} \simeq \log(n)$ and by Theorem 2.1.2, $A(n) \simeq n\log(n)$. □

6.3.4 Heapsort

From a grand perspective, *heapsort* is a superficial modification of selectionsort. The process of selecting maxima with a sequential scan is replaced by one that extracts the maximum of a heap. Since this maximum is always at the top of a heap the work is shifted to the process of making and remaking heaps. As can be seen by comparing Figures 6.8 and 6.11, the algorithms are structurally similar.

```
void heapsort(int a[],int n){
make_heap(a,0,n);
for(int i=n;i>0;){
   exchange(a[0],a[i]);
   fix_heap(a,0,--i);
   }
}
```

Figure 6.11: Heapsort

Here the array has first been mapped onto a left-balanced binary tree as described in section 1.5. The operation *make_heap* forms the binary tree into a heap, that is, a tree where the root of each subtree is maximal for that subtree. We shall see that a heap damaged by the modification of its root node is easily restored to the heap condition, this restoration being the task of the operation *fix_heap*. Like selectionsort, *heapsort* works on ever decreasing heaps, exchanging the maximum with the end of the working heap which then being no longer a heap is restored before the next iteration.

To make a heap recursively is very simple if the *fix_heap* routine is available, for then one makes both left and right subtrees first into heaps and then fixes the whole heap. The code appears in Figure 6.12.

Each test simply makes sure the appropriate tree is nonempty before proceeding. Now all that is left is the heap fixing routine. Here we basically do a binary

```
void make_heap(int a[],int root,int size){
if(root<=size){
   int left=2*root,right=left+1;
   if(left<=size){
      make_heap(a,left,size);
      if(right<=size)
         make_heap(a,right,size);
      fix_heap(a,root,size);}}}
```

Figure 6.12: Makeheap

search where the visit at each node makes sure to promote the largest of three elements, that at the current node and those rooting the left and right subtrees at that node. Again, some tests just check for nonempty subtrees.

```
void fix_heap(char *a[],int root,int n){
int left=2*root,right=left+1,max_index=root;
if(left<=n){
   if(string(a[left])>string(a[max_index]))
      max_index=left;
   if(right<=n&&string(a[right])>string(a[max_index]))
      max_index=right;
   if(max_index!=root){
      exchange(a,max_index,root);
      fix_heap(a,max_index,n);}}}
```

Figure 6.13: Fixheap

Let $W_{hs}(n), W_{mh}(n)$, and $W_{fh}(n)$ be the worst case time complexities of *heap_sort, make_heap,* and *fix_heap* respectively. Examining *heap_sort* we see that

$$W_{hs}(n) \simeq W_{mh}(n) + \sum_{i=n}^{1} W_{fh}(i). \tag{6.7}$$

For *make_heap* we get

$$W_{mh}(n) \simeq W_{fh}(n) + 2W_{mh}(\frac{n}{2}), \tag{6.8}$$

and for *fix_heap* we have

$$W_{fh}(n) \simeq \begin{cases} 1 + W_{fh}(\frac{n}{2}) & , n > 1 \\ 1 & , n = 1 \end{cases}$$

which yields $W_{fh}(n) \simeq \log(n)$ (recurrence #1 of Table 6.1). Substituting, 6.8 becomes

$$W_{mh}(n) \simeq \log(n) + 2W_{mh}\left(\frac{n}{2}\right),$$

which yields $W_{mh}(n) \simeq n$ (recurrence #7 of Table 6.1). Finally 6.7 becomes

$$\begin{aligned} W_{hs}(n) &\simeq n + \sum_{i=n}^{1} \log(i) \\ &= n + \log(n!) \\ &\simeq n \log(n). \end{aligned}$$

Bing both an in-place sort and having worst case time optimal makes *heapsort* the best comparison based sort considered here.

6.4 Simplifying Recurrences

Theorem 6.3.3 used in the analysis of Quicksort is an extremely valuable tool capable of saving much time in analysis. We restate it here:

Theorem 6.4.1. *If* $f_i(n) = g_i(n) + h(n)f_i(\beta(n))$ *where* $g_1 \simeq g_2$ *then* $f_1 \simeq f_2$.

It says that just as we have learned to recognize and use the simplest form for a time complexity function in representing its order, we can do the same for the nonrecursive part of a recurrence before solving. This means that for example, the recurrence $f(n) = 3n^2 + n\log_s(n) - n + 2\log_2(n) + 2f(\frac{n}{2})$ which would be unpleasant to solve exactly, has the same order as the recurrence $f_1(n) = n + 2f(\frac{n}{2})$. A few examples should serve to build an appreciation of the theorem.

Example 6.12. *For* $f(n) = n + 3\log_2(n) + f(\frac{n}{2}), f(n) \simeq n$.
First we establish the result using constructive induction directly. We have

$$\begin{aligned}
f(n) &= n + 3\log_2(n) + f\left(\frac{n}{2}\right) \\
&= n + 3\log_2(n) + \frac{n}{2} + 3\log_2\left(\frac{n}{2}\right) + f\left(\frac{n}{2^2}\right) \\
&= n\left(1 + \frac{1}{2}\right) + 3\left[\log_2(n) + \log_2\left(\frac{n}{2}\right)\right] + f\left(\frac{n}{2^2}\right) \\
&= n\left(1 + \frac{1}{2} + \frac{1}{2^2}\right) + 3\left[\log_2(n) + \log_2\left(\frac{n}{2}\right) + \log_2\left(\frac{n}{2^2}\right)\right] + f\left(\frac{n}{2^3}\right) \\
&\vdots \\
&= n\sum_{i=0}^{k-1}\frac{1}{2^i} + 3\sum_{i=0}^{k-1}\log_2\left(\frac{n}{2^i}\right) + f\left(\frac{n}{2^k}\right)
\end{aligned}$$

For $\frac{n}{2^k} = 1$ *we have*

$$\begin{aligned}
f(n) &= n\sum_{i=0}^{k-1}\frac{1}{2^i} + 3\sum_{i=0}^{k-1}\log_2\left(\frac{n}{2^i}\right) + 1 \\
&= n\left(\frac{\frac{1}{2^k} - 1}{\frac{1}{2} - 1}\right) + 3\sum_{i=0}^{k-1}(\log_2(n) - i) + 1 \\
&= 2(n-1) + 3k\log_2(n) - 3\sum_{i=0}^{k-1} i + 1 \\
&= 2n - 1 + 3\log_2^2(n) - 3\frac{(k-1)k}{2} \\
&= 2n + \frac{3}{2}\log_2^2(n) + \frac{1}{2}\log_2(n) - 1 \\
&\simeq n
\end{aligned}$$

Compare this analysis with that already done for the simplified recurrence $f(n) = n + f(\frac{n}{2})$ *given in Example 5.*

For some other kinds of functions, the theorem is indispensable. Consider the function $f(n) = \sqrt{n^2 - 1} + f(\frac{n}{2})$. Constructive induction eventually yields

$$f(n) = 1 + \sum_{i=0}^{\log_2(n)-1} \sqrt{\left(\frac{n}{2^i}\right) - 1}.$$

With some work it could be shown that this sum is of order n, but it is much simpler to recognize that $\sqrt{n^2 - 1} \simeq n$ and solve the simpler recurrence, again that of Example 5. To see that $\sqrt{n^2 - 1} \simeq n$, observe that

$$\lim_{n\to\infty} \frac{\sqrt{n^2 - 1}}{n} = \lim_{n\to\infty} \sqrt{1 - \frac{1}{n^2}} = 1.$$

Example 6.13. *For* $f(n) = \log_2(2^n + 3) + 2f(\frac{n}{2})$, $f(n) \simeq n\log(n)$.
Finding the limit

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_2(2^n + 3)}{n} &= \lim_{n\to\infty} \frac{\log_2(2^n + 3)}{\log_2(2^n)} \\
&\stackrel{L'}{=} \lim_{n\to\infty} \frac{\frac{1}{(2^n+3)\ln(2)}}{\frac{1}{2^n \ln(2)}} \\
&= \lim_{n\to\infty} \frac{2^n}{2^n + 3} \\
&= \lim_{n\to\infty} \frac{1}{1 + \frac{3}{2^n}} \\
&= 1
\end{aligned}$$

Thus $f(n)$ *simplifies to* $f_1(n) = n + 2f(\frac{n}{2})$ *which is solved in Example 8 having order* $n\log(n)$.

6.5 Fast Fourier Transform

6.5.1 The Discrete Fourier Transform

Let F be a scalar field, $\rho \in F$, and $V_\rho = (\rho^{ij})_{i,j=0}^{n-1}$ the Vandermonde matrix for ρ. Given $f = (f_i)_{i=0}^{n-1} \in F^n$, the Discrete Fourier Transform of f with respect to ρ is given by the product

$$\hat{f} = V_\rho f = \left(\sum_{j=0}^{n-1} \rho^{ij} f_j \right)_{i=0}^{n-1}.$$

If ρ is a principle n^{th} root of unity in F, then the mapping $f \to \hat{f}$ is invertible, that is, V_ρ has an inverse. Thus the vectors f are in a one-to-one correspondence with their transforms $\hat{f}$. This correspondence is a homomorphism for two different types of products on F^n, one being the simple componentwise product given for $f = (f_i)_{i=0}^{n-1}$ and $g = (g_i)_{i=0}^{n-1}$ by

$$fg = (f_i g_i)_{i=0}^{n-1}.$$

If f and g represent the values of two functions at n distinct points then this product would represent the values of the product of those functions at the same points. The other product is the vector convolution given by

$$f * g = \left(\sum_{j=0}^{n-1} f_j g_{i-j} \right)_{i=0}^{n-1}.$$

The homomorphism is given by $\widehat{f * g} = \hat{f}\hat{g}$, that is, the transform of the convolution is the componentwise product of the transforms. Appendix A.1 provides justifications of the above theory.

Computation time is not preserved across the homomorphism as one can see, the simple product is an order n operation while the convolution is and order n^2. operation. The correspondence of products offers another way of finding the convolution using transforms. Given f, g, find their transforms, $\hat{f}, \hat{g}$, form the componentwise product $\hat{f}\hat{g}$, and then find the inverse transform. If the transform and inverse transform operations can be done in better than n^2 time, this approach constitutes an improvement.

$$\begin{array}{ccc} f, g & \longrightarrow & f * g \\ V_\rho \big\downarrow & & \big\uparrow V_\rho^{-1} \\ \hat{f}, \hat{g} & \longrightarrow & \hat{f}\hat{g} \end{array}$$

Figure 6.14: Computing the Convolution using the DFT.

It is somewhat more familiar to realize the vector convolution as a polynomial product. Denoting by $f(x)$ and $g(x)$ the polynomials having f and g as coefficient vectors,

$$f(x) = \sum_{i=0}^{n-1} f_i x^i, \qquad \text{and} \qquad g(x) = \sum_{i=0}^{n-1} g_i x^i$$

respectively, the polynomial product of $f(x)$ and $g(x)$ is given by

$$\begin{aligned} f(x)g(x) &= \sum_{i=0}^{n-1} f_i x^i \sum_{j=0}^{n-1} g_j x^j \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_i g_j x^{i+j} \\ &= \sum_{i=0}^{n-1} \sum_{k=i}^{n-1+i} f_i g_{k-i} x^k,\ k = i+j \end{aligned}$$

This is the wrapped polynomial product. For simpler illustration consider $f, g \in F_0^n = \{(f_i)_{i=0}^{n-1} : f_i = 0, i \geq \frac{n}{2}\}$. Then the sum reduces to the familiar polynomial product

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i} f_j g_{i-j} x^i$$

where the degree of the product is the sum of the degrees of the factors.

We will now illustrate the process depicted in Figure 6.14. Let $n = 4, \rho = 4$, and $m = 17$. Then ρ is a principle n^{th} root of unity in Z_m, that is, for arithmetic modulo 17, and $n^{-1} = 13$. The transform matrix is given by

$$V_\rho = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 4 & -1 & -4 \\ 1 & -1 & 1 & -1 \\ 1 & 4 & -1 & 4 \end{pmatrix},$$

and its inverse by

$$(V_\rho)^{-1} = \frac{1}{n} V_{\rho^{-1}} = 13 \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -4 & -1 & 4 \\ 1 & -1 & 1 & -1 \\ 1 & 4 & -1 & -4 \end{pmatrix}.$$

Let $f = (2,3,0,0), g = (1,1,5,0)$, with associated polynomials $f(x) = 3x + 2$, $g(x) = 5x^2 + x + 1$. Then polynomial product of is given by $f(x)g(x) = 15x^3 + 13x^2 + 3x + 2$. Computing transforms we have

$$\hat{f} = V_\rho f = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 4 & -1 & -4 \\ 1 & -1 & 1 & -1 \\ 1 & 4 & -1 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ -3 \\ -1 \\ 7 \end{pmatrix},$$

and

$$\hat{g} = V_\rho g = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 4 & -1 & -4 \\ 1 & -1 & 1 & -1 \\ 1 & 4 & -1 & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 5 \\ 0 \end{pmatrix} = \begin{pmatrix} 7 \\ 0 \\ 5 \\ 8 \end{pmatrix}.$$

The componentwise product is

$$\hat{f}\hat{g} = \begin{pmatrix} 5 \\ -3 \\ -1 \\ 7 \end{pmatrix} \begin{pmatrix} 7 \\ 0 \\ 5 \\ 8 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -5 \\ -5 \end{pmatrix},$$

and its inverse transform

$$(V_\rho)^{-1}(\hat{f}\hat{g}) = 13 \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -4 & -1 & 4 \\ 1 & -1 & 1 & -1 \\ 1 & 4 & -1 & -4 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -5 \\ -5 \end{pmatrix} = \begin{pmatrix} 2 \\ 5 \\ 13 \\ 15 \end{pmatrix},$$

which is the coefficient vector for $f(x)g(x)$.

6.5.2 The Fast DFT Algorithm

Let $k > 1, n = 2^k$, and ρ a principle n^{th} root of unity. Except for the first, each sum in the transform $\hat{f} = (\sum_{j=0}^{n-1} f_j \rho^{ij})_{i=0}^{n-1}$ has a number of multiplications bounded by n, thus the computation of $\hat{f}$ normally proceeds in order n^2

time. Now define $g = (f_0, f_2, f_4, \ldots, f_{n-2})$, $h = (f_1, f_3, f_5, \ldots, f_{n-1})$, so that $f(x) = g(x^2) + xh(x^2)$ where $g(x), h(x)$ are the polynomials having g, h as coefficient vectors. For $m = \frac{n}{2}, \theta = \rho^2$, we have $\theta^m = (\rho^2)^m = \rho^n = 1$, so that θ is an mth root of unity and can be used to define Fourier Transforms using the matrix V_θ. θ will also be a principle m^{th} root of unity. Since $f(\rho^i) = g(\theta^i) + \rho^i h(\theta^i)$, we have

$$
\begin{aligned}
V_\rho f &= \left(g(\theta^i) + \rho^i h(\theta^i)\right)_{i=0}^{n-1} \\
&= \begin{pmatrix}
g(1) + h(1) \\
g(\rho^2) + \rho h(\rho^2) \\
g(\rho^4) + \rho^2 h(\rho^4) \\
\vdots \\
g(\rho^{n-2}) + \rho^{\frac{n}{2}-1} h(\rho^{n-2}) \\
g(\rho^n) + \rho^{\frac{n}{2}} h(\rho^n) \\
g(\rho^{n+2}) + \rho^{\frac{n}{2}+1} h(\rho^{n+2}) \\
g(\rho^{n+4}) + \rho^{\frac{n}{2}+2} h(\rho^{n+4}) \\
\vdots \\
g(\rho^{2n-2}) + \rho^{n-1} h(\rho^{2n-2})
\end{pmatrix}
= \begin{pmatrix}
g(1) + h(1) \\
g(\rho^2) + \rho h(\rho^2) \\
g(\rho^4) + \rho^2 h(\rho^4) \\
\vdots \\
g(\rho^{n-2}) + \rho^{\frac{n}{2}-1} h(\rho^{n-2}) \\
g(1) + \rho^{\frac{n}{2}} h(1) \\
g(\rho^2) + \rho^{\frac{n}{2}+1} h(\rho^2) \\
g(\rho^4) + \rho^{\frac{n}{2}+2} h(\rho^4) \\
\vdots \\
g(\rho^{n-2}) + \rho^{n-1} h(\rho^{n-2})
\end{pmatrix}
\end{aligned}
$$

Thus the computation of the transform for $f \in F^n$ is reduced to the computation of two transforms for $g, h \in F^{\frac{n}{2}}$. The time complexity is then defined by

$$
T(n) = \begin{cases} n + 2T(\frac{n}{2}) & , n > 1, \\ 1 & , n = 1. \end{cases}
$$

which is recurrence #8 from table 6.1 having order $n \log(n)$. This may seem to be quite a lot of work for a small improvement in order, but this particular improvement cannot be historically understated. It is one of the principle technologies responsible for the telecommunications revolution of the second half of the last century.

 ©2003 Steven Louis Davis

6.6 Exploiting Associations with Recursion

Here we will see two examples where recursion can be used to exploit simple algebraic associations. Accordingly, both examples are numeric in character and the straight forward solutions not employing recursion offer no hint at improvement. These are good examples of situations where simply adopting a recursive approach will suggest a new path toward solution.

6.6.1 Large Integer Arithmetic

Most arithmetic is well accommodated by the hardware of modern computing machines. There are instances however where the integers dealt with are too large to be directly manipulated by the machines processors. Since the early nineties, encryption software has become extremely important for on-line businesses, being used to insure the privacy of financial transactions. These systems routinely use integers beyond the machines direct capabilities. While most personal computers are limited to 32 bit or 64 bit integers, the machine word size, systems such as RSA typically require 1000 or more bits to guarantee secrecy. It is a straight forward exercise to write software to manipulate large integers by breaking them down into machine size pieces stored together in an array and then using ordinary techniques to manipulate these arrays as a whole. While it is natural to let the size of the pieces be that of the machine word size, we will work with arrays of decimal digits for simplicity. Thus a large integer such as $i = 865976394133249617$ is stored as an array of 18 decimal digits. To add two such numbers, one simply adds digit by digit beginning with the least significant and including carries into the next digit when necessary. The size of the integers is now limited only by memory if using dynamic arrays or by the maximum array size chosen in the case of static arrays. The usual restrictions of result size still apply of course, with sums generally requiring one additional digit for the last carry while multiples generally require twice the number of digits as the largest factor. Fixing a maximum digit array size of n, addition of large integers is clearly an order n operation. The grade school multiplication operation that multiplies each digit of one number by every digit of the other is an order n^2 algorithm. Another special multiplication will be important, that of multiplying by a power of 10 which involving nothing more than digit shifts is also an order n operation. It is our goal to improve on the order n^2 multiplication algorithm.

We begin with two large integers $a = \sum_{i=0}^{n} a_i 10^i$, and $b = \sum_{i=0}^{n} b_i 10^i$, where n is odd. Letting $k = \frac{n+1}{2}$ the following sum shows how to represent our size n integers using two size $\frac{n}{2}$ integers.

$$\begin{aligned} a &= \sum_{i=0}^{k-1} a_i 10^i + \sum_{i=k}^{n} a_i 10^i \\ &= \sum_{i=0}^{k-1} a_i 10^i + \sum_{j=0}^{k-1} a_{k+j} 10^{k+j} \\ &= \sum_{i=0}^{k-1} a_i 10^i + 10^k \sum_{i=0}^{k-1} a_{k+i} 10^i \end{aligned}$$

Similarly,

$$b = \sum_{i=0}^{k-1} b_i 10^i + 10^k \sum_{i=0}^{k-1} b_{k+i} 10^i.$$

Let

$$\begin{aligned} x_2 &= \sum_{i=0}^{k-1} a_i 10^i, \\ x_1 &= \sum_{i=0}^{k-1} a_{k+i} 10^i, \\ x_4 &= \sum_{i=0}^{k-1} b_i 10^i, \qquad \text{and} \\ x_3 &= \sum_{i=0}^{k-1} b_{k+i} 10^i, \end{aligned}$$

so that x_1 and x_3 are the most significant halves of the digit strings for a and b while x_2 and x_4 are the least significant. That is,

$$\begin{aligned} a &= 10^k x_1 + x2, \qquad \text{and} \\ b &= 10^k x_3 + x_4. \end{aligned}$$

The product is then given by

$$\begin{aligned} ab &= (10^k x_1 + x2)(10^k x_3 + x_4) \\ &= 10^n x_1 x_3 + 10^k (x_1 x_4 + x_2 x_3) + x_2 x_4. \end{aligned}$$

Suppose we employ a recursive strategy to multiply our size n numbers by making recursive calls to compute the four subordinate size $\frac{n}{2}$ products

$$\begin{aligned} p_1 &= x_1 x_3, \\ p_2 &= x_1 x_4, \\ p_3 &= x_2 x_3, \qquad \text{and} \\ p_4 &= x_2 x_4. \end{aligned}$$

Denoting the time complexity of this approach by $T(n)$ and noting that the addition and multiplication by powers of 10 are order n operations we have

$$T(n) = \begin{cases} 5n + 4T(\frac{n}{2}) & , n > 1 \\ 1 & , n = 1. \end{cases}$$

We use constructive induction to solve the recurrence.

$$
\begin{aligned}
T(n) &= 5n + 4T\left(\frac{n}{2}\right) \\
&= 5n + 4\left[5 \cdot \frac{n}{2} + 4T\left(\frac{n}{2^2}\right)\right] \\
&= 5n + 4 \cdot 5 \cdot \frac{n}{2} + 4^2\left[5 \cdot \frac{n}{2^2} + 4T\left(\frac{n}{2^3}\right)\right] \\
&= 5n(1 + 2 + 2^2) + 4^3 T\left(\frac{n}{2^3}\right) \\
&\vdots \\
&= 5n\sum_{i=0}^{k-1} 2^i + 4^k T\left(\frac{n}{2^k}\right)
\end{aligned}
$$

For $\frac{n}{2^k} = 1$ we have $2^k = n$ and

$$
\begin{aligned}
T(n) &= 5n\sum_{i=0}^{k-1} 2^i + 4^k \\
&= 5n\left(\frac{2^k - 1}{2 - 1}\right) + (2^k)^2 \\
&= 5n(n-1) + n^2 \\
&= 6n^2 - 5n
\end{aligned}
$$

which has order n^2. Thus a recursive strategy alone produces no benefit for this new type of large integer multiplication. It is possible to exploit the recurrence however by reducing the number of recursive calls from 4 to 3. This is achieved by taking advantage of a 5th product $p_5 = (x_1 + x_2)(x_3 + x_4)$, and the fact that two of the needed products p_2 and p_3 always occur summed. Since

$$
\begin{aligned}
p_5 &= (x_1 + x_2)(x_3 + x_4) \\
&= x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 \\
&= p_1 + p_2 + p_3 + p_4
\end{aligned}
$$

we have $p_2 + p_3 = p_5 - p_1 - p_4$. With only 3 subordinate products for the recursive calls we now have the time complexity

$$
T(n) = \begin{cases} 8n + 3T(\frac{n}{2}) & , n > 1 \\ 1 & , n = 1. \end{cases}
$$

The extra addition for p_5 and the two subtractions accounting for the increase from $5n$ to $8n$. We use constructive induction to solve the new recurrence.

$$
\begin{aligned}
T(n) &= 8n + 3T\left(\frac{n}{2}\right) \\
&= 8n + 3\left[8 \cdot \frac{n}{2} + 3T\left(\frac{n}{2^2}\right)\right] \\
&= 8n + 3 \cdot 8 \cdot \frac{n}{2} + 3^2\left[8 \cdot \frac{n}{2^2} + 3T\left(\frac{n}{2^3}\right)\right] \\
&= 8n\left[1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2\right] + 3^3T\left(\frac{n}{2^3}\right) \\
&\vdots \\
&= 8n\sum_{i=0}^{k-1}\left(\frac{3}{2}\right)^i + 3^kT\left(\frac{n}{2^k}\right)
\end{aligned}
$$

For $\frac{n}{2^k} = 1$ we have $k = \log_2(n)$ and

$$
\begin{aligned}
T(n) &= 8n\sum_{i=0}^{k-1}\left(\frac{3}{2}\right)^i + 3^k \\
&= 8n\left(\frac{(\frac{3}{2})^k - 1}{\frac{3}{2} - 1}\right) + 3^k \\
&= 16n\left(\frac{3^k}{n} - 1\right) + 3^k \\
&= 17 \cdot 3^k - 16n \\
&= 17 \cdot 3^{\log_2(n)} - 16n \\
&= 17n^{\log_2(3)} - 16n
\end{aligned}
$$

of order $n^{\log_2(3)}$ which is approximately $n^{1.58} \prec n^2$. For 1000 bit integers, roughly 100 decimal digits, the improvement is quite dramatic, $100^{1.58} \approx 1478$ being only about 15% of 100^2. We can also see the early cost of the recursive approach in the much larger principle coefficient. It is interesting to compare the two time complexities.
We can see tthat $6n^2 - 5n$ has overtaken $17n^{\log_2(3)} - 16n$ by $n = 8$. In fact for $n = 6$, $6n^2 - 5n = 186$ and $17n^{\log_2(3)} - 16n = 195$ so the recursive algorithm actually performs marginally more poorly than the usual algorithm for up to 6 digit numbers. At $n = 7$ the two algorithms perform the same with $f(n) = g(n) = 259$. This suggests a hybrid approach for the problem. For numbers smaller than 8 digits the normal multiplication algorithm can be used and the recursive algorithm is used otherwise. Such algorithms which solve problem instances different ways depending on the instance are called adaptive.

Table 6.2: compared values for $f(n) = 6n^2 - 5n$ and $g(n) = 17n^{\log_2(3)} - 16n$

n	$f(n)$	$g(n)$
1	1	1
2	14	19
3	39	49
4	76	89
5	125	138
6	186	195
7	259	259
8	344	331

6.6.2 Strassens Matrix Multiplication

Here we introduce another algebraic manipulation which improves on the order of square matrix multiplication. For square $n \times n$ matrices $A = (a_{ij})_{i,j=1}^n$, $B = (b_{ij})_{i,j=1}^n$, the product $C = (c_{ij})_{i,j=1}^n$ is defined by $c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$. As each of the $n \times n$ entries c_{ij} requires n multiplications, the order of square matrix multiplication is n^3. Now consider matrices for which $n = 2^k$. Each can be partitioned into 4 submatrices or minors and the matrix product can be performed by working with the minor matrices. Denoting by subscripted capitals the four minors we have

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

where

$$A_{11} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, A_{12} = \begin{pmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{pmatrix}, A_{21} = \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix},$$

$$A_{22} = \begin{pmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix}.$$

Carrying out the multiplication using minors we have

$$C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix},$$

that is,

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22}. \end{aligned}$$

Having seen how to reduce a problem of size n into 8 subproblems of size $\frac{n}{2}$ we naturally investigate a recursive strategy. If $T(n)$ is the cost of the original multiplication for size n matrices, then for each of the eight minor multiplications the cost is given by $T(\frac{n}{2})$. Since the addition of minors involves $\left(\frac{n}{2}\right)^2$ component additions, the time for the recursive strategy is given by

$$T(n) = \begin{cases} 4\left(\frac{n}{2}\right)^2 + 8T\left(\frac{n}{2}\right) & ,n > 1 \\ 1 & ,n = 1 \end{cases}$$
$$= \begin{cases} n^2 + 8T\left(\frac{n}{2}\right) & ,n > 1 \\ 1 & ,n = 1. \end{cases}$$

We proceed to solve using constructive induction.

$$\begin{aligned}
T(n) &= n^2 + 8T\left(\frac{n}{2}\right) \\
&= n^2 + 8\left[\left(\frac{n}{2}\right)^2 + 8T\left(\frac{n}{2^2}\right)\right] \\
&= n^2[1+2] + 8^2T\left(\frac{n}{2^2}\right) \\
&= n^2[1+2] + 8^2\left[\left(\frac{n}{2^2}\right)^2 + 8T\left(\frac{n}{2^3}\right)\right] \\
&= n^2[1+2+2^2] + 8^3T\left(\frac{n}{2^3}\right) \\
&\vdots \\
&= n^2\sum_{i=0}^{k-1} 2^i + 8^kT\left(\frac{n}{2^k}\right) \\
&= n^2(2^k - 1) + 8^k, \qquad \frac{n}{2^k} = 1 \\
&= n^2(n-1) + (2^k)^3 \\
&= 2n^3 - n^2
\end{aligned}$$

So again, a recursive approach alone does not yield any improvement over the straight forward approach. Here too however, there are ways to make use of rearrangements of sums and products to favor the number of sums over the number of products. Let 7 new matrices be defined by

$$\begin{aligned}
D_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
D_2 &= (A_{21} + A_{22})B_{11} \\
D_3 &= A_{11}(B_{12} - B_{22}) \\
D_4 &= A_{22}(B_{21} - B_{11}) \\
D_5 &= (A_{11} + A_{12})B_{22} \\
D_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
D_7 &= (A_{12} - A_{22})(B_{21} + B_{22}).
\end{aligned}$$

Then C can be computed using no further matrix multiplications with

 ©2003 Steven Louis Davis

$$\begin{aligned}
C_{11} &= D_1 + D_4 - D_5 + D_7, \\
C_{12} &= D_3 + D_5, \\
C_{21} &= D_2 + D_4, \text{and} \\
C_{22} &= D_1 + D_3 - D_2 + D_6.
\end{aligned}$$

As each of the new matrices has only one matrix multiplication we now have the time given by

$$\begin{aligned}
T(n) &= \begin{cases} 18\left(\frac{n}{2}\right)^2 + 7T(\frac{n}{2}) & ,n > 1 \\ 1 & ,n = 1. \end{cases} \\
&= \begin{cases} \frac{9}{2}n^2 + 7T(\frac{n}{2}) & ,n > 1 \\ 1 & ,n = 1. \end{cases}
\end{aligned}$$

Using Theorem 6.4.1 we may solve the simpler recurrence

$$T(n) = \begin{cases} n^2 + 7T(\frac{n}{2}) & ,n > 1 \\ 1 & ,n = 1. \end{cases}.$$

Solving we have

$$
\begin{aligned}
T(n) &= n^2 + 7T\left(\frac{n}{2}\right) \\
&= n^2 + 7\left[\left(\frac{n}{2}\right)^2 + 7T\left(\frac{n}{2^2}\right)\right] \\
&= n^2\left[1 + \frac{7}{4}\right] + 7^2 T\left(\frac{n}{2^2}\right) \\
&= n^2\left[1 + \frac{7}{4}\right] + 7^2\left[\left(\frac{n}{2^2}\right)^2 + 7T\left(\frac{n}{2^3}\right)\right] \\
&= n^2\left[1 + \frac{7}{4} + \left(\frac{7}{4}\right)^2\right] + 7^3 T\left(\frac{n}{2^3}\right) \\
&\vdots \\
&= n^2 \sum_{i=0}^{k-1}\left(\frac{7}{4}\right)^i + 7^k T\left(\frac{n}{2^k}\right) \\
&= n^2\left[\frac{\left(\frac{7}{4}\right)^k - 1}{\frac{7}{4} - 1}\right] + 7^k, \qquad \frac{n}{2^k} = 1 \\
&= \frac{4}{3}n^2\left[\frac{7^k}{(2^k)^2} - 1\right] + 7^k \\
&= \frac{7}{3}7^k - \frac{4}{3}n^2 \\
&= \frac{7}{3}7^{\log_2(n)} - \frac{4}{3}n^2 \\
&= \frac{7}{3}n^{\log_2(7)} - \frac{4}{3}n^2 \\
&\simeq n^{\log_2(7)}
\end{aligned}
$$

Since $\log_2(7) < 3$ we have obtained an improvement in order ($\log_2(7) \simeq 2.81$). It has been shown that an even more treacherous association of submatrices yields a system with only 5 matrix multiplications. This yields an algorithm of order $\log_2(5) \simeq 2.32$.

Chapter Exercises

E 6.1. *Write a recurrence relation for the time complexity of the algorithm:*

```
int x(int n){
if(n>1)
   return 2*x(n/2);
else
   return 1;
}
```

E 6.2. *Write a recurrence relation for the time complexity of the algorithm:*

```
int pfib(int n,int p){
if(n<p)
   return 0;
if(n==p)
   return 1;
int r=0;
for(int i=n-p;i<n;i++)
   r=r+pfib(i,p);
return r;
}
```

E 6.3. *Write a recurrence relation for the time complexity of a merge sort algorithm that splits three ways.*

Solve the recurrence exactly:

E 6.4. $f(n) = \begin{cases} 3 + f(\frac{n}{2}) & , n > 1, \\ 1 & , n = 1, \end{cases}$

E 6.5. $f(n) = \begin{cases} n + f(\frac{n}{3}) & , n > 1, \\ 1 & , n = 1, \end{cases}$

E 6.6. $f(n) = \begin{cases} 1 + 2f(\frac{n}{3}) & , n > 1, \\ 1 & , n = 1, \end{cases}$

Find the order of the recurrence:

E 6.7. $f(n) = \begin{cases} 3 + \frac{1}{n^2} + f(\frac{n}{2}) & , n > 1, \\ 1 & , n = 1, \end{cases}$

E 6.8. $f(n) = \begin{cases} n + 250 + f(\frac{n}{3}) & , n > 1, \\ 1 & , n = 1, \end{cases}$

E 6.9. $f(n) = \begin{cases} \frac{(n+1)(n+2)}{(n-1)^2} + f(\frac{n}{2}) & , n > 1, \\ 1 & , n = 1, \end{cases}$

E 6.10. *Partition the array:*

39	41	43	88	29	46	16	92	75	86	25	63	12	79	81	24

E 6.11. *Fix the heap:*

1	2	3	4	5	6	7	8	9	10	11	12	13	14
26	42	46	37	22	45	41	25	19	4	13	17	28	31

E 6.12. *Construct a 9 element array of integers on which quicksort will exhibit its worst case time. [do not use a sorted array as your example]*

III

Appendices

Appendix A

Mathematical Reference

A.1 Discrete Fourier Transform

For a principle n^{th} root of unity $\omega \neq 1$, each of $\omega^2, \omega^2, \ldots, \omega^{n-1}$ is also a principle n^{th} root of unity. Thus each is a root of the polynomial

$$\frac{x^n - 1}{x - 1} = \sum_{i=0}^{n-1} x^i,$$

that is,

$$\sum_{j=0}^{n-1} \omega^{ij} = 0, \text{ for } i = 1, 2, \ldots, n-1. \tag{A.1}$$

It is now a simple matter to demonstrate the inverse $V_\omega^{-1} = \frac{1}{n} V_{\omega^{-1}}$. In fact,

$$\begin{aligned}
(V_\omega V_{\omega^{-1}})_{ij} &= \left(\sum_{k=0}^{n-1} (V_\omega)_{ik} (V_{\omega^{-1}})_{kj} \right)_{ij} \\
&= \left(\sum_{k=0}^{n-1} \omega^{ik} \omega^{-kj} \right)_{ij} \\
&= \left(\sum_{k=0}^{n-1} \omega^{k(i-j)} \right)_{ij}.
\end{aligned}$$

By (A.1) the sum vanishes for $i \neq j$, and for $i = j$ it reduces to n.

We next demonstrate the homomorphism $\widehat{f * g} = \hat{f}\hat{g}$.

$$\begin{aligned}
\widehat{f * g} &= \left(\sum_{j=0}^{n-1} \omega^{ij} \sum_{k=0}^{n-1} f_k g_{j-k} \right)_{i=0}^{n-1} \\
&= \left(\sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \omega^{ij} f_k g_{j-k} \right)_{i=0}^{n-1} \\
&= \left(\sum_{j=0}^{n-1} \sum_{l=-k}^{n-1-k} \omega^{i(k+l)} f_k g_l \right)_{i=0}^{n-1}, \ l = j - k \\
&= \left(\sum_{j=0}^{n-1} \omega^{ik} f_k \sum_{l=-k}^{n-1-k} \omega^{il} g_l \right)_{i=0}^{n-1} \\
&= \left(\sum_{j=0}^{n-1} \omega^{ik} f_k \sum_{l=0}^{n-1} \omega^{il} g_l \right)_{i=0}^{n-1} \\
&= \hat{f}\hat{g}.
\end{aligned}$$

After inspection of the terms, the second sum is seen to be independent of k thereby justifying the reindexing.

Theorem A.1.1. *Given n, ω, both powers of 2, then ω is a principle n^{th} root of unity in $F = Z_m$ for $m = \omega^{\frac{n}{2}} + 1$, and $\frac{1}{n} = m - \frac{1}{n}(m-1)$.*

Proof. □

A.2 Binomial Coefficients

Theorem A.2.1. $(n-k)\binom{n}{k} = n\binom{n-1}{k}$.

Proof.

$$\begin{aligned}
(n-k)\binom{n}{k} &= (n-k)\frac{n!}{k!(n-k)!} \\
&= \frac{n!}{k!(n-k-1)!} \\
&= n\frac{(n-1)!}{k!(n-1-k)!} \\
&= n\binom{n-1}{k}
\end{aligned}$$

□

Theorem A.2.2. $k\binom{n}{k} = n\binom{n-1}{k-1}$.

Proof.

$$\begin{aligned}
k\binom{n}{k} &= k\frac{n!}{k!(n-k)!} \\
&= \frac{n!}{(k-1)!(n-k)!} \\
&= n\frac{(n-1)!}{(k-1)!(n-k)!} \\
&= n\binom{n-1}{k-1}
\end{aligned}$$

□

Theorem A.2.3. $(a+b)^n = \sum_{k=0}^{n}\binom{n}{k}a^{n-k}b^k$.

Proof. The formula holds trivially for $n = 0, 1$. Assuming the formula true for

n we shall prove its truth for $n+1$. We have

$$\begin{aligned}
\sum_{k=0}^{n+1}\binom{n+1}{k}a^{n+1-k}b^k &= a^{n+1}+\sum_{k=1}^{n}\binom{n+1}{k}a^{n+1-k}b^k+b^{n+1} \\
&= a^{n+1}+\sum_{k=1}^{n}\left[\binom{n}{k}+\binom{n}{k-1}\right]a^{n+1-k}b^k+b^{n+1} \\
&= a^{n+1}+\sum_{k=1}^{n}\binom{n}{k}a^{n+1-k}b^k+\sum_{j=1}^{n}\binom{n}{j-1}a^{n+1-j}b^j+b^{n+1} \\
&= a^{n+1}+\sum_{k=1}^{n}\binom{n}{k}a^{n+1-k}b^k+\sum_{k=0}^{n-1}\binom{n}{k}a^{n-k}b^{k+1}+b^{n+1} \\
&= a\left[a^n+\sum_{k=1}^{n}\binom{n}{k}a^{n-k}b^k\right]+b\left[\sum_{k=0}^{n-1}\binom{n}{k}a^{n-k}b^k+b^n\right] \\
&= (a+b)\sum_{k=0}^{n}\binom{n}{k}a^{n-k}b^k \\
&= (a+b)(a+b)^n \\
&= (a+b)^{n+1}
\end{aligned}$$

□

Corollary A.2.1. $\sum_{k=0}^{n}\binom{n}{k}=2^n$.

Proof. Using $a=b=1$ we have

$$\begin{aligned}
\sum_{k=0}^{n}\binom{n}{k} &= \sum_{k=0}^{n}\binom{n}{k}1^{n-k}1^k \\
&= (1+1)^n \\
&= 2^n
\end{aligned}$$

□

Theorem A.2.4. $\sum_{k=1}^{n}k\binom{n}{k}=n2^{n-1}$.

Proof.

$$\begin{aligned}
\sum_{k=1}^{n} k\binom{n}{k} &= \sum_{k=1}^{n} n\binom{n-1}{k-1} && \text{[by Theorem A.2.2]} \\
&= n\sum_{k=1}^{n}\binom{n-1}{k-1} \\
&= n\sum_{j=0}^{n-1}\binom{n-1}{j} && [j = k-1] \\
&= n2^{n-1} && \text{[by Theorem A.2.1]}
\end{aligned}$$

□

Theorem A.2.5. $a^b \leq \binom{ab}{b} \leq (b[a-1]+1)^b$.

Proof.

$$\begin{aligned}
\binom{ab}{b} &= \frac{(ab)!}{b!(ab-b)!} \\
&= \frac{ab(ab-1)(ab-2)...(ab-b+1)}{b!} \\
&= \frac{ab}{b}\frac{ab-1}{b-1}\frac{ab-2}{b-2}...\frac{ab-b+1}{1}
\end{aligned}$$

Each factor is bounded above by $ab - b + 1$ and below by a. □

Lemma A.2.1. *For* $s \leq r$, $\frac{r-1}{s-1} \geq \frac{r}{s}$.

Proof.

$$\begin{aligned}
-s &\geq -r, \\
rs - s &\geq rs - r, \\
s(r-1) &\geq r(s-1)
\end{aligned}$$

□

Theorem A.2.6. $2^k \leq \binom{2k}{k} \leq (k+1)^k$.

Proof.

$$\begin{aligned}
\binom{2k}{k} &= \frac{(2k)!}{(k!)^2} \\
&= \frac{2k(2k-1)(2k-2)...(k+1)}{k!} \\
&= \frac{2k}{k}\frac{2k-1}{k-1}\frac{2k-2}{k-2}...\frac{k+1}{1}
\end{aligned}$$

Now apply the lemma to each of these k factors. □

Theorem A.2.7. $k^{2k-1} \leq \binom{k(2k-1)}{2k-1} \leq (4k^3 - 4k^2 - k + 2)^{2k-1}$.

Proof. Apply the lemma. □

Theorem A.2.8 (Stirling)**.** $\left(\frac{n}{e}\right)^n \sqrt{n} \simeq n!$.

Proof. We will show that

$$\lim_{n\to\infty} \frac{\left(\frac{n}{e}\right)^n \sqrt{n}}{n!} = \frac{1}{\sqrt{2\pi}}.$$

First,

$$\begin{aligned}
\int_0^{\frac{\pi}{2}} \sin^n(x)\,\mathrm{d}x &= -\cos(x)\sin^{n-1}(x)\Big|_0^{\frac{\pi}{2}} + (n-1)\int_0^{\frac{\pi}{2}} \sin^{n-2}(x)\cos^2(x)\,\mathrm{d}x \\
&= (n-1)\int_0^{\frac{\pi}{2}} \sin^{n-2}(x)(1-\sin^2(x))\,\mathrm{d}x \\
&= (n-1)\left(\int_0^{\frac{\pi}{2}} \sin^{n-2}(x)\,\mathrm{d}x - \int_0^{\frac{\pi}{2}} \sin^n(x)\,\mathrm{d}x\right)
\end{aligned}$$

so that for $I_n = \int_0^{\frac{\pi}{2}} \sin^n(x)\,\mathrm{d}x$ we have $I_n = (n-1)(I_{n-2} - I_n)$, and hence $I_n = \frac{n-1}{n} I_{n-2}$. Then,

$$\begin{aligned}
I_{2n} &= \frac{2n-1}{2n} I_{n-2} \\
&= \frac{(2n-1)(2n-3)}{(2n)(2n-2)} I_{2n-4} \\
&= \frac{(2n-1)(2n-3)(2n-5)}{(2n)(2n-2)(2n-4)} I_{2n-6} \\
&= \frac{(2n)(2n-1)(2n-2)(2n-3)(2n-4)(2n-5)}{(2n)^2(2n-2)^2(2n-4)^2} I_{2n-6} \\
&\vdots \\
&= \frac{(2n)!}{(n!)^2(2^2)^n} I_0 \\
&= \frac{(2n)!\pi}{(n!)^2 2^{2n+1}}
\end{aligned}$$

Also,

$$\begin{aligned}
I_{2n+1} &= \frac{2n}{2n+1} I_{2n-1} \\
&= \frac{(2n)(2n-2)}{(2n+1)(2n-1)} I_{2n-3} \\
&= \frac{(2n)(2n-2)(2n-4)}{(2n+1)(2n-1)(2n-3)} I_{2n-5} \\
&= \frac{(2n+1)(2n)(2n-1)(2n-2)(2n-3)(2n-4)}{(2n+1)^2(2n-1)^2(2n-3)^2} I_{2n-5} \\
&\vdots \\
&= \frac{2^{2n}(n!)^2}{(2n+1)!} I_1 \\
&= \frac{2^{2n}(n!)^2}{(2n+1)!}
\end{aligned}$$

Now, for $x \in [0, \frac{\pi}{2}]$, $\sin(x) \in [0, 1]$, and therefore,

$$I_{2n+1} \leq I_{2n} \leq I_{2n-1}.$$

That is,

$$\frac{2^{2n}(n!)^2}{(2n+1)!} \leq \frac{(2n)!\pi}{(n!)^2 2^{2n+1}} \leq \frac{2^{2(n-1)}((n-1)!)^2}{(2(n-1)+1)!}.$$

From the first inequality we obtain

$$\frac{(2^{2n})^2(n!)^4}{(2n)!(2n+1)!} \leq \frac{\pi}{2},$$

and from the second,

$$\begin{aligned}
\frac{n\pi}{2n+1} &\leq \frac{(2^{4n-1})(n!)^4}{n(2n)!(2n-1)!} \\
&\leq \frac{(2^{4n})(n!)^4}{(2n)!(2n+1)!}.
\end{aligned}$$

Since $\frac{n\pi}{2n+1} \to \frac{\pi}{2}$, and

$$\frac{n\pi}{2n+1} \leq \frac{(2^{4n})(n!)^4}{(2n)!(2n+1)!} \leq \frac{\pi}{2},$$

it follows that

$$\lim_{n\to\infty} \frac{(2^{4n})(n!)^4}{(2n)!(2n+1)!} = \frac{\pi}{2}.$$

Finally, let

$$a = \lim_{n\to\infty} \frac{\left(\frac{n}{e}\right)^n \sqrt{n}}{n!}. \tag{A.2}$$

Then we also have

$$a = \lim_{n\to\infty} \frac{\left(\frac{2n}{e}\right)^{2n} \sqrt{2n}}{(2n)!},$$

and hence,

$$\begin{aligned}
\frac{1}{a^2} = \frac{a^2}{a^4} &= \frac{\left[\lim_{n\to\infty} \frac{\left(\frac{2n}{e}\right)^{2n}\sqrt{2n}}{(2n)!}\right]^2}{\left[\lim_{n\to\infty} \frac{\left(\frac{n}{e}\right)^{n}\sqrt{n}}{n!}\right]^4} \\
&= \lim_{n\to\infty} \frac{\left[\frac{\left(\frac{2n}{e}\right)^{2n}\sqrt{2n}}{(2n)!}\right]^2}{\left[\frac{\left(\frac{n}{e}\right)^{n}\sqrt{n}}{n!}\right]^4} \\
&= \lim_{n\to\infty} \frac{\left(\frac{2n}{e}\right)^{4n}(2n)(n!)^4}{\left(\frac{n}{e}\right)^{4n} n^2((2n)!)^2} \\
&= \lim_{n\to\infty} \frac{2^{4n+1}(n!)^4}{n((2n)!)^2} \\
&= \lim_{n\to\infty} \frac{2^{4n}(n!)^2 2(2n+1)}{n(2n)!(2n+1)!} \\
&= \left(\lim_{n\to\infty} \frac{2^{4n}(n!)^2}{(2n)!(2n+1)!}\right)\left(\lim_{n\to\infty} \frac{2(2n+1)}{n}\right) \\
&= 2\pi
\end{aligned}$$

Thus $a = \frac{1}{\sqrt{2\pi}}$. The careful reader may have noticed that we have used the limit A.2 without actually proving its existence! □

Theorem A.2.9. *For $a < e, n! \prec \sqrt{n}\left(\frac{n}{a}\right)^n$, and $n! \succ \sqrt{n}\left(\frac{n}{a}\right)^n$, for $a > e$.*

Proof.

$$\begin{aligned}
\lim_{n\to\infty} \frac{n!}{\sqrt{n}\left(\frac{n}{a}\right)^n} &= \lim_{n\to\infty} \frac{\sqrt{n}\left(\frac{n}{e}\right)^n}{\sqrt{n}\left(\frac{n}{a}\right)^n} \\
&= \lim_{n\to\infty} \left(\frac{a}{e}\right)^n
\end{aligned}$$

□

Theorem A.2.10. $\sum_{i=0}^{n-m} \frac{1}{i!} \in [1, e), \qquad 0 \le m \le n.$

Proof. For $m = 0$, $\sum_{i=0}^{n-m} \frac{1}{i!} = \sum_{i=0}^{n} \frac{1}{i!} \nearrow_n e$. □

Theorem A.2.11. $\sum_{k=0}^{n-m} \binom{n-m}{k} k! \preceq (n-m)!$.

Proof.

$$\begin{aligned}\sum_{k=0}^{n-m} \binom{n-m}{k} k! &= (n-m)! \sum_{k=0}^{n-m} \frac{1}{(n-m-k)!} \\ &= (n-m)! \sum_{i=0}^{n-m} \frac{1}{i!}, \qquad i = n-m-k\end{aligned}$$

□

A.3 More Sums

Theorem A.3.1. $\sum_{i=1}^{m} i^3 = \left[\frac{m(m+1)}{2}\right]^2$

Proof. □

Use Lemma 1.7.1 and Theorem 1.7.2

Theorem A.3.2. $\sum_{i=1}^{m} ia^i = \frac{a}{(a-1)^2}\left([m(a-1)-1]a^m+1\right)$

Proof. Use Lemma 1.7.1 and Theorem 1.7.3 □

A.3.1 Polygeometric Sums

A simple technique is used to derive sums of the form $\sum_{i=1}^{n} i^m a^i$ for $m > 0$ beginning from the simple geometric sum $\sum_{i=1}^{n} a^i$.

Lemma A.3.1.

$$\sum_{i=1}^{n} a^i = \frac{a^{n+1}-a}{a-1}$$

Lemma A.3.2. *For $m > 0$,*

$$\sum_{i=1}^{n} i^m a^i = a\frac{d}{da}\left[\sum_{i=1}^{n} i^{m-1}a^i\right]$$

Proof.

$$\begin{aligned}
\sum_{i=1}^{n} i^m a^i &= \sum_{i=1}^{n}(i^{m-1}a)ia^{i-1} \\
&= \sum_{i=1}^{n}(i^{m-1}a)\frac{d}{da}[a^i] \\
&= a\sum_{i=1}^{n} i^{m-1}\frac{d}{da}[a^i] \\
&= a\sum_{i=1}^{n}\frac{d}{da}[i^{m-1}a^i] \\
&= a\frac{d}{da}\left[\sum_{i=1}^{n} i^{m-1}a^i\right]
\end{aligned}$$

□

Denoting by S_m the sum $\sum_{i=1}^{n} i^m a^i$ and noting that $S_0 = \frac{a^{n+1}-a}{a-1}$, one might attempt a general formula for S_m based on repeated application of the formal differential operator $a\frac{d}{da}$ as this appears redilly obtainable.

To illustrate:

$$
\begin{aligned}
S_m &= a\frac{d}{da}[S_{m-1}] \\
&= a\frac{d}{da}\left[a\frac{d}{da}[S_{m-2}]\right] \\
&= \left(a\frac{d}{da}\right)^{[2]}[S_{m-2}] \\
&= \left(a\frac{d}{da}\right)^{[3]}[S_{m-3}] \\
&\vdots \\
&= \left(a\frac{d}{da}\right)^{[m]}[S_0] \\
&= \left(a\frac{d}{da}\right)^{[m]}\left[\frac{a^{n+1}-a}{a-1}\right]
\end{aligned}
$$

This however is not of immediate practical use as the composition $\left(a\frac{d}{da}\right)^{[m]}$ becomes quite a complex differential operator as m grows. One can see the first four of these in the following illustration:

$$
\begin{aligned}
S_m &= \left(a\frac{d}{da}\right)[S_{m-1}] \\
&= \left(a\frac{d}{da} + a^2\frac{d^2}{da^2}\right)[S_{m-2}] \\
&= \left(a\frac{d}{da} + a(a+2)\frac{d^2}{da^2} + a^2\frac{d^3}{da^3}\right)[S_{m-3}] \\
&= \left(a\frac{d}{da} + a(3a+4)\frac{d^2}{da^2} + a^2(a+5)\frac{d^3}{da^3} + a^3\frac{d^4}{da^4}\right)[S_{m-4}]
\end{aligned}
$$

One may thus be contented with applying lemma A.3.2 directly to derive the first few poly-geometric sums.

Theorem A.3.3.

$$\sum_{i=1}^{n} ia^i = \frac{a}{(a-1)^2}\left([n(a-1)-1]a^n+1\right)$$

Proof.

$$\begin{aligned}
\sum_{i=1}^{n} ia^i &= a\frac{d}{da}\left[\sum_{i=1}^{n} a^i\right] \\
&= a\frac{d}{da}\left[\frac{a^{n+1}-a}{a-1}\right] \\
&= a\left(\frac{[a-1][(n+1)a^n-1]-[a^{n+1}-a]}{[a-1]^2}\right) \\
&= \frac{a}{(a-1)^2}\left([a-1][n+1]a^n-[a-1]-a^{n+1}+a\right) \\
&= \frac{a}{(a-1)^2}\left([a-1][n+1]a^n+1-a^{n+1}\right) \\
&= \frac{a}{(a-1)^2}\left([(a-1)(n+1)-a]a^n+1\right) \\
&= \frac{a}{(a-1)^2}\left([(a-1)n+a-1-a]a^n+1\right) \\
&= \frac{a}{(a-1)^2}\left([(a-1)n-1]a^n+1\right) \\
&= \frac{a}{(a-1)^2}\left([n(a-1)-1]a^n+1\right)
\end{aligned}$$

□

Theorem A.3.4.

$$\sum_{i=1}^{n} i^2a^i = \frac{a}{(a-1)^3}\left(\left[(n[a-1]-1)^2+a\right]a^n-a-1\right)$$

Proof.

$$\begin{aligned}
\sum_{i=1}^{n} i^2a^i &= a\frac{d}{da}\left[\sum_{i=1}^{n} ia^i\right] \\
&= a\frac{d}{da}\left[\frac{a}{(a-1)^2}\left([n(a-1)-1]\,a^n+1\right)\right] \\
&\vdots \\
&= \frac{a}{(a-1)^3}\left(\left[(n[a-1]-1)^2+a\right]a^n-a-1\right)
\end{aligned}$$

□

Theorem A.3.5.

$$\sum_{i=1}^{n} i^3 a^i = \frac{a}{(a-1)^4} \left(\left[n^3a^3 - (3n^3+3n^2-3n+1)a^2 + (3n^3+6n^2-4)a - (n+1)^3 \right] a^n + a^2 + 4a + 1 \right)$$

The author has yet to find the time to compute the formula for $\sum_{i=1}^{n} i^4 a^i$, the formula for $\sum_{i=1}^{n} i^3 a^i$ having been computed at the cost of several hours of algebraic manipulation. At this point one might desire to employ a symbolic language such as scheme to automate the differentiation and simplification process.

The first two poly-geometric sums appear regularly in the analysis of recurrences deriving from computer algorithms, usually in the form $\sum_{i=1}^{n} i^m 2^i$, that is with the base $a = 2$.

Corollary A.3.1.

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

Corollary A.3.2.

$$\sum_{i=1}^{n} i^2 2^i = \left((n-1)^2 + 2\right) 2^{n+1} - 6$$

Corollary A.3.3.

$$\sum_{i=1}^{n} i^3 2^i = \left((n-1)^3 + 6n - 14\right) 2^{n+1} + 26$$

A.4 Limits of Sums

Theorem A.4.1 (Geometric Series)**.** *For* $|a| < 1$, $\lim\limits_{n\to\infty} \sum_{i=0}^{m} a^i = \frac{1}{1-a}$.

Proof. Using Theorem 1.7.3 we have

$$\begin{aligned} \lim_{n\to\infty} \sum_{i=0}^{m} a^i &= \lim_{n\to\infty} \frac{a^{m+1}-1}{a-1} \\ &= \frac{-1}{a-1} \end{aligned}$$

□

Theorem A.4.2. $\sum_{i=0}^{m} \frac{a^i}{i!} \longrightarrow e^a$.

Proof. This is the statement of convergence of the Taylor Series for e^a expanded about 0, also called a McLauren Series. □

Corollary A.4.1. $\sum_{i=0}^{m} \frac{1}{i!} \longrightarrow e$.

Theorem A.4.3 (Harmonic Series)**.** $\sum_{i=1}^{m} \frac{1}{i} \longrightarrow \infty$.

Proof.

$$\begin{aligned} \sum_{i=1}^{2^k} \frac{1}{i} &= 1 + \sum_{i=2}^{2^k} \frac{1}{i} \\ &\geq 1 + \sum_{j=0}^{k} 2^j \frac{1}{2^{j+1}} \\ &= 1 + \sum_{j=0}^{k} \frac{1}{2} \\ &= 1 + \frac{k+1}{2} \end{aligned}$$

Thus

$$\begin{aligned} \lim_{n\to\infty} \sum_{i=1}^{m} \frac{1}{i} &= \lim_{n\to\infty} \sum_{i=1}^{2^k} \frac{1}{i} \\ &\geq \lim_{n\to\infty} 1 + \frac{k+1}{2} \\ &= \infty \end{aligned}$$

□

Theorem A.4.4 (Alternating Harmonic Series)**.** $\sum_{i=1}^{m} \frac{(-1)^{i+1}}{i} \longrightarrow \ln(2)$.

Proof.

□

Theorem A.4.5. $\ln(m+1) < \sum_{i=1}^{m} \frac{1}{i} < \ln(m) + 1$.

Proof.

□

Appendix B

General Solution of Elementary Recurrences

Functional iteration is employed in algorithm analysis to characterize and solve recursively defined functions, where it takes on a decidedly less analytical flavor since there is no true limiting behavior, the number of iterations always being finite. None the less it proves to be an interesting subject for general investigation yielding some important results in the theory of order classification. Here a more direct approach to the so called 'master method' of solving elementary recurrence equations, those of the form $f(n) = g(n) + af(n/b)$, is given in a new and much more general context. The result is extended to all functions of the form $f(n) = g(n) + h(n)f(\beta(n))$, where β can represent almost any function used to yield diminished arguments. The result is then used to justify the simplification of g for the purpose of order classification of f.

B.1 Terminal Compositions

iterated composition

Denote the m-fold application of a function $\beta(n)$ by $\beta^{[m]}(n)$, that is,

$$\beta^{[m]}(n) = \beta(\beta(\beta(...\beta(n)...))), m - 1 \text{ compositions of } \beta.$$

More formally,

Definition B.1.1.

$$\beta^{[m]}(n) = \begin{cases} n & , m = 0 \\ \beta(\beta^{[m-1]}(n)) & , m \geq 1 \end{cases}$$

iterated composition of contractions

Let β be defined on $\mathcal{N}^+ = \{1, 2, 3, ...\}$ (the subset of the natural numbers $\mathcal{N} = \{0, 1, 2, 3, ...\}$ not including 0) with values also in $\mathcal{N}^+$. β is called a dilation if $\beta(n) > n$ for all n, and a contraction if $\beta(n) < n$ for all $n > 1$. For a contraction $\beta(1) = 1$ is required.

The remainder is concerned only with contractions and the behavior of their iterates. These play an important role in the description of some of the extreme representatives of the order hierarchy and also in the definition of recurrences. Two of the most common examples encountered within recurrences are given by:

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

and

$$\beta(n) = \begin{cases} \lfloor \frac{n}{2} \rfloor & , n > 1, \\ 1 & , n = 1. \end{cases}$$

degree of terminal composition

Of particular interest is how many applications of β are required to yield the base value $n_0 = 1$, as reaching the base value terminates the nontrivial behavior of iterates[1]. Any expression of the form $\beta^{[i]}(n) = n_0$ is referred to as a terminal composition. For the first terminal composition $\beta^{[i]}(n)$ the number i of applications is called degree of terminal composition.

The symbol $\beta^*(n)$ will be used to denote this minimal number of applications:

$$\beta^*(n) = \min \{i \geq 0 : \beta^{[i]}(n) = n_0\}$$

For the examples above $\beta^*(n) = n - 1$ and $\beta^*(n) = log_2(n)$[2], respectively. For the first of these all integer values from 1 to $n - 1$ are assumed by the sequence of iterates

$$\begin{aligned} \beta(n) &= n - 1, \\ \beta^{[2]}(n) &= n - 2, \\ \beta^{[3]}(n) &= n - 3, \\ &\vdots \\ \beta^{[n-1]}(n) &= 1. \end{aligned}$$

For the second however the sequence has gaps and only for n in the lattice $\mathcal{N}_\beta$ of inverse compositions,

$$\mathcal{N}_\beta = \{1, \beta^{[-1]}(1), \beta^{[-2]}(1), \ldots, \},$$

[1] $(\beta^{[i]}(n))_{i=0}^{\infty}$ is a stationary sequence

[2] for n a power of 2

are exact results obtained. For convenience it is preferred to work within $\mathcal{N}_\beta$.

The previous examples are easily generalized for $b > 1$. First for

$$\beta(n) = \begin{cases} n - b & , n > b \\ 1 & , n \le b, \end{cases}$$

one has:

$$\mathcal{N}_\beta = \{1 + nb : n \ge 0\},$$

and for $n \in \mathcal{N}_\beta$,

$$\beta^*(n) = \frac{n-1}{b}.$$

Also, for

$$\beta(n) = \begin{cases} \lfloor \frac{n}{b} \rfloor & , n > b \\ 1 & , n \le b, \end{cases}$$

one has:

$$\mathcal{N}_\beta = \{b^n : n \ge 0\},$$

and for $n \in \mathcal{N}_\beta$,

$$\beta^*(n) = log_b(n).$$

Another interesting example is given by:

$$\beta(n) = \begin{cases} \lfloor log_b(n) \rfloor & , n > b \\ 1 & , n \le b \end{cases}$$

for which

$$\mathcal{N}_\beta = \{1, b, b^b, b^{b^b}, b^{b^{b^b}}, \ldots\},$$

and for $n \in \mathcal{N}_\beta$,

$$\beta^*(n) = log_b^*(n).$$

properties of terminal compositions

The following properties of β^* are evident:

$$\beta^*(1) = 0$$
$$\beta^*(\beta^{[i]}(n)) = \beta^*(n) - i$$
$$\beta^*(\beta^{[-i]}(n)) = \beta^*(n) + i$$
$$\beta^*(\beta^{[i]}(n)) = \beta^*(n) - i$$
$$\beta^*(\beta^{[\beta^*(n)-i]}(n)) = i$$
$$\beta^{[\beta^*(n)+i]}(n) = 1, \qquad \text{for } i \ge 0$$
$$\beta^{[\beta^*(n)-i]}(n) = \beta^{[-i]}(1) \tag{B.1}$$

$$\beta^{[i]}(n) = \beta^{[i-\beta^*(n)]}(1)$$

The following characterizations of the initial segments of $\mathcal{N}_\beta$ provide some flexibility in the indexing of sums:

$$\mathcal{N}_\beta \cap [1,n] = \begin{array}{l} \{\beta^{[i]}(n) : i \in \mathcal{N} \cap [0, \beta^*(n)]\} \\ \{\beta^{[\beta^*(n)-i]}(n) : i \in \mathcal{N} \cap [0, \beta^*(n)]\} \\ \{\beta^{[-i]}(1) : i \in \mathcal{N} \cap [0, \beta^*(n)]\} \end{array}$$

B.2 General Solution Of Elementary Recurrences

elementary recurrence equations

A class of simple recurrence equations commonly occurring in elementary algorithm analysis is now characterized. These are the functions f characterized completely for nonnegative nondecreasing functions g and $h \simeq 1$, and contraction β, by

$$f(n) = \begin{cases} g(n) + h(n)f(\beta(n)), & n > 1, \\ 1, & n = 1. \end{cases} \tag{B.2}$$

Theorem B.2.1. *For $n \in \mathcal{N}_\beta$,*

$$f(n) = \sum_{i=0}^{\beta^*(n)} g(\beta^{[i]}(n)) \prod_{j=0}^{i-1} h(\beta^{[j]}(n)).$$

Proof. Let $f_\beta(n)$ denote the sum defined on $\mathcal{N}_\beta$ and without loss of generality assume $g(1) = 1$. Proof is by induction on $\mathcal{N}_\beta$. Since $\beta^*(1) = 0$ it follows that

$f_\beta(1) = g(1) = f(1)$. Now assume $f_\beta(m) = f(m)$ for $m \in \mathcal{N}_\beta$. Then

$$\begin{aligned}
f_\beta(\beta^{[-1]}(m)) &= \sum_{i=0}^{\beta^*(\beta^{[-1]}(m))} g(\beta^{[i]}(\beta^{[-1]}(m))) \prod_{j=0}^{i-1} h(\beta^{[j]}(\beta^{[-1]}(m))) \\
&= g(\beta^{[-1]}(m)) + \sum_{i=1}^{\beta^*(m)+1} g(\beta^{[i-1]}(m)) \prod_{j=0}^{i-1} h(\beta^{[j-1]}(m)) \\
&= g(\beta^{[-1]}(m)) + \sum_{k=0}^{\beta^*(m)} g(\beta^{[k]}(m)) \prod_{j=0}^{k} h(\beta^{[j-1]}(m)) \qquad (k = i-1) \\
&= g(\beta^{[-1]}(m)) + h(\beta^{[-1]}(m)) \sum_{k=0}^{\beta^*(m)} g(\beta^{[k]}(m)) \prod_{j=1}^{k} h(\beta^{[j-1]}(m)) \\
&= g(\beta^{[-1]}(m)) + h(\beta^{[-1]}(m)) \sum_{k=0}^{\beta^*(m)} g(\beta^{[k]}(m)) \prod_{p=0}^{k-1} h(\beta^{[p]}(m)) \qquad (p = j-1) \\
&= g(\beta^{[-1]}(m)) + h(\beta^{[-1]}(m)) f_\beta(m) \\
&= g(\beta^{[-1]}(m)) + h(\beta^{[-1]}(m)) f(m) \\
&= f(\beta^{[-1]}(m))
\end{aligned}$$

□

alternate forms for f_β

In the sequel it is convenient to denote the product $\prod_{j=0}^{i-1} h(\beta^{[j]}(n))$ by $H_i(n)$. Observe that the sum above includes the empty product $H_0(n) \equiv 1$. A simple change of variable together with B.1 provides another useful form:

$$\begin{aligned}
f_\beta(n) &= \sum_{i=0}^{\beta^*(n)} g(\beta^{[i]}(n)) H_i(n) \\
&= \sum_{i=0}^{\beta^*(n)} g(\beta^{[\beta^*(n)-i]}(n)) H_{\beta^*(n)-i}(n) \qquad \text{(B.3)} \\
&= \sum_{i=0}^{\beta^*(n)} g(\beta^{[-i]}(1)) H_{\beta^*(n)-i}(n)
\end{aligned}$$

manipulations of $H_i(n)$

Lemma B.2.1. $H_i(n) = h(\beta^{[i-1]}(n)) H_{i-1}(n)$.

Lemma B.2.2. $H_i(n) = \frac{h(\beta^{[i-1]}(n))}{h(\beta^{[-1]}(n))} H_i(\beta^{[-1]}(n))$.

Proof.

$$\begin{aligned}
h(\beta^{[-1]}(n))H_i(n) &= h(\beta^{[-1]}(n)) \prod_{j=0}^{i-1} h(\beta^{[j]}(n)) \\
&= h(\beta^{[-1]}(n)) \prod_{j=0}^{i-2} h(\beta^{[j]}(n))h(\beta^{[i-1]}(n)) \\
&= h(\beta^{[-1]}(n)) \prod_{k=1}^{i-1} h(\beta^{[k-1]}(n))h(\beta^{[i-1]}(n)) \\
&= \prod_{k=0}^{i-1} h(\beta^{[k-1]}(n))h(\beta^{[i-1]}(n)) \\
&= \prod_{k=0}^{i-1} h(\beta^{[k]}(\beta^{[-1]}(n)))h(\beta^{[i-1]}(n)) \\
&= H_i(\beta^{[-1]}(n))h(\beta^{[i-1]}(n))
\end{aligned}$$

□

Lemma B.2.3. $H_i(\beta^{[-1]}(n)) = h(\beta^{[-1]}(n))H_{i-1}(n) = \frac{h(\beta^{[-1]}(n))}{h(\beta^{[i-1]}(n))}H_i(n)$.

Lemma B.2.4. $H_{i-1}(\beta^{[-1]}(n)) = \frac{h(\beta^{[-1]}(n))}{h(\beta^{[i-2]}(n))}H_{i-1}(n) = \frac{h(\beta^{[-1]}(n))}{h(\beta^{[i-1]}(n))h(\beta^{[i-2]}(n))}H_i(n)$.

simple bounds

Before making use of theorem B.2.1 it is necessary to justify the use of the subsequential lattice $\mathcal{N}_\beta$ in characterizing the order of f.

Rationally Bounded Functions

Definition B.2.1. *A non-decreasing function f is rationally bounded on a sequence $(n_i)_{i=i_0}^{\infty}$ if there is a positive number $b > 0$ such that for all i, $f(n_{i+1}) \leq bf(n_i)$.*

Rationally bounded functions on geometric sequences are sometimes referred to as 'smooth'. If the sequence is understood reference is simply made to the function as 'rationally bounded'. In particular, for f given by B.2, f is understood to be rationally bounded on the sequence $\mathcal{N}_\beta$. Most often of concern are asymptotically rationally bounded functions which are defined with the usual asymptotic extension. As with other asymptotic properties these are often referred to as rationally bounded 'eventually'. It should be clear that both sum and product of rationally bounded functions are rationally bounded. Also for any contraction β, β^* is rationally bounded.

Lemma B.2.5. *For nondecreasing rationally bounded f on $\{1, c, c^2, c^3, ...\}$, $c > 1$, and any integer $k \geq 1$, $f(n+k) \preceq f(n)$.*

Proof. For $n > \lceil \frac{k}{c-1} \rceil$ find i such that $c^{i-1} \leq n < c^i$. Since $nc > n + k$ one has $f(n+k) \leq f(nc) \leq f(c^{i+1}) \leq b^2 f(c^{i-1}) \leq b^2 f(n)$. □

Lemma B.2.6. *Let f, g be eventually non-decreasing functions. If g is eventually rationally bounded on a sequence $(n_i)_i$ and $f(n_i) \preceq g(n_i)$ (f and g are 'interlaced'), then $f \preceq g$.*

Proof. Without loss of generality assume f and g are non-decreasing, that g is rationally bounded, and that there is $c_1 > 0$ such that $f(n_i) \leq c_1 g(n_i)$. Choose $c_2 > 0$ such that $g(n_{i+1}) \leq c_2 g(n_i)$. Given n, find i_0 such that $n_{i_0} \leq n < n_{i_0+1}$. Then $f(n) \leq f(n_{i_0+1}) \leq c_1 g(n_{i_0+1}) \leq c_1 c_2 g(n_{i_0}) \leq c_1 c_2 g(n)$. □

Lemma B.2.7. *For $h \simeq 1$, f_β is eventually rationally bounded on $\mathcal{N}_\beta$.*

Proof. Let $c > 0$ such that $h(n) \geq c$ for large n. Then

$$\begin{aligned} f_\beta(\beta(n)) &= \frac{f_\beta(n) - g(n)}{h(n)} \\ &\leq \frac{1}{c_2} f_\beta(n). \end{aligned}$$

□

Theorem B.2.2. *For f given by B.2 with $h \simeq 1$, $f \simeq f_\beta$.*

The crudest of bounds are immediately obtained by bounding the terms and factors appearing in f_β.

Theorem B.2.3. *For $h = 1$, $\beta^* g(\beta^{[\frac{1}{2}\beta^*]}) \preceq f \preceq \beta^* g$.*

Proof.

$$\begin{aligned} f_\beta(n) &= \sum_{i=0}^{\beta^*(n)} g(\beta^{[i]}(n)) \\ &\leq (1 + \beta^*(n)) g(n) \end{aligned}$$

and,

$$\begin{aligned} f_\beta(n) &\geq \sum_{i=0}^{\frac{1}{2}\beta^*(n)} g(\beta^{[i]}(n)) \\ &\geq (1 + \frac{1}{2}\beta^*(n)) g(\beta^{[\frac{1}{2}\beta^*(n)]}(n)) \end{aligned}$$

□

Theorem B.2.4. *For $h \not\simeq 1, g(\beta^{[\frac{1}{2}\beta^*]} h^{\frac{1}{2}\beta^*}(\beta^{[\frac{1}{2}\beta^*]})) \preceq f \preceq g h^{\beta^*}$.*

Proof.

$$\begin{aligned}
f_\beta(n) &\le g(n) \sum_{i=0}^{\beta^*(n)} H_i(n) \\
&\le g(n) \sum_{i=0}^{\beta^*(n)} h^i(n) \\
&= g(n) \frac{h^{1+\beta^*(n)}(n) - 1}{h(n) - 1}
\end{aligned}$$

Also,

$$\begin{aligned}
f_\beta(n) &\ge \sum_{i=0}^{\frac{1}{2}\beta^*(n)} g(\beta^{[i]}(n)) H_i(n) \\
&\ge g(\beta^{[\frac{1}{2}\beta^*(n)]}(n)) \sum_{i=0}^{\frac{1}{2}\beta^*(n)} H_i(n) \\
&\ge g(\beta^{[\frac{1}{2}\beta^*(n)]}(n)) \sum_{i=0}^{\frac{1}{2}\beta^*(n)} h^i(\beta^{[i-1]}(n)) \\
&\ge g(\beta^{[\frac{1}{2}\beta^*(n)]}(n)) \sum_{i=0}^{\frac{1}{2}\beta^*(n)} h^i(\beta^{[\frac{1}{2}\beta^*(n)]}(n)) \\
&\ge g(\beta^{[\frac{1}{2}\beta^*(n)]}(n)) \frac{h^{1+\frac{1}{2}\beta^*(n)}(\beta^{[\frac{1}{2}\beta^*(n)]})(n) - 1}{h(\beta^{[\frac{1}{2}\beta^*(n)]}(n)) - 1}
\end{aligned}$$

□

Theorem B.2.5. $f \succeq h^{\beta^*}(\beta^{[-1]}(1))$.

Proof.

$$\begin{aligned}
f_\beta(n) &\ge g(1) \sum_{i=0}^{\beta^*(n)} \prod_{j=0}^{i-1} h(\beta^{[j]}(n)) \\
&\ge g(1) \sum_{i=0}^{\beta^*(n)} h^i(\beta^{[i-1]}(n)) \\
&= g(1) \sum_{i=0}^{\beta^*(n)} h^i(\beta^{[i-1-\beta^*(n)]}(1)) \\
&\ge g(1) \sum_{i=0}^{\beta^*(n)} h^i(\beta^{[-1]}(1)) \\
&= g(1) \frac{h^{1+\beta^*(n)}(\beta^{[-1]}) - 1}{h(\beta^{[-1]}(1)) - 1}
\end{aligned}$$

□

familiar results

The following corollaries can usually be found in algorithm analysis text books near theorems bearing names such as "Master Theorem", or "Master Method". They all concern recurrences of the form

$$f(n) = g(n) + bf\left(\frac{n}{c}\right).$$

Since the recursive term $bf\left(\frac{n}{c}\right)$ is arguably the most general characteristic of divide and conquer algorithms, these deserve special attention.

Corollary B.2.1. *For constant $h(n) = b > 0$,*

$$\begin{aligned} f_\beta(n) &= \sum_{i=0}^{\beta^*(n)} b^i g(\beta^{[i]}(n)) \\ &= b^{\beta^*(n)} \sum_{i=0}^{\beta^*(n)} \frac{g(\beta^{[-i]}(1))}{b^i}. \end{aligned}$$

Corollary B.2.2. *For constant $h(n) = b > 0$, and $\beta(n) = \frac{n}{c}$, for $c > 1$,*

$$f(n) \preceq \begin{cases} g(n)log_c(n), & b = 1 \\ g(n)n^{log_c(b)}, & b \neq 1 \end{cases}$$

Proof.

$$\begin{aligned} f_\beta(n) &= \sum_{i=0}^{log_c(n)} b^i g\left(\frac{n}{c^i}\right) \\ &\leq g(n) \sum_{i=0}^{log_c(n)} b^i \end{aligned}$$

For $b = 1$ the sum reduces to $1 + log_c(n)$ and otherwise to $\frac{b\log_c(n)-1}{b-1}$. □

We next derive one particular example of these recurrences.

Example B.1. *For $f(n) = n^p + bf(\frac{n}{c}), f(n) \simeq \begin{cases} n^{\log_c(b)} & ,b > c^p, \\ n^p \log_c(n) & ,b = c^p, \\ n^p & ,b < c^p \end{cases}$ From*

Corollary B.2.1 and for $b \neq c^p$ we have

$$
\begin{aligned}
f(n) &\simeq \sum_{i=0}^{\log_c(n)} b^i \left(\frac{n}{c^i}\right)^p \\
&= n^p \sum_{i=0}^{\log_c(n)} \left(\frac{b}{c^p}\right)^i \\
&= n^p \left(\frac{\left[\frac{b}{c^p}\right]^{\log_c(n)+1} - 1}{\frac{b}{c^p} - 1} \right) \\
&= \frac{c^p}{b - c^p} n^p \left(\frac{b}{c^p} n^{\log_c(\frac{b}{c^p})} - 1 \right) \\
&= \frac{c^p}{b - c^p} \left(\frac{b}{c^p} n^{\log_c(b)} - n^p \right) \\
&= \frac{b}{b - c^p} n^{\log_c(b)} + \frac{c^p}{c^p - b} n^p
\end{aligned}
$$

Now whether $b > c^p$ or $b < c^p$ determines which of the two terms is positive and therefore dominant. If $b = c^p$ then

$$
\begin{aligned}
f(n) &\simeq \sum_{i=0}^{\log_c(n)} (c^p)^i \left(\frac{n}{c^i}\right)^p \\
&= \sum_{i=0}^{\log_c(n)} n^p \\
&= n^p(\log_c(n) + 1)
\end{aligned}
$$

The trichotomy demonstrated by this example is actually characteristic of the general divide and conquer recurrence as theorem B.2.6 will demonstrate. First we need some preliminary results.

Lemma B.2.8. *For $f(n) = g(n) + bf(\frac{n}{c})$, $f(n) \simeq n^{\log_c(b)} + \sum_{i=0}^{\log_c(b)-1} b^i g(\frac{n}{c^i})$.*

Proof. Split off the last term of f_β. □

Lemma B.2.9. *For functions g_1, g_2, u, if $g_1(n) \preceq g_2(n)$, then $g_1(u(n)) \preceq g_2(u(n))$, and for any finite set I, $\sum_{i\in I} g_1(u(i)) \preceq \sum_{i \in I} g_2(u(i))$.*

The following theorem is a standard centerpiece of the theory of elementary recurrences and is often called the "Master Theorem".

Theorem B.2.6. *For $f(n) = g(n) + bf(\frac{n}{c})$,*

$$
f(n) \simeq \begin{cases} n^{\log_c(b)} & , g(n) \preceq n^{\log_c(b)-\epsilon}, \epsilon > 0, \\ n^{\log_c(b)} \log(n) & , g(n) \simeq n^{\log_c(b)}, \\ g(n) & , g(n) \succeq n^{log_c(b)+\epsilon}, \epsilon > 0, bg(\frac{n}{c}) \leq \delta g(n), \delta < 1. \end{cases}
$$

Proof. For $g(n) \preceq n^{\log_c(b)-\epsilon}$ the previous lemma gives

$$
\begin{aligned}
f(n) &\simeq n^{\log_c(b)} + \sum_{i=0}^{\log_c(n)-1} b^i g\left(\frac{n}{c^i}\right) \\
&\preceq n^{\log_c(b)} + \sum_{i=0}^{\log_c(n)-1} b^i \left(\frac{n}{c^i}\right)^{\log_c(b)-\epsilon} \\
&= n^{\log_c(b)} \left(1 + n^{-\epsilon} \sum_{i=0}^{\log_c(n)-1} \left(\frac{b}{c^{\log_c(b)-\epsilon}}\right)^i\right) \\
&= n^{\log_c(b)} \left(1 + n^{-\epsilon} \sum_{i=0}^{\log_c(n)-1} (c^\epsilon)^i\right) \\
&= n^{\log_c(b)} \left(1 + n^{-\epsilon} \left[\frac{c^{\epsilon \log_c(n)} - 1}{c^\epsilon - 1}\right]\right) \\
&= n^{\log_c(b)} \left(1 + n^{-\epsilon} \left[\frac{n^\epsilon - 1}{c^\epsilon - 1}\right]\right) \\
&\simeq n^{\log_c(b)},
\end{aligned}
$$

establishing the first part. Next, for $g(n) \simeq n^{\log_c(b)}$ we have

$$
\begin{aligned}
f(n) &\simeq \sum_{i=0}^{\log_c(n)} b^i g\left(\frac{n}{c^i}\right) \\
&= \sum_{i=0}^{\log_c(n)} b^i \left(\frac{n}{c^i}\right)^{\log_c(b)} \\
&= \sum_{i=0}^{\log_c(n)} n^{\log_c(b)} \\
&= n^{\log_c(b)}(\log_c(n) + 1)
\end{aligned}
$$

Finally assuming $g(n) \succeq n^{log_c(b)+\epsilon}, \epsilon > 0$, and that $bg(\frac{n}{c}) \leq \delta g(n)$ for some $\delta < 1$, we have $g(\frac{n}{c}) \leq \frac{d}{b} g(n), g(\frac{n}{c^2}) \leq (\frac{d}{b})^2 g(n)$, and in general, $g(\frac{n}{c^i}) \leq (\frac{d}{b})^i g(n)$,

so that $b^i g(\frac{n}{c^i}) \leq d^i g(n)$ and

$$\begin{aligned} f(n) &\preceq n^{\log_c(b)} + \sum_{i=0}^{\log_c(n)-1} b^i g\left(\frac{n}{c^i}\right) \\ &\leq \sum_{i=0}^{\log_c(n)} d^i g(n) \\ &\leq g(n) \sum_{i=0}^{\infty} d^i \\ &= g(n)\left(\frac{1}{1-d}\right) \\ &\simeq g(n), \end{aligned}$$

since $n^{\log_c(b)} \preceq g(n)$. To establish $f \succeq g$, note that the sum $\sum_{i=0}^{\log_c(n)} b^i g\left(\frac{n}{c^i}\right)$ is bounded below by its first term $g(n)$. □

The following table adds to the list of elementary recurrences given in section 6.1. Most of the order results (third column) can now be more easily derived with the aid of Theorems B.2.1 and B.2.2 even though exact solutions have been provided in most cases. The last four recurrences do not fit the hypothesis of Theorem B.2.2 and therefore no use of Theorem B.2.1 is justified for these. It is interesting to note however that the solutions for these are all predicted by Theorem B.2.1 none the less. This suggests that the hypothesis of Theorem B.2.2 could be relaxed. The last four examples in the table doubtfully correspond to any meaningful algorithms and are included only for practice sake.

B.3 Further Applications

simplification of recurrences

f_β is now employed in the simplification of g. This can dramatically reduce the difficulty of computations encountered during the order classification of f.

Consider two simple recurrences f_1 and f_2 sharing both h and β,

$$f_1(n) = \begin{cases} g_1(n) + h(n) f_1(\beta(n)), & n > 1 \\ 1, & n = 1 \end{cases}$$

$$f_2(n) = \begin{cases} g_2(n) + h(n) f_2(\beta(n)), & n > 1 \\ 1, & n = 1 \end{cases}$$

---

* $f(1) = 1$

** equality holds only for n in the lattice $\mathcal{N}_\beta$ of of inverse compositions,

$$\mathcal{N}_\beta = \{1, \beta^{[-1]}(1), \beta^{[-2]}(1), \ldots, \},$$

where $\beta(n) \in \{\log(n), \frac{n}{2}, n-1, \frac{n}{c}, n-c\}$.

Table B.1: More Recurrences

	$f(n)^*$	solution**	order
16	$1+f(\log_c(n))$	$\log_c^*(n)+1$	$\log^*(n)$
17	$n+cf(\log_c(n))$	$\log_c^*(n)(\log_c^*(n)+1)$	$(\log^*(n))^2$
18	$\log_c(n)+f(\frac{n}{c})$	$\frac{1}{2}\log_c(n)(\log_c(n)+1)+1$	$\log^2(n)$
19	$\log_c(n)+cf(\frac{n}{c})$	$\frac{1}{(c-1)^2}\left[(c^2+3c+1)n-(c-1)\log_c(n)-c)\right]$	n
20	$n\log_c(n)+f(\frac{n}{c})$	$\frac{c}{c-1}\left(n\log_c(n)+\frac{c}{(c-1)^2}(1-n)\right)+1$	$n\log(n)$
21	$n\log_c(n)+cf(\frac{n}{c})$	$\frac{1}{2}\log_c(n)(\log_c(n)+1)+n$	$n\log^2(n)$
22	$n^p+bf(\frac{n}{c})$	$\frac{b}{b-c^p}n^{log_c(b)}+\frac{c^p}{c^p-b}n^p$	$n^{max\{p,log_c(b)\}}, b\neq c^p$
23	$n^p+bf(n-c), b,c\neq 1$		$n^p(\sqrt[c]{b})^n$
24	$1+n^pf(\frac{n}{c})$	$1+n^p\sum_{i=0}^{\log_c(n)-1}\left(\frac{n}{c^{\frac{i+1}{2}}}\right)^{ip}$	$n^{\frac{1}{4}\log_c(n)}\prec f(n)\prec n^{\log_c(n)-1}$
25	$n^2+nf(n-1)$	$n!\sum_{i=0}^{n-1}\frac{i+1}{i!}$	$n!$
26	$1+n^pf(n-1)$	$(n!)^p\sum_{i=1}^{n}\frac{1}{(i!)^p}$	$(n!)^p$
27	$1+n^nf(n-1)$		$(n!)^n$

With the aid of theorems B.2.1 and B.2.2, it is easy to show that the non-strict order relationship between f_1 and f_2 is completely determined by that between g_1 and g_2. We now obtain Lemma 6.3.1 more directly.

Theorem B.3.1. *If $g_1 \preceq g_2$ then $f_1 \preceq f_2$.*

Proof. Choose $c_1 > 0$, and N such that $g_1(n) \le c_1 g_2(n)$ for $n \ge N$. Let

$$i_N = \max\{i \ge 0 : \beta^{[-i]}(1) < N\},$$
$$c_2 \ge \max\left\{c_1, \frac{\max\limits_{0\le i\le i_N} g_1(\beta^{[-i]}(1))}{\min\limits_{0\le i\le i_N} g_2(\beta^{[-i]}(1))}\right\},$$

and $\overline{N} > N$ such that $\beta^*(n) > i_N$ for $n \ge \overline{N}$. For $n \ge \overline{N}$, using B.3, one then

has

$$
\begin{aligned}
\sum_{i=0}^{i_N} g_1(\beta^{[-i]}(1))H_{\beta^*(n)-i}(n) &\leq \max_{0\leq i\leq i_N} g_1(\beta^{[-i]}(1)) \sum_{i=0}^{i_N} H_{\beta^*(n)-i}(n) \\
&\leq c_2 \min_{0\leq i\leq i_N} g_2(\beta^{[-i]}(1)) \sum_{i=0}^{i_N} H_{\beta^*(n)-i}(n) \\
&\leq c_2 \sum_{i=0}^{i_N} g_2(\beta^{[-i]}(1))H_{\beta^*(n)-i}(n)
\end{aligned}
$$

hence

$$
\begin{aligned}
(f_1)_\beta(n) &= \sum_{i=0}^{\beta^*(n)} g_1(\beta^{[-i]}(1))H_{\beta^*(n)-i}(n) \\
&= \sum_{i=0}^{i_N} g_1(\beta^{[-i]}(1))H_{\beta^*(n)-i}(n) + \sum_{i=i_N+1}^{\beta^*(n)} g_1(\beta^{[-i]}(1))H_{\beta^*(n)-i}(n) \\
&\leq c_2 \sum_{i=0}^{i_N} g_2(\beta^{[-i]}(1))H_{\beta^*(n)-i}(n) + c_1 \sum_{i=i_N+1}^{\beta^*(n)} g_2(\beta^{[-i]}(1))H_{\beta^*(n)-i}(n) \\
&\leq c_2 \sum_{i=0}^{\beta^*(n)} g_2(\beta^{[-i]}(1))H_{\beta^*(n)-i}(n) \\
&= c_2 (f_2)_\beta(n)
\end{aligned}
$$

□

Corollary B.3.1. *If* $g_1 \succeq g_2$ *then* $f_1 \succeq f_2$.

Corollary B.3.2. *If* $g_1 \simeq g_2$ *then* $f_1 \simeq f_2$.

A simple counterexample serves to show that theorem B.3.1 does not extend to a similar theorem on strict order relationship. For $g_1(n) = 1, g_2(n) = n$, and shared $h(n) = 2$, and $\beta(n) = n - 1$, f_1 and f_2 have the same order but g_1 and g_2 do not.

Appendix C

More Fibonacci Algorithms

The classical Fibonacci sequence (based on summing two terms) is defined by:

$$F_n = \begin{cases} n & , n = 0, 1, \\ F_{n-1} + F_{n-2} & , n > 1. \end{cases}$$

We have already seen that the solution obtained by implementing this recurrence directly with a recursive function is at least exponential, while the natural (dynamic programming) algorithm is first order. For better (or at least more interesting) solutions more problem analysis is in order. The direct implementation of De Moivre's formula,

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

combined with a fast (logarithmic time) algorithm for integer exponentiation, Figure 6.3, gives a logarithmic time solution, Figure C.1.

```
int dem_fib(int n){
float r5=sqrt(5),b1=(1+r5)/2,b2=(1-r5)/2;;
return (fexp(b1,n)-fexp(b2,n))/r5;
}
```

Figure C.1: De Moivre's Algorithm

Though in terms of running time this algorithm is as superior to the dynamic algorithm as the dynamic is to the recursive algorithm, one usually prefers an integer only algorithm. This still does not put a logarithmic time solution out of reach however, since it is known that any linear recurrence can be solved in logarithmic time. Again, the log time characteristic will come from an exponentiation algorithm applied this time to matrices. Any linear recurrence can

be specified by a square matrix. An order k linear recurrence has the form

$$r_n = a_k r_{n-k} + a_{k-1} r_{n-k+1} + \cdots + a_2 r_{n-2} + a_1 r_{n-1} + a_0$$

where $a_0, a_1, \ldots, a_k$ are constants. For simplicity we consider only homogeneous recurrences, that is $a_0 = 0$, with fundamental sequences [first k terms] defined by $r_{k-1} = 1$ and $r_i = 0$, for $i < k-1$. The first two Fibonacci numbers are then a fundamental sequence for the second order homogeneous linear recurrence defined by $a_1 = a_2 = 1$. To define a matrix for any such recurrence r_n, let $R = (\rho_{ij})_{i,j=1}^{k}$ where

$$\begin{aligned} \rho_{i,i+1} &= 1, i \leq k \\ \rho_{k,j} &= a_{k-j+1}, 1 \leq j < k \\ \rho_{i,j} &= 0, \text{otherwise} \end{aligned}$$

When this matrix is repeatedly applied to the vector $X = (0, 0, \ldots, 0, 1)$, we obtain a vector of k consecutive elements of the sequence generated by the recurrence.

$$\begin{aligned} R^1 X &= (0, 0, 0, \ldots, 0, 1, a_1) = (r_1, r_2, \ldots, r_k) \\ R^2 X &= (0, 0, \ldots, 0, 1, a_1, a_1 + a_2) = (r_2, r_3, \ldots, r_k, r_{k+1}) \\ R^3 X &= (0, \ldots, 0, 1, a_1, a_1 + a_2, a_2 + a_3) = (r_3, r_4, \ldots, r_k, r_{k+1}, r_{k+2}) \end{aligned}$$

and in general,

$$R^{n-k} X = (r_{n-k}, r_{n-k+1}, \ldots, r_{n-2}, r_{n-1}).$$

Since we can find r_n as the kth component of $R^{n-k+1}X$, the problem is reduced to matrix exponentiation which can be done in log time with a suitable generalization of the fast exponentiation algorithm. For the Fibonacci numbers, we use

$$R = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix},$$

and then,

$$R^{n-1} X = (F_{n-1}, F_n), n > 1.$$

We obtain the algorithm in Figure C.2
where M2X2 is a suitably defined matrix class with log time exponentiation operator $exp(int)$. We can get a measure of the running time of this algorithm by counting the number of full integer multiplications used for each of the matrix multiplications [8 for 2x2 matrices] and multiplying by the time t_m required for

```
int fast_fib(int n){
M2X2 R=(0,1,1,1);
R=R.exp(n);
return R[1][2];
}
```

Figure C.2: Fast Fibonacci Algorithm using Matrix Exponentiation

integer multiplication, the time spent on mod2, *2, and /2 operations as well as additions being insignificant. Since the *exp* method will run its loop between $\log_2(n)$ and $2\log_2(n)$ times (see Figure 6.3), we have $T(n) = C\log_2(n)$ where $8t_m \leq C \leq 16t_m$.

The following well known algorithm refines this approach by reducing the number of multiplications on average. The cost in this case seems to be algorithm clarity.

```
int obscure(int n){
int i=1,j=0,k=0,h=1,t;
while(n&gt;0){
   if(n%2){
      t=j*h;
      j=i*h+j*k+t;
      i=i*k+t;
      }
   t=h*h;
   h=2*k*h+t;
   k=k*k+t;
   n=n/2;
   }
return j;
}
```

Figure C.3: Obscure Fast Fibonacci Algorithm

Here one has about a 100 percent improvement, $3t_m \leq C \leq 7t_m$. We have illustrated remarkable improvements in speed at the expense of some analysis and code complexity. The last and most efficient algorithm in particular seems almost incomprehensible and requires some treachery to understand. It is our goal here to illustrate the derivation of an algorithm with the same order which is both simpler and more efficient, the efficiency being gained by using about half as many full multiplications. We start with the demonstration of a property of Fibonacci numbers which we call the Lucas formula. For any n,

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

This states that any evenly indexed Fibonacci number is the product of the Fibonacci and Lucas numbers at half the index value. [the sum $L_n = F_{n+1} + F_{n-1}$ is the nth Lucas number]
To see this observe that

$$\begin{aligned} F_m &= F_{m-1} + F_{m-2} \\ &= F_{m-2} + F_{m-3} + F_{m-2} \\ &= 2F_{m-2} + F_{m-3} \\ &= 2(F_{m-3} + F_{m-4}) + F_{m-3} \\ &= 3F_{m-3} + 2F_{m-4} \\ &= 3(F_{m-4} + F_{m-5}) + 2F_{m-4} \\ &= 5F_{m-4} + 3F_{m-5} \\ &= 5(F_{m-5} + F_{m-6}) + 3F_{m-5} \\ &= 8F_{m-5} + 5F_{m-6} \end{aligned}$$

One immediately notices the emergence of the Fibonacci sequence in the coefficients of the two terms in each of the reduced expressions. In fact, for $k > 1$, we have

$$F_m = F_{k+1}F_{m-k} + F_kF_{m-k-1}.$$

Letting $m = 2n$, and $k = n$ yields

$$\begin{aligned} F_{2n} &= F_{n+1}F_{2n-n} + F_nF_{2n-n-1} \\ &= F_{n+1}F_n + F_nF_{n-1} \\ &= F_n(F_{n+1} + F_{n-1}) \end{aligned}$$

We may now construct a simple Fibonacci algorithm for dyadic indices. For $n = 2^k$, we start with F_1 and apply the formula repeatedly to obtain $F_{2^1}, F_{2^2}, \ldots, F_{2^k}$. To obtain F_{2^i} however, we need a couple of Fibonacci numbers for each preceding dyadic index ± 1, that is, F_{2^i} requires $F_{[2^{i-1}]}$, and both $F_{[2^{i-1} \pm 1]}$.

Since F_{2^i} requires 3 previous elements of the sequence, it would seem that each of these requires its own 3 for a total of 9. The method employed uses the fact that having a pair of consecutive Fibonacci numbers is enough to regenerate their double-indexed successors using Lucas formula. If we have F_i and F_{i+1}, then we can obtain F_{2i} and $F_{2(i+1)}$. First use F_i and F_{i+1} to recover the neighboring numbers $F_{i-1} = F_{i+1} - F_i$ and $F_{i+2} = F_i + F_{i+1}$. Next apply Lucas formula to obtain $F_{2i} = F_i(F_{i-1} + F_{i+1})$ and $F_{2i+2} = F_{i+1}(F_i + F_{i+2})$. Finally recover the neighbor $F_{2i+1} = F_{2i+2} - F_{2i}$. More simply, it suffices to keep a set of four consecutive Fibonacci numbers y_1, y_2, y_3, and y_4 which after k updates will yield $y_2 = F_{2^k}$. At each iteration $i = 0, 1, \ldots, k, y_2$ will be the value of F_{2^i}. Given

$$
\begin{aligned}
y_1 &= F_{[2^i-1]} \\
y_2 &= F_{2^i} \\
y_3 &= F_{[2^i+1]} \\
y_4 &= F_{[2^i+2]}
\end{aligned}
$$

each iteration makes the following changes:

$$
\begin{aligned}
y_1 &= y_3 - y_2 \\
y_2 &= y_2(y_1 + y_3) = F_{[2^{i+1}]} = F_{2^i}(F_{[2^i-1]} + F_{[2^i+1]}) \\
y_3 &= y_4 - y_2 \\
y_4 &= y_3(y_2 + y_4) = F_{[2^{i+1}+2]} = F_{[2^{i+1}]}(F_{2^i} + F_{[2^i+2]})
\end{aligned}
$$

The order of computation is y_4, y_2, y_3, y_1. First y_4, and y_2 are computed using Lucas' formula and then y_3, y_1 are computed from the new values of y_4, y_2. With y_j initialized to F_j for $j = 1, 2, 3, 4$; that is, $y_1 = 1, y_2 = 1, y_3 = 2$, and $y_4 = 3$, the kth iteration produces $y_2 = F_{2^k}$. The algorithm of figure C.4 is obtained.

```
int dyadic_fibonacci(int k){ // computes F_{2^k}
int y1=1,y2=1,y3=2,y4=3;
while(k&gt;0){
   y4=y3*(y2+y4);
   y2=y2*(y1+y3);
   y3=y4-y2;
   y1=y3-y2;
   k--;
   }
return y2;
}
```

Figure C.4: Dyadic Fibonacci Algorithm

An obvious extension for generating any F_n is to generate the largest dyadic fibonacci number F_{2^k} where $2^k \leq n$ and then use the dynamic algorithm to span the gap from 2^k to n. Unfortunately this gives a linear time algorithm in the worst case because the largest dyadic index can be nearly $n/2$ requiring the dynamic algorithm to make up the other $n/2$ numbers in linear time. For example the computation of F_{2047} requires only 10 iterations of the dyadic loop to find $F_{2^{10}} = F_{1024}$ but then requires 1023 iterations of the dynamic algorithm. This is in fact the worst case example of a non-dyadic integer for the algorithm.

To successfully apply this dyadic strategy for any value of n, one would need to know the starting point p [possibly a number other than 1] and the number of iterations k required to bring p^k to the value n. For example 5 iterations applied

to $p = 3$ would be sufficient for the computation of F_{243}. The initializations of y_1, y_2, y_3, and y_4 would require $y_2 = F_3$ instead of F_2:

$$\begin{aligned} y_1 &= F_2 = 1 \\ y_2 &= F_3 = 2 \\ y_3 &= F_4 = 3 \\ y_4 &= F_5 = 5 \end{aligned}$$

We then have an algorithm to compute F_{p^k} for p-adic Fibonacci indices $n = p^k$, but of course not every integer is expressible this way. For indices in general we use a hybrid approach. Starting from an arbitrary index n we repeatedly anticipate the application of Lucas' formula from the top down. For example, to compute F_{37} one would apply Lucas' formula to obtain F_{36} from F_{18} and then shift the y variables. F_{18} can be found from F_9, F_9 from F_8 by another shift, F_8 from F_4, and F_4 from F_2. Denoting the operations of applying Lucas' formula and shifting by L and S respectively, the sequence of operations for this example would be L,L,S,L,S,L. This top down approach could be implemented by using a stack to keep track of the points where a shift is needed as does the algorithm in Figure C.5.

If, however, one closely examines the stack contents, it becomes apparent that they follow the binary encoding of n. One can therefore do away with the stack anticipator and simply mask the bits of n. The algorithm of Figure C.6 follows.

The reader will no doubt recognize some of the structure of the obscure algorithm here. The only advantage is that the number of full multiplications has now been reduced to exactly two per iteration. This represents a 33 to 70 percent improvement over the obscure solution.

```
int fib(int n){
int y1=1,y2=1,y3=2,y3=3,p=n,q;stack s;
while(p&gt;2)
   if(p%2)
      s.push(--p);
   else
      p/=2;
while(p!=n){
   if(s.empty())
      q=n;
   else
      q=s.pop();
   while(p<q){
      y4=y3*(y2+y4);
      y2=y2*(y1+y3);
      y3=y4-y2;
      y1=y3-y2;
      p*=2;
      }
   if(p==n)
      return y2;
   else{
      y1=y2;
      y2=y3;
      y3=y4;
      y4=y2+y3;
      p++;
      }
   }
return y2;
}
```

Figure C.5: Hybrid Fibonacci Algorithm using a Stack

```
int fib(int n){
int y1=0,y2=1,y3=1,y4=2,m=pow(2,floor(log(n)));
while(m<=n) m*=2; m/=2;
while(m>1){
   m/=2;
   y4=y3*(y2+y4);
   y2=y2*(y1+y3);
   y3=y4-y2;
   y1=y3-y2;
   if(m&n){
      y1=y2;
      y2=y3;
      y3=y4;
      y4=y2+y3;
      }
   }
return y2;
}
```

Figure C.6: Hybrid Fibonacci Algorithm using a mask