AN ABSTRACT OF THE THESIS OF

James L. Holloway for the degree of Master of Science in Computer Science presented on May 10, 1988.

Title: Algorithms for Computing Fibonacci Numbers Quickly

Redacted for Privacy

Abstract approved: ______________________

Paul Cull.

A study of the running time of several known algorithms and several new algorithms to compute the nth element of the Fibonacci sequence is presented. Since the size of the nth Fibonacci number grows exponentially with n, the number of bit operations, instead of the number of integer operations, was used as the unit of time. The number of bit operations used to compute f_n is reduced to less than $\frac{1}{2}$ of the number of bit operations used to multiply two n bit numbers. The algorithms were programmed in Ibuki Common Lisp and timing runs were made on a Sequent Balance 21000. Multiplication was implemented using the standard n^2 algorithm. Times for the various algorithms are reported as various constants times n^2. An algorithm based on generating factors of Fibonacci numbers had the smallest constant. The Fibonacci sequence, arranged in various ways, is searched for redundant information that could be eliminated to reduce the number of operations. Cycles in the bth bit of f_n were discovered but are not yet completely understood.

Algorithms for Computing Fibonacci Numbers Quickly

By J. L. Holloway

A Thesis submitted to

Oregon State University

in partial fulfillment of the

requirements for the degree of

Master of Science

Completed June 2, 1988

Commencement June 1989.

Approved:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis presented May 10, 1988

ACKNOWLEDGEMENTS

I wish to thank:

Dr. Paul Cull for his seemingly unending patience, guidance in studying algorithms to compute Fibonacci numbers and the eight (fifteen) puzzle, and teaching the computer science theory classes in a most unique and interesting manner.

Nick Flann and Russell Ruby for their careful reading of this thesis. Their comments have directly lead to the clarification of many points that would have been even more abstruse.

Judy and David Kelble, and also Mildred and David Chapin, who have, perhaps unwittingly, taught me what is important and what is not.

Thank you.

This thesis is dedicated to the memory of Almon Chapin:
He would have enjoyed the trip.

Table of Contents

1 Introduction 1

1.1 Problem Definition 1

1.2 History 2

1.3 Applications 2

1.4 The Model of Computation 4

1.5 A guide to this Thesis 4

2 Methods of Computation 6

2.1 Natural recursive 6

2.2 Repeated addition 7

2.3 Binet's formula 9

2.4 Matrix Methods of Gries, Levin and Others 12

2.5 Matrix Methods 14

2.5.1 Three Multiply Matrix Method 15

2.5.2 Two Multiply Matrix Method 17

2.6 Extending N. N. Vorob'ev's methods 18

2.7 Binomial Coefficients 21

2.8 Generating function 25

2.9 Product of Factors 25

3 Results and Analysis 31

3.1 Natural recursive 32

3.2 Repeated addition 33

3.3 Binet's Formula 34
3.4 Matrix Multiplication Methods 35
3.4.1 Three Multiply Matrix Method 37
3.4.2 Two Multiply Matrix Method 37
3.5 Extended N. N. Vorob'ev's Method 38
3.6 Product of Factors 38
3.7 f_n for any n using Product of Factors 39
3.8 Matrix Method of Gries and Levin 40
3.9 Binomial Coefficients 41
3.10 Actual running times 45

4 Redundant Information 49
4.1 Compression 50
4.1.1 Ziv Lempel Compression 51
4.1.2 Dynamic Huffman Codes 55
4.2 Testing for Randomness 57
4.2.1 Equidistribution 58
4.2.2 Serial Test 61
4.2.3 Wald-Wolfowitz Total Number of Runs Test 61
4.3 Cycles in the b Bit of f_n 62

5 Conclusions and Future Research 68

Bibliography 71

Appendices 74

A List of Fibonacci Relations 74

B Lisp Functions 75

B.1 Natural Recursion 75

B.2 Repeated Addition 75

B.3 Binet's Formula 75

B.4 N. N. Vorob'ev's Methods 76

B.5 Matrix Methods 77

B.5.1 Three Multiply Matrix Method 77

B.5.2 Two Multiply Matrix Method 77

B.6 Generating Function 77

B.7 Binomial Coefficients 79

B.8 Generalized Fibonacci Numbers 79

List of Figures

2.1 Recursive algorithm to compute f_n 7

2.2 Repeated addition algorithm 8

2.3 Addition algorithm for f_n^k 9

2.4 Approximation of Binet's formula 11

2.5 Recursive Binet's approximation 11

2.6 Three multiply matrix algorithm 16

2.7 Two multiply matrix algorithm 17

2.8 Extended N. N. Vorob'ev algorithm to compute f_n 20

2.9 Call tree for f_{16} 20

2.10 Goetgheluck's algorithm 24

2.11 Algorithm to compute β 27

2.12 Algorithm to compute f_{2^i} using β 27

2.13 Product of factors algorithm to compute any f_n 29

2.14 Recursive section of algorithm to compute any f_n 30

3.1 Elements of Pascal's triangle used to compute a binomial coefficient 42

4.1 Pseudo-code for ZL compression algorithm 53

4.2 Huffman's algorithm to compute letter frequencies 56

List of Tables

2.1 Running times of algorithms to compute f_n^k 13

2.2 Pascal's triangle 22

3.1 Constants for algorithms to compute f_n 45

3.2 Running time to compute f_n for any n in CPU seconds 46

3.3 Running time to compute f_n, $n = 2^k$, in CPU seconds, part 1 . 47

3.4 Running time to compute f_n, $n = 2^k$, in CPU seconds, part 2 . 48

4.1 Compression ratios of Ziv–Lempel for various sources 51

4.2 Compression ratios of the modified Lempel–Ziv–Welch algorithm 52

4.3 Compression ratios for Unix Ziv-Lempel implementation 52

4.4 Compression ratios of the dynamic Huffman code algorithm . . 55

4.5 Equidistribution test for randomness 58

4.6 Serial test for randomness 59

4.7 Wald-Wolfowitz test for randomness 60

4.8 Repeated patterns in $\mathcal{B}_i$ 65

5.1 Summary of bit operation complexity using n^2 multiply 69

Chapter 1

Introduction

1.1 Problem Definition

In the year 1202 Leonardo Pisano, sometimes referred to as Leonardo Fibonacci, proposed a model of the growth of a rabbit population to be used as an exercise in addition. The student is told that rabbits are immortal, rabbits mature at the age of one month, and all pairs of fertile rabbits produce one pair of offspring each month. The question "Starting with one fertile pair of rabbits, how many pairs of rabbits will there be after one year?" is then posed. [9,19]

Initially there is one pair of rabbits, after the first month there will be two pairs of rabbits, the second month three pair, the third month five pairs, the fourth month eight pairs, the fifth month thirteen pairs and so on. The sequence of numbers 0, 1, 1, 2, 3, 5, 8, 13, ... where each number is the sum of the previous two numbers is referred to as the Fibonacci sequence. The sequence is formally defined as:

$$f_{n+2} = f_{n+1} + f_n,\ n \geq 0,\ f_0 = 0,\ f_1 = 1 \tag{1.1}$$

The Fibonacci numbers may be generalized to allow the nth number to be the sum of the previous k numbers in the sequence. The sequence of order-k Fibonacci numbers is defined by [15,25,33] as:

$$f_0^k = f_1^k = \cdots = f_{k-3}^k = f_{k-2}^k = 0,\ f_{k-1}^k = 1$$

and

$$f_n^k = \sum_{i=1}^{k} f_{n-i}^k \text{ for } n \geq k.$$

Unless explicitly specified otherwise the term "Fibonacci numbers" refers to the order-2 sequence of Fibonacci numbers and f_n will refer to the nth element of the order-2 Fibonacci sequence.

1.2 History

Leonardo Fibonacci first proposed the sequence of numbers named in his honor in his book *Liber Abbaci* (Book of the Abacus). The rabbit problem was an exercise in addition and not an attempt to accurately model a rabbit population. It appears that the relation $f_{n+2} = f_{n+1} + f_n$ was not recognized by Fibonacci but was first noted by Albert Girard in 1634. In 1844 G. Lamé became the first person to use the Fibonacci numbers in a "practical" application when he used the Fibonacci sequence to prove the number of divisions needed to find the greatest common divisor of two positive integers using the Euclidean algorithm does not exceed five times the number of digits in the smaller of the two integers. [1] The use of the term "Fibonacci numbers" was initiated by E. Lucas in the 1870's. Many relations among the Fibonacci and related numbers are due to Lucas and a recurring series first proposed by Lucas has taken his name. The Lucas numbers are defined as:

$$l_{n+2} = l_{n+1} + l_n, \; n \geq 0 \; l_0 = 2, \; l_1 = 1$$

Another series, that appears later in this thesis, first described by Lucas is:

$$\beta_{n+1} = \beta_n^2 - 2, \; \beta_1 = 3$$

giving the series 3, 7, 47, ... A more complete history of Fibonacci numbers can be found in chapter 17 of [9].

1.3 Applications

Applications for large Fibonacci numbers do arise in computer science and routinely appear in undergraduate data structure texts in chapters on searching, sorting,

[1]This proof may be found on page 62 of [29].

and memory management [16,22,27]. Fibonacci search assumes that the list being searched is $f_n - 1$ elements long and we know the numbers f_n and f_{n-1}. The method of dividing the list into two parts and choosing the upper or lower part depending on a comparison is the same as the binary search. The difference is that the Fibonacci search, instead of bisecting the list being searched at the $n/2$ element as the binary search does, bisects the list at the $(\sqrt{5} - 1)/2$ element by adding or subtracting f_{n-i}, where this is the ith split. No multiplications or divisions are required, only additions and subtractions, to find the next bisection point. The complete algorithm can be found in [16]. The buddy system of memory management can be generalized to divide a block of memory of size f_{n+2} into two blocks of memory of sizes f_{n+1} and f_n. It is clear that this could be further generalized to order-k Fibonacci numbers so the division of a block of memory would yield k, instead of two, smaller blocks of memory [22]. Polyphase merging is a method of merging large files on various tape drives into one file. The generalized Fibonacci numbers give the optimal number of files that should be on each tape drive for before merging begins. The order of the Fibonacci numbers best suited for the problem depends on the number of tape drives used to merge the files. In practice this is used to create, by internal sort routines, the proper number of files to place on the tape drive initially to make optimal use of the tape drives [27].

Fibonacci numbers appear in disciplines other than computer science as well. Peter Anderson has described a family of polycyclic aromatic hydrocarbons (benzene, naphthalene, phenanthrene, ...), such that the nth member has f_{n-1} Kekulé structures. Definitions of Kekulé structures, the specific hydrocarbons under consideration and the recurrence relations used appear in [2]. Joseph Lahr in his paper "Fibonacci and Lucas numbers and the Morgan-Voyce polynomials in ladder networks and in electric line theory" presents an application and cites several references of Fibonacci numbers to electrical engineering [21]. Fibonacci numbers appeared in biology as early as 1611 in the work of Kepler on phyllotaxis and more recently Roger Jean in his book *Mathematical Approach to Pattern and Form in Plant Growth* [17] makes extensive use of the Fibonacci sequence.

1.4 The Model of Computation

The previous work on fast algorithms to compute the Fibonacci numbers used the standard straight line code model of computation. [10,11,15,25,32,33]. The standard straight line model of computation uses the uniform cost function, that is, it assumes that the cost of an operation is the same regardless of the size of its operands. When the size of the operands becomes large the uniform cost assumption is no longer valid and the variable time required to operate on large operands must be taken into account. Since f_n grows exponentially (see theorem 2.2 of [29]) with n, f_n is large for reasonable size n.

Therefore the uniform cost function is unsuitable as a measure of the time used to compute Fibonacci numbers. The bitwise model is used to analyize and compare the various algorithms used to compute Fibonacci numbers since the bitwise model reflects the logarithmic cost of operations on variable sized operands [1]. The bitwise model is used to analyize and compare the various algorithms used to compute Fibonacci numbers. In this model all variables are assumed to be one bit in length and all operations are logical instead of arithmetic. Under the the bitwise model, operations on the integers i and j require at least the $\log i + \log j$ time units to execute.

1.5 A guide to this Thesis

Chapter two presents several methods of computing Fibonacci numbers found in the literature and several methods that do not, as yet, appear in the literature. Each section of the chapter will present one algorithm, a proof of correctness of that algorithm, and possible variations of the algorithm. Chapter three presents an analysis of the running time of each algorithm and actual running times of the implementation of each algorithm.

In an attempt to find redundant information in the binary representation of the Fibonacci numbers an analysis of the structure of the bits in f_n was undertaken. The methods and results of this analysis are presented in chapter four. The final

chapter summarizes the results of this thesis and proposes future areas of research. Appendix A contains a list of the important equations with equations numbers where appropriate. Appendix B presents the lisp functions that implement the major algorithms discussed in this thesis. Many variations of the algorithms and the supporting functions are not given but are available upon request from the author.

Chapter 2

Methods of Computation

There are many ways to compute Fibonacci numbers. Each section of this chapter will describe one method (and closely related methods) of computing the Fibonacci numbers.

The following algorithms in this chapter have been seen in print: natural recursive in section 2.1, repeated addition in section 2.2, Binet's formula in section 2.3, the matrix methods of Gries, Levin and Others in section 2.4, three multiply matrix method in section 2.5.1, and binomial coefficients in section 2.7. The remaining algorithms have been developed by the author and Dr. Paul Cull and are believed to be original: extended N. N. Vorob'evs Method in section 2.6, two multiply matrix method in section 2.5.2, generating function in section 2.8, Product of factors in section 2.9.

2.1 Natural recursive

Several introductory programming texts introduce recursion by computing the Fibonacci numbers. The function $\mathsf{fib}(n)$ that computes f_n will return zero if $n = 0$, one if $n = 1$ and the sum of $\mathsf{fib}(n-1)$ and $\mathsf{fib}(n-2)$ if $n \geq 2$, is proposed as the solution. More succinctly,

$$\mathsf{fib}(n) = \begin{cases} 0 & \text{if } 0 \leq n = 0 \\ 1 & \text{if } n = 1 \\ \mathsf{fib}(n-1) + \mathsf{fib}(n-2) & \text{if } n \geq 2 \end{cases} \tag{2.1}$$

```
fib (n)
    if n = 0 return 0
    else if n = 1 return 1
    else return fib(n-1) + fib(n-2)
```

Figure 2.1: Recursive algorithm to compute f_n

Order-k Fibonacci numbers can be computed using a similar method:

$$\mathsf{kfib}(n) = \begin{cases} 0 & \text{if } 0 \leq n < k \\ 1 & \text{if } n = k \\ \sum_{i=1}^{k} \mathsf{kfib}(n-i) & \text{if } n > k \end{cases} \tag{2.2}$$

Theorem 2.1 *The algorithm presented in figure 2.1, when given a non-negative integer n, will correctly compute f_n.*

Proof. Using induction on n. Base: when n = 0 the function will take the first if and return $0 = f_0$. When $n = 1$ the function will take the second if and return 1 $= f_1$. For the inductive step assume $n \geq 2$ and the algorithm correctly computes $\mathsf{fib}(n-2)$ and $\mathsf{fib}(n-1)$ and show that it correctly computes $\mathsf{fib}(n)$. Since $n \geq 2$ the last else statement will be executed and by the assumption above $\mathsf{fib}(n-2) = f_{n-2}$ and $\mathsf{fib}(n-1) = f_{n-1}$. The function $\mathsf{fib}(n)$ will return $f_{n-2} + f_{n-1}$ which is the definition of f_n. ∎

The results of each call to $\mathsf{fib}(n)$ could be cached to avoid the redundant computations. This algorithm could easily be modified to compute order-k Fibonacci numbers, see equation 2.2.

2.2 Repeated addition

If, instead of first computing f_{n-1} and f_{n-2} as was done in the recursive solution, each element of the sequence $f_0, f_1, f_2, \ldots, f_n$ was computed in order, the number

```
fib (n)
    previous = 0
    fibonacci = 1
    for loop = 2 to n
        temp = fibonacci
        fibonacci = fibonacci + previous
        previous = temp
    return fibonacci
```

Figure 2.2: Repeated addition algorithm

of operations could be reduced. The algorithm presented in figure 2.2 computes Fibonacci numbers in this manner.

Theorem 2.2 *The algorithm presented in figure 2.2 will, when given a positive integer n, correctly compute f_n.*

Proof. Using induction on n. Base: when n = 1, previous will be assigned $f_0 = 0$ and fibonacci will be assigned $f_1 = 1$. The loop is not executed since $n < 2$ and the value of fibonacci is returned. For the inductive step assume the algorithm correctly computes f_{n-1} and show that it correctly computes f_n. Immediately before the last iteration of the loop, by the assumption, fibonacci is equal to f_{n-1} and previous is equal to f_{n-2}. In the last iteration of the loop the value of f_{n-1} is stored in temp, fibonacci is assigned the value of $f_{n-1} + f_{n-2}$, and previous gets the value stored in temp $= f_{n-1}$. Since that was the last iteration of the loop the value of fibonacci $= f_{n-1} + f_{n-2} = f_n$ is returned. ∎

This algorithm can easily be extended to compute order-k Fibonacci numbers by adding the previous k numbers instead the previous two numbers. Another addition algorithm for order-k Fibonacci numbers can be created by noticing

$$f_n^k = 2f_{n-1}^k - f_{n-k-1}^k \tag{2.3}$$

```
kfib (n, k)
    if n < k − 1 return 0
    else if n ≤ k return 1
    else
        store f_0^k = 0, f_1^k = 0, ..., f_{k-2}^k = 0, f_{k-1}^k = 1, f_k^k = 1
        for loop = k+1 to n
            let f_loop^k = 2f_{loop-1}^k − f_{loop-k-1}^k
            store f_loop^k
            remove f_{loop-k-1}^k from storage
    return f_loop^k
```

Figure 2.3: Addition algorithm for f_n^k

Since adding f_{n-1}^k to

$$f_{n-1}^k = f_{n-2}^k + f_{n-3}^k + f_{n-4}^k + \cdots + f_{n-k-1}^k$$

we get

$$2f_{n-1}^k = f_{n-1}^k + f_{n-2}^k + f_{n-3}^k + \cdots + f_{n-k}^k + f_{n-k-1}^k$$

and subtracting f_{n-k-1}^k we get

$$2f_{n-1}^k - f_{n-k-1}^k = f_{n-1}^k + f_{n-2}^k + f_{n-3}^k + \cdots + f_{n-k}^k = f_n^k$$

which is equation 2.3. This algorithm is presented in figure 2.3.

2.3 Binet's formula

While the following is usually called Binet's formula for f_n, Knuth [19] credits A. de Moivre with discovering that a closed form for f_n, is:

$$f_n = \frac{1}{\sqrt{5}}(\lambda_1^n - \lambda_2^n) \tag{2.4}$$

where

$$\lambda_1 = \frac{1+\sqrt{5}}{2}$$

and

$$\lambda_2 = \frac{1-\sqrt{5}}{2}$$

From equation 1.1 we can get the characteristic polynomial

$$\lambda^2 = \lambda + 1$$

with roots

$$\lambda_1 = \frac{1+\sqrt{5}}{2}, \quad \lambda_2 = \frac{1-\sqrt{5}}{2}$$

so

$$f_n = a\lambda_1^n + b\lambda_2^n$$

$$f_0 = 0 = a + b \Rightarrow b = -a$$

$$\begin{aligned} f_1 &= 1 = a\lambda_1 + b\lambda_2 \\ &= a(\lambda_1 - \lambda_2) \\ a &= \frac{1}{\lambda_1 - \lambda_2} \\ &= \frac{1}{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}} \\ &= \frac{1}{\sqrt{5}} \end{aligned}$$

so

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] \tag{2.5}$$

which is the result we wanted.

Since λ_2 is approximately equal to -0.61803, $|\lambda_2^n|$ gets small as n increases. Knuth [19] states that

$$f_n = \left\lfloor \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + 0.5 \right\rfloor, \; n \geq 0 \tag{2.6}$$

Capocelli and Cull [8] extend this result and show that the generalized Fibonacci numbers are also rounded powers.

```
fib (n)

    compute the $\mathcal{N}n + \log n$ most significant bits of $\sqrt{5}$

    return $\left\lfloor \frac{1}{\sqrt{5}} \left(\frac{(1+\sqrt{5})}{2}\right)^n + 0.5 \right\rfloor$
```

Figure 2.4: Approximation of Binet's formula

```
fib (n)

    if n = 1 $f_1 \leftarrow 1$

    else $f_n \leftarrow \left\lceil (\text{fib}(n/2))^2 \cdot \sqrt{5} \right\rceil$

    return $f_n$
```

Figure 2.5: Recursive Binet's approximation

Theorem 2.3 *The algorithm presented in figure 2.4 correctly computes f_n for the non-negative integer n.*

Proof. If $\log_2 f_n = \mathcal{N}n$ for some constant $\mathcal{N}$ [1] then the result of equation 2.6 must be correct to at least $\mathcal{N}n$ bits. The first line of the algorithm computes the first $\mathcal{N}n + \log n$ bits of $\sqrt{5}$. At most, $\log n$ bits will be lost computing the nth power since no more than one bit is lost for each multiplication and exponentiation can be done using $\log n$ squarings. So, the first $\mathcal{N}n$ bits of the result will be correct and all but the first $\mathcal{N}n$ bits are truncated. ∎

A recursive version of the approximation of Binet's formula is given in figure 2.5.

Theorem 2.4 *The algorithm presented in figure 2.5 correctly computes f_n such that $n = 2^k$ for the non-negative integer k.*

Proof.

$$f_n = \frac{1}{\sqrt{5}}(\lambda_1^n - \lambda_2^n)$$

[1] See Chapter 3 introduction for the definition of $\mathcal{N}$.

$$
\begin{aligned}
(f_n)^2 &= \frac{1}{\sqrt{5}\sqrt{5}}\left(\lambda_1^{2n} - 2(\lambda_1\lambda_2)^n + \lambda_2^{2n}\right) \\
\sqrt{5}(f_n)^2 &= \frac{1}{\sqrt{5}}\left(\lambda_1^{2n} - 2(-1)^n + \lambda_2^{2n}\right)
\end{aligned}
$$

$$
\begin{aligned}
f_{2n} &= \frac{1}{\sqrt{5}}\left(\lambda_1^{2n} - \lambda_2^{2n}\right) \\
f_{2n} - \sqrt{5}(f_n)^2 &= \frac{1}{\sqrt{5}}\left(-\lambda_2^{2n} + 2(-1)^n - \lambda_2^{2n}\right) \\
\epsilon &= f_{2n} - \sqrt{5}(f_n)^2 \\
&= \frac{2}{\sqrt{5}}\left(-\lambda_2^{2n} + (-1)^n\right) \\
0 &< \epsilon < 1
\end{aligned}
$$

so

$$
\begin{aligned}
f_{2n} &= \sqrt{5}(f_n)^2 + \epsilon \\
&= \left\lceil \sqrt{5}(f_n)^2 \right\rceil
\end{aligned}
$$

∎

2.4 Matrix Methods of Gries, Levin and Others

In the late 1970's and early 1980's several papers were published on the number of operations needed to compute generalized Fibonacci numbers [10,11,15,32,33]. The problem addressed in these papers was to find a fast method of computing the nth element of the order-k Fibonacci sequence or, more succinctly, f_n^k. All of these papers used the uniform cost criteria allowing them to perform multiplication and addition on any size operands in a constant amount of time, therefore these papers report the running time of their algorithms as the number of multiplies, and additions instead, as will be done here, in the number of bit operations. Table 2.1, compiled from information by [10,12], shows the running time, number of multiplies and the number of additions for the algorithms presented by [10,12,15,32,33]. Fiduccia's algorithm uses the term $\mu(k)$ to be the total number of arithmetic operations required

Time to compute f_n^k			
Author	Running time	Multiplies	Additions
Gries	$\Theta(1.5k^2 \, log \, n)$	$\Theta(1.5k^2)$	$\Theta(3k^2)$
Shortt	$\Theta(k^2 \, log \, n)$	$\Theta(k^2)$	$\Theta(k^3)$
Pettrossi	$\Theta(1.5k^3 \, log \, n)$	$\Theta(1.5k^3)$	$\Theta(1.5k^3)$
Urbanek	$\Theta(k^3 \, log \, n)$	$\Theta(k^3 + 0.5k^2)$	$\Theta(k^3 + 0.5k^2)$
Fiduccia	$\Theta(\mu(k) \, \log n)$	$\Theta(\mu(k) \, \log n)$	$\Theta(\mu(k) \, \log n)$

Table 2.1: Running times of algorithms to compute f_n^k

to multiply two polynomials of degree $k - 1$. If the fast Fourier transform can be used, the number of arithmetic operations used by Fiduccia's algorithm would be $\Theta(k \cdot \log k \cdot \log n)$.

The algorithm that is presented by Gries and Levin [15] will be presented here, the methods of [10,32,33] are similar.

Let

$$A = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

and use the recurrence relation

$$\begin{bmatrix} f_n^k \\ f_{n-1}^k \\ f_{n-2}^k \\ \vdots \\ f_{n-k+1}^k \end{bmatrix} = A \begin{bmatrix} f_{n-1}^k \\ f_{n-2}^k \\ f_{n-3}^k \\ \vdots \\ f_{n-k}^k \end{bmatrix}$$

So, to compute f_n^k we only need to compute A^{n-k+1} since

$$\begin{bmatrix} f_n^k \\ f_{n-1}^k \\ f_{n-2}^k \\ \vdots \\ f_{n-k+1}^k \end{bmatrix} = A^{n-k+1} \begin{bmatrix} f_{k-1}^k = 1 \\ f_{k-2}^k = 0 \\ f_{k-3}^k = 0 \\ \vdots \\ f_0^k = 0 \end{bmatrix}$$

By noticing that any row, except the first, of A^i is the same as the previous row of A^{i-1} Gries and Levin were able to construct an algorithm that computes A^{i+1} from A and A^i by computing the bottom row of A^{i+1} and copying the remaining rows from A^i. This reduced the number of multiplies from $\Theta(k^3)$ to $\Theta(k^2)$ to do the matrix multiply. Since A^{n-k} can be computed using $\Theta(\log n)$ matrix multiplies by using repeated squaring, the time to compute f_n^k using Gries and Levin's algorithm is $\Theta(k^2 \log n)$. In the following section we will examine the order-2 Fibonacci numbers and let the array $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ which has the same form as the matrix A above.

2.5 Matrix Methods

The matrix identity

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

can be used to compute Fibonacci numbers. The identity is proven by induction on n. The base case, when $n = 1$, is true: $f_{n+1} = f_2 = 1$, $f_n = f_1 = 1$ and $f_{n-1} = f_0 = 0$. Assume that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

and show that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} = \begin{pmatrix} f_{n+2} & f_{n+1} \\ f_{n+1} & f_n \end{pmatrix}$$

is true.

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} + f_n & f_n + f_{n-1} \\ f_{n+1} & f_n \end{pmatrix} = \begin{pmatrix} f_{n+2} & f_{n+1} \\ f_{n+1} & f_n \end{pmatrix}$$

By the assumption above and matrix multiplication it can be seen that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} = \begin{pmatrix} f_{n+2} & f_{n+1} \\ f_{n+1} & f_n \end{pmatrix}$$

is true. ∎

By using repeated squarings, f_n could be computed using about $\log n$ matrix multiplies. One method of computing the matrix $\begin{pmatrix} f_{2n+1} & f_{2n} \\ f_{2n} & f_{2n-1} \end{pmatrix}$ from $\begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$ is to compute the top row of the new matrix using four multiplies. Using equations 2.7, 2.12 and 1.1

$$\begin{aligned} f_{2n+1} &= f_{n+1}^2 + f_n^2 \\ f_{2n} &= f_{n+1} f_n + f_n f_{n-1} \\ f_{2n-1} &= f_{2n+1} - f_{2n} \end{aligned}$$

all of the elements of the matrix can be computed. The following two methods will use three and two multiplies, respectively, to compute each succeeding matrix.

2.5.1 Three Multiply Matrix Method

The three multiply matrix method is shown in Figure 2.6. The following derivation, along with equation 2.13 below, show that $f_{2n+1} = f_{n+1}^2 + f_n^2$ and $f_{2n-1} = f_n^2 + f_{n-1}^2$.

$$\begin{aligned} f_{2n+1} &= f_{n+1} f_{n-1} + f_n f_{n+2} \\ &= f_{n+1}(f_{n+1} - f_n) + f_n(f_{n+1} + f_n) \\ &= f_n^2 + f_{n+1}^2 \end{aligned} \qquad (2.7)$$

which is the result wanted.

fib (n)

if $n = 1$ return the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

else

$\begin{pmatrix} f_{n/2+1} & f_{n/2} \\ f_{n/2} & f_{n/2-1} \end{pmatrix} \leftarrow$ fib ($n/2$)

$a \leftarrow (f_{n/2+1})^2$

$b \leftarrow (f_{n/2})^2$

$c \leftarrow (f_{n/2-1})^2$

return the matrix $\begin{pmatrix} a+b & a-c \\ a-c & b+c \end{pmatrix}$

Figure 2.6: Three multiply matrix algorithm

Theorem 2.5 *The three multiply matrix algorithm, given a positive integer n such that $n = 2^k$ for some non-negative integer k, correctly computes the matrix* $\begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$.

Proof. Using induction on n. Base: when $n = 1$ the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ is returned. Since $f_2 = f_1 = 1$ and $f_0 = 0$ the base case is correct. For the inductive step assume that the algorithm correctly computes $\begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$ and show that it correctly computes $\begin{pmatrix} f_{2n+1} & f_{2n} \\ f_{2n} & f_{2n-1} \end{pmatrix}$ Equation 2.7 shows, given f_{n-1}^2, f_n^2, and f_{n+1}^2, that f_{2n+1} and f_{2n-1} can be computed. Equation 1.1 shows that $f_{2n+1} = f_{2n} + f_{2n-1}$ or, rearranging, $f_{2n} = f_{2n+1} - f_{2n-1}$ so the three multiply matrix algorithm is correct.

∎

fib (n)

if $n = 1$ return the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

else

$\begin{pmatrix} f_{n/2+1} & f_{n/2} \\ f_{n/2} & f_{n/2-1} \end{pmatrix} \leftarrow$ fib $(n/2)$

$f_{n+2} \leftarrow f_{n/2+1}(f_{n/2+1} + 2f_{n/2})$

$f_n \leftarrow f_{n/2}(2f_{n/2+1} - f_{n/2})$

return the matrix $\begin{pmatrix} f_{n+2} - f_n & f_n \\ f_n & f_{n+2} - 2f_n \end{pmatrix}$

Figure 2.7: Two multiply matrix algorithm

2.5.2 Two Multiply Matrix Method

These matrices can be multiplied using only two multiplications using the two multiply algorithm is given in figure 2.7. To prove the two multiply matrix algorithm is correct we need to show that $f_{2n+2} = f_{n+1}(f_{n+1} + 2f_n)$ and $f_{2n} = f_n(2f_{n+1} - f_n)$. Using equation 2.12

$$\begin{aligned} f_{n+1}(f_{n+1} + 2f_n) &= f_{n+1}(f_n + f_{n+2}) \\ &= f_n f_{n+1} + f_{n+1} f_{n+2} \\ &= f_{2n+2} \end{aligned} \tag{2.8}$$

and, remembering that $f_{n+1} = f_n + f_{n-1}$,

$$\begin{aligned} f_n(2f_{n+1} - f_n) &= f_n(f_{n+1} + f_{n-1}) \\ &= f_n f_{n+1} + f_n f_{n-1} \\ &= f_{2n} \end{aligned} \tag{2.9}$$

Theorem 2.6 *The two multiply matrix algorithm given in figure 2.7 correctly computes the matrix* $\begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$ *when given a positive integer* n *such that* $n = 2^k$.

Proof. Using induction on n. Base: when $n = 1$ the algorithm returns the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Since $f_2 = f_1 = 1$ and $f_0 = 0$ the base case is correct. For the inductive step assume that, when given n, the algorithm correctly computes $\begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$ and show that it correctly computes $\begin{pmatrix} f_{2n+1} & f_{2n} \\ f_{2n} & f_{2n-1} \end{pmatrix}$. From the assumption we have f_{n+1}, f_n, and f_{n-1}. From equation 2.8 we can compute f_{2n+2} from f_{n+1} and f_n. From equation 2.9 we can compute f_{2n} given f_{n+1} and f_n. Since $f_{2n+1} = f_{2n+2} - f_{2n}$ and $f_{2n-1} = f_{2n+2} - 2f_{2n}$ the two multiply matrix algorithm correctly computes $\begin{pmatrix} f_{2n+1} & f_{2n} \\ f_{2n} & f_{2n-1} \end{pmatrix}$. ∎

2.6 Extending N. N. Vorob'ev's methods

N. N. Vorob'ev, in his book *Fibonacci Numbers*, presents several methods of computing f_n. The following relationships, although surely developed many years ago by others, are now standard relationships between Fibonacci numbers. They are presented with Vorob'ev's name because his book [36] presents a clear and concise derivation of the following important relationship:

$$f_{n+k} = f_{n-1}f_k + f_n f_{k+1}. \tag{2.10}$$

This relationship will be proven by induction on k. Let the inductive hypothesis, $H(k)$, be: $f_{n+k} = f_{n-1}f_k + f_n f_{k+1}$. For the first base case let $k = 1$ giving $f_{n+1} = f_{n-1}f_1 + f_n f_2$. Since $f_1 = f_2 = 1$ we get $f_{n+1} = f_{n-1} + f_n$ which is true. For the second base case let $k = 2$ giving $f_{n+2} = f_{n-1}f_2 + f_n f_3$. Since $f_2 = 1$ and $f_3 = 2$ we get $f_{n+2} = f_{n-1} + 2f_n$. By replacing $f_{n-1} + f_n$ with f_{n+1} we get $f_{n+2} = f_{n+1} + f_n$ which is true.

For the inductive step assume $H(k)$ and $H(k+1)$ are true and prove $H(k+2)$ is true. This gives the two equations:

$$\begin{aligned} f_{n+k} &= f_{n-1}f_k + f_nf_{k+1} \\ f_{n+k+1} &= f_{n-1}f_{k+1} + f_nf_{k+2} \end{aligned}$$

Adding these two equations gives:

$$\begin{aligned} f_{n+k} + f_{n+k+1} &= f_{n-1}f_k + f_nf_{k+1} + f_{n-1}f_{k+1} + f_nf_{k+2} \\ f_{n+k+2} &= f_{n-1}(f_k + f_{k+1}) + f_n(f_{k+1} + f_{k+2}) \\ f_{n+k+2} &= f_{n-1}f_{k+2} + f_nf_{k+3} \end{aligned}$$

The last equation is the result we are trying to prove. ▮

If we let $k = n$ in equation 2.10 above we get

$$\begin{aligned} f_{2n} &= f_{n-1}f_n + f_nf_{n+1} \\ &= f_n(f_{n-1} + f_{n+1}) \qquad (2.11) \\ &= (f_{n+1} - f_{n-1})(f_{n-1} + f_{n+1}) \\ &= f_{n+1}^2 - f_{n-1}^2 \qquad (2.12) \end{aligned}$$

This gives an interesting relationship but is not of much use, alone, in computing Fibonacci numbers since it requires f_{n+1} and f_{n-1}, to compute f_{2n}, and if n is even we can not use equation 2.12 to compute f_{n-1} and f_{n+1}. By letting $k = n+1$ we will get a relation that will compute f_{2n+1} and allow us to compute both even and odd Fibonacci numbers.

$$\begin{aligned} f_{2n+1} &= f_{n-1}f_{n+1} + f_nf_{n+2} \\ &= f_{n-1}f_{n+1} + f_n(f_{n+1} + f_n) \qquad (2.13) \end{aligned}$$

To compute f_{2n}, $n = 2^k$, we compute f_n and f_{n+1} using equation 2.12 and equation 2.13. Equation 2.12 was chosen because it uses only one multiply and equation 2.13 was chosen since it uses three Fibonacci numbers instead of four.

Figure 2.8 gives an algorithm to compute f_{2^i} using equation 2.12 and equation 2.13. The call to fib(n) will recursively compute $f_{n/4}$, $f_{n/4+1}$ and $f_{n/2}$ and with

```
fib (n)
    if n = 4 return f_2 = 1, f_3 = 2, f_4 = 3
    else
        f_{n/4}, f_{n/4+1}, f_{n/2} ← fib (n/2)
        f_{n/2+1} ← f_{n/4-1} f_{n/4+1} + f_{n/4}(f_{n/4+1} + f_{n/4})
        f_n ← f_{n/2}(f_{n/2-1} + f_{n/2+1})
        return f_{n/2}, f_{n/2+1}, f_n
```

Figure 2.8: Extended N. N. Vorob'ev algorithm to compute f_n

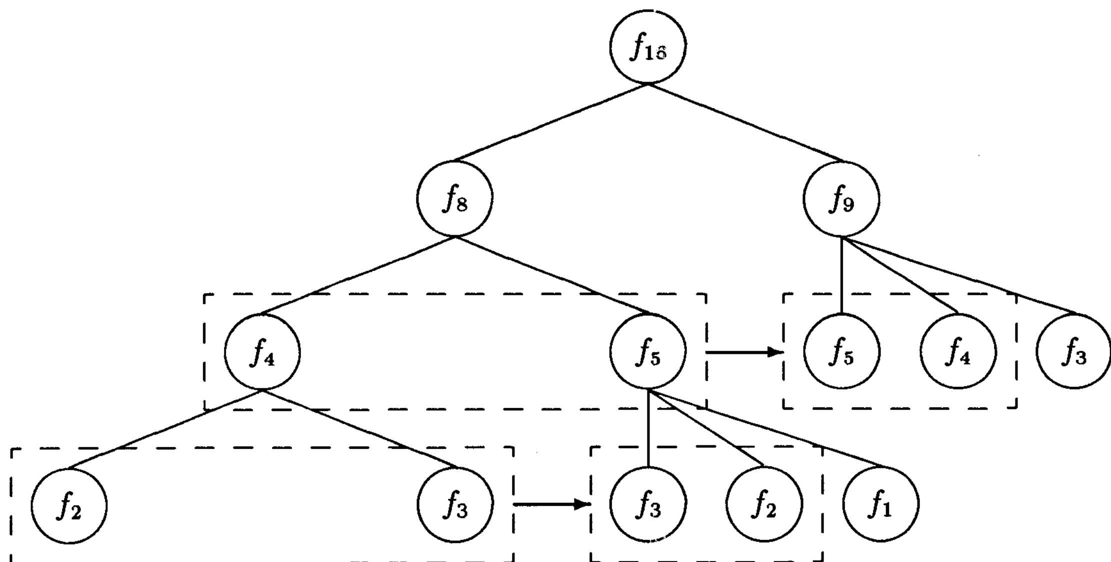

Figure 2.9: Call tree for f_{16}

these values, f_{n+1} can be computed (see equation 2.13) and f_{2n} can be computed (see equation 2.12).

Figure 2.9 shows a call tree for this algorithm and graphically shows how the results of computing $f_{n/2}$ are used to compute f_{n+1}. To compute f_{16} the algorithm first computes f_8 recursively, resulting in f_4, f_5 and f_8. To compute f_{16} we need f_8, which we have and f_9. To compute f_9 we need f_5, f_4, which we have as a result of computing f_8 and f_3 which we can compute with one subtraction, $f_3 = f_5 - f_4$.

Theorem 2.7 *The algorithm in figure 2.8 correctly computes $f_{n/2}$, $f_{n/2+1}$ and f_n, $n = 2^k$, for the integer $k \geq 2$.*

Proof. Using induction on k. Base: when $k = 2$ the algorithm returns $f_{2^1} = f_2 = 1$, $f_{2^1+1} = f_3 = 2$ and $f_{2^2} = f_4 = 3$. For the inductive step assume the algorithm correctly computes $f_{2^{k-2}}$, $f_{2^{k-2}+1}$ and $f_{2^{k-1}}$ and show that it correctly computes $f_{2^{k-1}}$, $f_{2^{k-1}+1}$ and f_{2^k}. By the assumption the first line of the else correctly computes $f_{2^{k-2}}$, $f_{2^{k-2}+1}$ and $f_{2^{k-1}}$. From equation 2.13 we can compute $f_{2^{k-1}+1}$ and equation 2.12 gives us f_{2^k} and then fib(n) returns $f_{2^{k-1}}$, $f_{2^{k-1}+1}$ and f_{2^k}, the desired result. ∎

2.7 Binomial Coefficients

An interesting relationship between binomial coefficients and Fibonacci numbers exists and is easiest to show when the binomial coefficients are organized as Pascal's triangle. Binomial coefficients, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, and the two following relationships [4] lead to Pascal's triangle.

$$\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1} \tag{2.14}$$

and

$$\binom{n}{k} = \binom{n}{n-k} \tag{2.15}$$

n						
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1
k	0	1	2	3	4	5

Table 2.2: Pascal's triangle

Figure 2.2 shows the binomial coefficients arranged in a table for $0 \leq n \leq 5$ and $0 \leq k \leq 5$. This arrangement is commenly referred to as Pascal's triangle because a similar table appeared in the book *Traité du triangle arithmétique* by Blaise Pascal in 1653. Binomial coefficients, and this particular arrangement, appear much earlier; see [19] for a short history.

Let the nth diagonal of Pascal's triangle, d_n, be defined as the sequence of binomial coefficients:

$$\binom{n}{0}, \binom{n-1}{1}, \binom{n-2}{2}, \ldots, \binom{0}{n}$$

Theorem 2.8 *The sum of the nth diagonal, d_n, of Pascal's triangle is equal to f_{n+1}.*

Proof. Using induction on n. Base: the first base case the sum $d_0 = \binom{0}{0} = 1 = f_1$ and the second base case the sum of $d_1 = \binom{1}{0} + \binom{0}{1} = 1 = f_2$. For the inductive step we need to show for $n \geq 2$, assuming that $d_{n-2} = f_{n-1}$ and $d_{n-1} = f_n$, that $d_n = d_{n-2} + d_{n-1} = f_{n+1}$. Adding d_{n-2} and d_{n-1} we get

$$\binom{n-1}{0} + \binom{n-2}{0} + \binom{n-2}{1} + \binom{n-3}{1} + \binom{n-3}{2} + \cdots$$

$$+\binom{0}{n-2}+\binom{0}{n-1}$$

Since

$$\binom{n-1}{0}=\binom{n}{0}=1$$

we can replace the first term with $\binom{n}{0}$ and apply equation 2.14 to all subsequent pairs of terms to get

$$\binom{n}{0}+\binom{n-1}{1}+\binom{n-2}{2}+\cdots+\binom{1}{n-1}$$

Since $n \geq 2$ and $\binom{0}{n-1}=0$, the sum of d_n is equal to the sum of d_{n-2} plus the sum of d_{n-1}. So, since $d_{n-2}=f_{n-1}$ and $d_{n-1}=f_n$, $d_n=d_{n-2}+d_{n-1}=f_{n-1}+f_n=f_{n+1}$. ∎

The algorithm to compute f_n is to take the sum of d_{n-1}. The interesting question is "Is there a fast way to compute binomial coefficients?". The standard method of computing binomial coefficients is [14]:

$$\binom{n}{k}=\binom{n-1}{k}+\binom{n-1}{k-1} \tag{2.16}$$

and

$$\binom{n}{0}=1$$

and if $n<k$

$$\binom{n}{k}=0$$

P. Goetgheluck [14] presents an algorithm to compute the prime power factorization of $\binom{n}{k}$. His algorithm is presented in figure 2.10, E is the power of

```
input n, k and p
E ← 0, r ← 0
if p > n − k then E ← 1; end.
if p > n/2 then E ← 0; end.
if p * p > n then if n mod p < k mod p then E ← 1; end.
Repeat
    a ← n mod p
    n ← ⌊n/p⌋
    b ← (k mod p) + r
    k ← ⌊k/p⌋
    if a < b then E ← E + 1, r ← 1
    else r ← 0
until n = 0
```

Figure 2.10: Goetgheluck's algorithm

the prime p in the factorization of $\binom{n}{k}$. Knowing the factors of $\binom{n}{k}$ for all $p \leq n$ the binomial coefficient can be computed by taking the product of $p_i^{E_i}$ for all $i \ni p_i \leq n$.

2.8 Generating function

We can use a generating function, $G(z)$, to represent the entire Fibonacci sequence.

$$G(z) = f_0 + f_1 z + f_2 z^2 + \cdots = \sum_{n \geq 0} f_n z^n. \tag{2.17}$$

Let

$$\begin{aligned} zG(z) &= f_0 z^1 + f_1 z^2 + f_2 z^3 \\ z^2 G(z) &= f_0 z^2 + f_1 z^3 + f_2 z^4 \end{aligned}$$

By computing $G(z) - zG(z) - z^2G(z)$ we get

$$\begin{aligned} G(z) - zG(x) - z^2 G(z) &= f_0 z^0 + (f_1 - f_0) z^1 + (f_2 - f_1 - f_0) z^2 + \\ &\quad (f_3 - f_2 - f_1) z^3 + \cdots \\ G(z)(1 - z - z^2) &= 0 + z + 0z^2 + 0z^3 + \cdots \\ G(z) &= z/(1 - z - z^2) \end{aligned} \tag{2.18}$$

We can evaluate the polynomial $G(z) = z/(1 - z - z^2)$ at the roots of unity ω^0, ω^1, ω^2, ..., ω^{n-1} getting the values v_0, v_1, v_2, ..., v_{n-1}. To find the coefficients, the Fibonacci numbers, perform the inverse Fourier transformation. To find a specific coefficient, f_k, use only the $k-1$ row of the Fourier matrix, $\mathcal{F}$.

$$f_n = \sum_{i=0}^{n-1} \mathcal{F}_{n-1,i} \cdot v_i$$

2.9 Product of Factors

The evaluation of the generating function above produced approximations of f_n. Several small terms were involved in computing the approximation and they were

eliminated one by one since they became very small as n grew. From empirical observations eliminating the small terms seemed to improve the approximation and when all of the small terms were gone the results seemed to be exact Fibonacci numbers.

The formula that was suggested is

$$f_{2^i} = \prod_{j=1}^{i} \beta_i$$

where

$$\beta_i = \beta_{i-1}^2 - 2, \quad \beta_1 = 3. \tag{2.19}$$

The numbers $\beta_1, \beta_2, \ldots, \beta_i$ are a factorization, although not necessarily prime, of f_{2^i}. The Fibonacci relationship this suggests is

$$\left(\frac{f_{2n}}{f_n}\right)^2 - 2(-1)^n = \frac{f_{4n}}{f_{2n}} \tag{2.20}$$

To show this is true, by equation 2.4,

$$\begin{aligned} \frac{f_{4n}}{f_{2n}} &= \frac{\lambda_1^{4n} - \lambda_2^{4n}}{\lambda_1^{2n} - \lambda_2^{2n}} \\ &= \lambda_1^{2n} + \lambda_2^{2n}. \end{aligned}$$

and, remembering that $\lambda_1 \cdot \lambda_2 = -1$

$$\begin{aligned} \left(\frac{f_{2n}}{f_n}\right)^2 - 2(-1)^n &= \left(\frac{\lambda_1^{2n} - \lambda_2^{2n}}{\lambda_1^n - \lambda_2^n}\right)^2 - 2(-1)^n \\ &= (\lambda_1^n + \lambda_2^n)^2 - 2(-1)^n \\ &= \lambda_1^{2n} + 2(\lambda_1\lambda_2)^n + \lambda_2^{2n} - 2(-1)^n \\ &= \lambda_1^{2n} + \lambda_2^{2n} \end{aligned}$$

So, equation 2.20 is correct. ∎

Lemma 2.1 *The algorithm in figure 2.11, when given a positive integer i, correctly computes a factorization of $f_{2^{i+1}}$.*

Proof. Using induction on i. Base: when $i = 1$ the algorithm returns a list with the one element $f_{2^{i+1}} = f_4 = 3$ so the base case is correct. For the inductive step assume

```
beta (i)
    if i = 1 return the list (3)
    else
        temp ← beta (i - 1)
        last ← last element of temp
        newbeta ← temp * temp - 2
        return the list temp with newbeta appended to the end.
```

Figure 2.11: Algorithm to compute β

```
fib (i)
    if i = 0 return 1
    else if i = 1 return 1
    else
        let temp = beta (i - 1)
        return the product of all elements in temp
```

Figure 2.12: Algorithm to compute f_{2^i} using β

that the algorithm correctly computes a factorization of f_{2^i} and show that for i it correctly computes a factorization of $f_{2^{i+1}}$. Since $n = 2^i$, $i \geq 1$, n is even and, by the assumption, $\frac{f_n}{f_{n/2}}$ is computed correctly, 2.20 tells us that $\left(\frac{f_n}{f_{n/2}}\right)^2 - 2(-1)^n = \frac{f_{2n}}{f_n}$. Since n is even $(-1)^n = 1$ and $-2(-1)^n$ becomes -2 so the step in the algorithm that computes **newbeta** is correct. So $\frac{f_{2n}}{f_n}$ can be computed and is added as the last element in the list temp and since the list **temp**, by the assumption, already contains a factorization of f_n and $f_n \cdot \frac{f_{2n}}{f_n} = f_{2n}$ the algorithm correctly computes a factorization of $f_{2^{i+1}}$. ∎

Theorem 2.9 *The algorithm in figure 2.12 correctly computes f_n, $n = 2^i$, for the non-negative integer i.*

Proof. If $i = 0$ the algorithm correctly returns $f_{2^0} = f_1 = 1$, if $i = 1$ the algorithm correctly returns $f_{2^1} = f_2 = 1$. For $i \geq 2$ the algorithm returns the product of the elements returned by the call to **beta** (i-1). By lemma 2.1 **beta** (j) correctly computes a factorization of $f_{2^{j+1}}$ so this algorithm correctly computes f_{2^i}. ∎

The algorithm in figure 2.12 could generate the sequence of Fibonacci numbers

$$f_4, f_8, f_{16}, \ldots, f_{2^i}$$

by printing the results of each multiplication. This can be generalized to generate the sequence

$$f_{4+k}, f_{8+k}, f_{16+k}, \ldots, f_{2^i+k}$$

for any integer k using the relation

$$f_{2^i+k} = f_{2^{i-1}+k}\beta_i - f_k \tag{2.21}$$

To show this is true, by equation 2.4,

$$\begin{aligned}
\left(\lambda_1^{2^i+k} - \lambda_2^{2^i+k}\right) &= \left(\lambda_1^{2^{i-1}+k} - \lambda_2^{2^{i-1}+k}\right)\frac{\left(\lambda_1^{2^i} - \lambda_2^{2^i}\right)}{\left(\lambda_1^{2^{i-1}} - \lambda_2^{2^{i-1}}\right)} - \left(\lambda_1^k - \lambda_2^k\right) \\
&= \left(\lambda_1^{2^{i-1}+k} - \lambda_2^{2^{i-1}+k}\right)\left(\lambda_1^{2^{i-1}} - \lambda_2^{2^{i-1}}\right) - \left(\lambda_1^k - \lambda_2^k\right) \\
&= \lambda_1^{2^i+k} - \lambda_1^{2^{i-1}}\lambda_2^{2^{i-1}+k} + \lambda_1^{2^{i-1}+k}\lambda_2^{2^{i-1}} - \lambda_2^{2^i+k} - \left(\lambda_1^k - \lambda_2^k\right) \\
&= \lambda_1^{2^i+k} - \lambda_2^{2^i+k} - (\lambda_1\lambda_2)^{2^{i-1}}\lambda_2^k - (\lambda_1\lambda_2)^{2^{i-1}}\lambda_1^k - \left(\lambda_1^k - \lambda_2^k\right)
\end{aligned}$$

```
fib (n)
    sl ← ⌈log n⌉ − 1
    β-sequence ← β_0, β_1, β_2, ..., β_{sl−1}
    f1-sequence ← f_1, f_3, f_7, ..., f_{2^{sl}−1}
    f2-sequence ← f_2, f_4, f_8, ..., f_{2^{sl}}
    fib-help (n sl f1-sequence f2-sequence)
```

Figure 2.13: Product of factors algorithm to compute any f_n

since $\lambda_1 \cdot \lambda_2 = -1$

$$\begin{aligned} &= \lambda_1^{2^i+k} - \lambda_2^{2^i+k} - \lambda_2^k + \lambda_1^k - \left(\lambda_1^k - \lambda_2^k\right) \\ &= \lambda_1^{2^i+k} - \lambda_2^{2^i+k} \end{aligned}$$

∎

By choosing $k = 0$ we get the sequence generated by the algorithm in figure 2.12

$$f_2,\ f_4,\ f_8,\ f_{16},\ \ldots$$

and by choosing $k = -1$ we get the sequence

$$f_1,\ f_3,\ f_7,\ f_{15},\ \ldots$$

The sequences

$$f_1,\ f_3,\ f_7,\ \ldots,\ f_{2^i-1} \tag{2.22}$$

and

$$f_2,\ f_4,\ f_8,\ \ldots,\ f_{2^i} \tag{2.23}$$

and equation 2.10 lead to the algorithm presented in figure 2.13

Lemma 2.2 *The algorithm presented in figure 2.14 will, when given a non-negative integer n, expo* $= \lceil \log n \rceil - 1$*, the sequence* $f_1, f_3, f_7, \ldots, f_{2^{expo}-1}$*, and the sequence* $f_2, f_4, f_8, \ldots, f_{2^{expo}}$ *correctly compute* f_n.

```
fib-help (n expo f1seq f2seq)
    if n = 0 return 0
    else if n = 1 return 1
    else if n = 2 return 1
    else
        return f1seq[expo] · fib-help (n − 2^expo, ⌈log(n − 2^expo)⌉ − 1, f1seq, f2seq) +
            f2seq[expo] · fib-help (n − 2^expo + 1 ⌈log(n − 2^expo)⌉ − 1, f1seq, f2seq)
```

Figure 2.14: Recursive section of algorithm to compute any f_n

Proof. Using induction on n. Base: when n = 0, $f_0 = 0$ is returned, when n = 1, $f_1 = 1$ is returned, and when n = 2, $f_2 = 1$ is returned. For the inductive step assume that the algorithm correctly computes f_j for all $0 \leq j \leq n$ and show that the algorithm correctly computes f_{n+1}. Since the ith element of f1-sequence is f_{2^i-1}, the ith element of the f2-sequence is f_{2^i}, and the assumption; the algorithm will return the value $f_{2^{expo}-1} \cdot f_{n-2^{expo}} + f_{2^{expo}} \cdot f_{n-2^{expo}+1}$. By equation 2.10 this is f_n. ∎

Theorem 2.10 *The algorithm presented in figure 2.13, when given an integer n, $3 \leq n$, will corectly compute f_n.*

Proof. The three sequences β-sequence, f1-sequence, and f2-sequence can be correctly computed using equations 2.19 and 2.21. By lemma 2.2, the algorithm correctly computes f_n. ∎

Chapter 3

Results and Analysis

$M(n)$ will be used for the number of bit operations used to multiply two n bit numbers. For most of the analysis it will be assumed that n^2 bit operations are used to do multiplication although better algorithms are known. The $\frac{3}{2}$ multiply algorithm presented in [1] is $\Theta(n^{\log_2 3})$ bit operations and the Schönhage–Strassen integer multiplication algorithm, also in [1], uses $\Theta(n \log n \log\log n)$ bit operations. These algorithms are not generally used for multiplication since, although they are asymptotically faster than the standard n^2 multiplication, they are slower for typical size numbers. Some of the faster algorithms are capable of computing Fibonacci numbers well over 10^5 bits in length, large enough so that the advantage of the $\frac{3}{2}$ multiply algorithm, and probably the Schönhage–Strassen integer multiplication algorithm, would be evident. The $\frac{3}{2}$ multiply algorithm was implemented and a significant increase in speed resulted (about a factor of 7 for the largest Fibonacci numbers).

The number of bit operations used by an algorithm to solve a problem of size n will be represented as $T(n)$. To facilitate counting the number of bit operations used to compute f_n the number of bits in f_n must be known. Since

$$f_n = \left\lfloor \frac{\lambda_1^n}{\sqrt{5}} + 0.5 \right\rfloor$$

the number of bits used to represent f_n is $\log_2 f_n$ or about $\log_2 \lambda_1^n$. We will use $\mathcal{N}n$ to represent the number of bits in the nth Fibonacci number where $\mathcal{N}$ is a constant.

$$\mathcal{N}n = \log_2 \lambda_1^n = n \log_2 \lambda_1 \tag{3.1}$$

So $\mathcal{N} = \log_2 \lambda_1 \approx 0.69424$.

3.1 Natural recursive

The number of bit operations required to compute f_n using the algorithm in figure 2.1 is the sum of the number of bit operations to compute f_{n-1} plus the number of bit operations to compute f_{n-2} plus the number of bit operations to add f_{n-1} and f_{n-2}. This gives the recurrence relation

$$T_n = T_{n-1} + T_{n-2} + \mathcal{N}n$$

with initial conditions

$$\begin{aligned} T_0 &= 0 \\ T_1 &= 0 \end{aligned}$$

Since this is a linear constant coefficient difference equation the general solution can be written

$$x = v + \sum_{i=1}^{k} \alpha_i \cdot \lambda_i^n$$

where v is the particular solution, λ_i is the ith root of the characteristic polynomial of the difference equation, and the α's are constants.

Solving the recurrence relation, we find the roots of the characteristic polynomial.

$$\begin{aligned} \lambda^2 - \lambda - 1 &= 0 \\ \lambda_1 &= \frac{1+\sqrt{5}}{2} \\ \lambda_2 &= \frac{1-\sqrt{5}}{2} \end{aligned}$$

Find the particular solution

$$\begin{aligned} V_n &= C_1\mathcal{N}n + C_2 \\ (C_1\mathcal{N}n + C_2) - (C_1\mathcal{N}(n-1) + C_2) - (C_1\mathcal{N}(n-2) + C_2) &= \mathcal{N}n \\ C_1 &= -1 \\ C_2 &= -3\mathcal{N} \end{aligned}$$

Solve the following two equations for α_1 and α_2

$$\begin{aligned} T_0 - V_0 &= \alpha_1 \lambda_1^0 + \alpha_2 \lambda_2^0 \\ T_1 - V_1 &= \alpha_1 \lambda_1^1 + \alpha_2 \lambda_2^1 \end{aligned}$$

Giving

$$\begin{aligned} \alpha_1 &= \mathcal{N}\left(3 - \frac{4 - 3\lambda_1}{\lambda_2 - \lambda 1}\right) \\ \alpha_2 &= \mathcal{N}\left(\frac{4 - 3\lambda_1}{\lambda_2 - \lambda_1}\right) \end{aligned}$$

Let $\beta = \mathcal{N}\left(3 - \frac{4-3\lambda_1}{\lambda_2-\lambda_1}\right)$ So

$$T_n = \beta\lambda_1^n + (3\mathcal{N} - \beta)\lambda_2^n - \mathcal{N}n - 3\mathcal{N} \quad (3.2)$$

Since $0 < \beta < 3$ both α_1 and α_2 are positive. Since $0 < \alpha_1$, $0 < \alpha_2$, $-1 < \lambda_2 < 0$, and $1.5 < \lambda_1 < 2$ the number of bit operations used by the algorithm in figure 2.1 is $\Theta(\lambda_1^n)$. Since $\lambda_1 > 1.5$ and this algorithm is exponential in n, it is unlikely to be useful for large n.

3.2 Repeated addition

Using the algorithm presented in Figure 2.2 each f_i, $2 \leq i \leq n$, is computed with the addition $f_i = f_{i-1} + f_{i-2}$. Each of these additions uses $\mathcal{N}(n-1)$ bit operations so the total number of bit operations needed to compute f_n is

$$\begin{aligned} T(n) &= T(n-1) + \mathcal{N}(n-1), \; T(0) = 0, \; T(1) = 0 \\ &= \sum_{i=2}^{n} \mathcal{N}(i-1) \\ &= \mathcal{N}\sum_{i=0}^{n-2}(i+1) \\ &= \mathcal{N}\left(\frac{(n-2)(n-1)}{2}\right) + \mathcal{N}(n-1) \\ &= \mathcal{N}\left(\frac{n^2}{2} - \frac{n}{2}\right) \quad (3.3) \end{aligned}$$

For the generalized Fibonacci numbers, f_n^k, a similar algorithm using k additions at each step will use about $\mathcal{N}\frac{kn^2}{2}$ bit operations.

$$\begin{aligned} T_k(n) &= T_k(n-1) + (k-1)\mathcal{N}(n-1),\ T(0)=0,\ T(1)=0 \\ &= \sum_{i=2}^{n}(k-1)\mathcal{N}(i-1) \\ &= \mathcal{N}\left(\frac{(k-1)(n-2)(n-1)}{2}\right) + \mathcal{N}(n-1) \\ &= \mathcal{N}\left(\frac{(k-1)n^2}{2} - \frac{(k-1)n}{2}\right) \end{aligned} \tag{3.4}$$

When $k = 2$ (the Fibonacci numbers) we get

$$T_2(n) = \mathcal{N}\left(\frac{n^2}{2} - \frac{n}{2}\right)$$

the same as equation 3.3 above.

The algorithm presented in Figure 2.3 for computation of f_n^k decreases the number of bit operations by a factor of $(k-1)/2$. The value of $\mathcal{N}$ will be different for different values of k so when $k \neq 2$ the constant will be denoted $\mathcal{N}_k$. $\mathcal{N}_k$ will always be less than one. Since the multiply is by two, a shift can be used instead and the computation of the next f_n^k will use two $\mathcal{N}_k(n-1)$ bit instructions, a shift and a subtract.

$$\begin{aligned} T_k(n) &= T_k(n-1) + 2\mathcal{N}_k(n-1),\ \ T(0)=0,\ T(1)=0 \\ &= \sum_{i=2}^{n} 2\mathcal{N}_k(i-1) \\ &= \sum_{i=0}^{n-2} 2\mathcal{N}_k(i+1) \\ &= 2\mathcal{N}_k(n-1) + 2\mathcal{N}_k\sum_{i=0}^{n-2} i \\ &= 2\mathcal{N}_k\left(n-1+\frac{(n-2)(n-1)}{2}\right) \\ &= \mathcal{N}_k(n^2-n) \end{aligned} \tag{3.5}$$

3.3 Binet's Formula

There are three operations that must be considered to count the bits operations needed to compute f_n using the algorithm presented in figure 2.4. The $\mathcal{N}n + \log n$

bits of the square root of five must be computed, an $\mathcal{N}n + \log n$ bit number must be exponentiated to the nth power, and a $\mathcal{N}n + \log n$ bit division must be done.

The $\sqrt{5}$ can be computed using $\log \mathcal{N}n + \log n$ iterations of Newton's method [13]. Each iteration of Newton's method requires a division and the length of the significant result doubles with each iteration. It will be assumed that a division requires n^2 bit operations, denoted as D(n).

$$\begin{aligned} T(n) &= T(n/2) + D(\mathcal{N}n/2) \\ &= \frac{(\mathcal{N}n)^2}{4} + \frac{(\mathcal{N}n)^2}{16} + \frac{(\mathcal{N}n)^2}{64} + \cdots \\ &= \sum_{i=1}^{\infty} \frac{(\mathcal{N}n)^2}{2^{2i}} \\ &= (\mathcal{N}n)^2 \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i \\ &= \frac{(\mathcal{N}n)^2}{3} \end{aligned} \tag{3.6}$$

During each $\log i$ iterations of the exponentiation, only the most significant $\mathcal{N}n + \log n$ bits need be kept. Therefore the time needed to perform the exponentiation is $\log n(\mathcal{N}n + \log n)^2$ bit operations.

The operands for the final division are $\mathcal{N}n$ bits and $\mathcal{N}n + \log n$ bits so the division will take less than $(\backslash n + \log n)^2$ bit operations.

the total number of bit operations using the algorithm presented in figure 2.4 will be dominated by the $\log n(\mathcal{N}n + \log n)^2$ bit operations used by the exponentiation giving $\Theta(\log n \cdot M(n))$ bit operations.

3.4 Matrix Multiplication Methods

In this section, and the next, the matrix algorithms and the extended N. N. Vorob'ev algorithm will be analyzed producing a sequence of the number of bit operations to compute f_n from $\frac{8(n\mathcal{N})^2}{3}$ down to $\frac{5(n\mathcal{N})^2}{12}$. Similar methods are used to find the running time for each of the algorithms in this section and the next so one general introduction is presented here and the specifics for each algorithm is presented in the appropriate sections.

All of the algorithms use some number of additions and some other number of multiplications. Since the additions can be done in time linear with respect to the size of the operands and it will be assumed that multiplication requires n^2 time for operands of size n, the bit operations resulting from the additions will be ignored and only those resulting from multiplications will be counted.

The simple matrix multiplication algorithm introduced in the first course in linear algebra requires *row·column* multiplications of size $n/2$ for a resulting matrix with elements of size n. In the case of computing

$$A^n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

this method would require eight multiplications of size $n/2$ to compute the matrix

$$\begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

from the matrix

$$\begin{pmatrix} f_{n/2+1} & f_{n/2} \\ f_{n/2} & f_{n/2-1} \end{pmatrix}$$

by squaring the second matrix. The number of bit operations needed to compute the matrix containing f_n using the above method with eight multiplies is $\frac{8 \cdot M(n\mathcal{N})}{3}$.

$$\begin{aligned}
T(n) &= T(n/2) + 8M(\mathcal{N}n/2) \\
&= 8M(\mathcal{N}n/2) + 8M(\mathcal{N}n/4) + 8M(\mathcal{N}n/8) + \cdots \\
&= \frac{8(\mathcal{N}n)^2}{4} + \frac{8(\mathcal{N}n)^2}{16} + \frac{8(\mathcal{N}n)^2}{64} + \cdots \\
&= 8(\mathcal{N}n)^2 \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i \\
&= 8(\mathcal{N}n)^2 \cdot \frac{1}{3} \\
&= \frac{8(\mathcal{N}n)^2}{3}
\end{aligned} \tag{3.7}$$

Since we can compute the first row of A^{n+1} from A^n using four multiplications and the second row can be computed using only an addition and four multiplications

are needed. Using a sequence of operations similar to those above we get $\frac{4 \cdot M(n\mathcal{N})}{3}$ bit operations.

$$\begin{aligned} T(n) &= T(n/2) + 4M(\mathcal{N}n/2) \\ &= \frac{4(\mathcal{N}n)^2}{3} \end{aligned} \tag{3.8}$$

3.4.1 Three Multiply Matrix Method

The three multiply matrix method presented in figure 2.6 performs three half size multiplies, three additions, and calls itself recursively with half size arguments. Since we are ignoring the additions we get $M(n\mathcal{N})$ bit operations.

$$\begin{aligned} T(n) &= T(n/2) + 3M(\mathcal{N}n/2) \\ &= 3M(\mathcal{N}n/2) + 3M(\mathcal{N}n/4) + 3M(\mathcal{N}n/8) + \cdots \\ &= \frac{3(\mathcal{N}n)^2}{4} + \frac{3(\mathcal{N}n)^2}{16} + \frac{3(\mathcal{N}n)^2}{64} + \cdots \\ &= \sum_{i=1}^{\infty} \frac{3(\mathcal{N}n)^2}{2^{2i}} \\ &= 3(\mathcal{N}n)^2 \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i \\ &= 3(\mathcal{N}n)^2 \cdot \frac{1}{3} \\ &= (\mathcal{N}n)^2 \end{aligned} \tag{3.9}$$

3.4.2 Two Multiply Matrix Method

The two multiply matrix method presented in figure 2.7 does two half size multiplies, four additions, one shift, and calls itself recursively with a half size argument. The number of bit operations used by this algorithm is $\frac{2 \cdot M(\mathcal{N}n)}{3}$.

$$\begin{aligned} T(n) &= T(n/2) + 2M(\mathcal{N}n/2) \\ &= 2M(\mathcal{N}n/2) + 2M(\mathcal{N}n/4) + 2M(\mathcal{N}n/8) + \cdots \\ &= \frac{2(\mathcal{N}n)^2}{4} + \frac{2(\mathcal{N}n)^2}{16} + \frac{2(\mathcal{N}n)^2}{64} + \cdots \\ &= \sum_{i=1}^{\infty} \frac{2(\mathcal{N}n)^2}{2^{2i}} \end{aligned}$$

$$\begin{aligned}
&= 2(\mathcal{N}n)^2 \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i \\
&= 2(\mathcal{N}n)^2 \cdot \frac{1}{3} \\
&= \frac{2(\mathcal{N}n)^2}{3}
\end{aligned} \tag{3.10}$$

3.5 Extended N. N. Vorob'ev's Method

By using the results of computing f_{2i} to compute f_{2i+1} the algorithm presented in figure 2.8 is able to replace one $\frac{n}{2}$ bit multiply with two $\frac{n}{4}$ bit multiplies each time it is called. The extended N. N. Vorob'ev algorithm uses $\frac{M(\mathcal{N}n)}{2}$ bit operations.

$$\begin{aligned}
T(n) &= T(n/2) + M(\mathcal{N}n/2) + 2M(\mathcal{N}n/4) \\
&= M(\mathcal{N}n/2) + 3M(\mathcal{N}n/4) + 3M(\mathcal{N}n/8) + \cdots \\
&= \frac{(\mathcal{N}n)^2}{4} + \frac{3(\mathcal{N}n)^2}{16} + \frac{3(\mathcal{N}n)^2}{64} + \cdots \\
&= \frac{(\mathcal{N}n)^2}{4} + \sum_{i=2}^{\infty} \frac{3(\mathcal{N}n)^2}{2^{2i}} \\
&= \frac{(\mathcal{N}n)^2}{4} + 3(\mathcal{N}n)^2 \sum_{i=2}^{\infty} \left(\frac{1}{4}\right)^i \\
&= \frac{(\mathcal{N}n)^2}{4} + 3(\mathcal{N}n)^2 \cdot \frac{1}{12} \\
&= \frac{(\mathcal{N}n)^2}{2}
\end{aligned} \tag{3.11}$$

3.6 Product of Factors

The product of factors algorithm presented in section 2.9 uses two multiplies at each recursive call. One multiply is to compute β_{i-1} by squaring a $n/4$ bit number and the other is to compute f_n by multiplying β_{i-1} by the product of $\beta_1, \beta_2, \ldots, \beta_{i-2}$ both $n/2$ bit numbers. The number of bit operations this algorithm uses to compute the n_{th} Fibonacci number is $\frac{5 \cdot M(\mathcal{N}n)}{12}$

$$T(n) = T(n/2) + M(\mathcal{N}n/2) + M(\mathcal{N}n/4)$$

$$\begin{aligned}
&= M(\mathcal{N}n/2) + 2M(\mathcal{N}n/4) + 2M(\mathcal{N}n/8) + \cdots \\
&= \frac{(\mathcal{N}n)^2}{4} + \frac{2(\mathcal{N}n)^2}{16} + \frac{2(\mathcal{N}n)^2}{64} + \cdots \\
&= \frac{(\mathcal{N}n)^2}{4} + 2(\mathcal{N}n)^2 \sum_{i=2}^{\infty} \left(\frac{1}{4}\right)^i \\
&= \frac{(\mathcal{N}n)^2}{4} + 2\frac{(\mathcal{N}n)^2}{12} \\
&= \frac{5(\mathcal{N}n)^2}{12} \qquad (3.12)
\end{aligned}$$

3.7 f_n for any n using Product of Factors

The algorithm in Figure 2.13 computes the three sequences

$$\beta_1, \beta_2, \beta_3, \ldots, \beta_i$$

$$f_3, f_7, f_{15}, \ldots, f_{2^i-1}$$

$$f_4, f_8, f_{16}, \ldots, f_{2^i}$$

for $i = \lceil \log_2 n \rceil - 1$. The worst case for computing these sequences is $n = 2^i + 1$ where the size of β_i is about half of the size of f_n with f_{2^i-1} and f_{2^i} about the same size as f_n. The best case for this computation is $n = 2^i$, where β_i is about one fourth the size of f_n with f_{2^i-1} and f_{2^i} half of the size of f_n. We will see that the best and worst cases are switched for the second phase of this algorithm.

In the worst case the number of bit operations for the the first phase, computing the three sequences, of the algorithm presented in figure 2.13 is

$$\begin{aligned}
T(n) &= T(n/2) + 2M(\mathcal{N}n) + M(\mathcal{N}n/2) \\
&= 2M(\mathcal{N}n) + 3M(\mathcal{N}n/2) + 3M(\mathcal{N}n/4) + \cdots \\
&= 2(\mathcal{N}n)^2 + \frac{3(\mathcal{N}n)^2}{4} + \frac{3(\mathcal{N}n)^2}{16} + \cdots \\
&= 2(\mathcal{N}n)^2 + \sum_{i=1}^{\infty} \frac{3(\mathcal{N}n)^2}{2^{2i}} \\
&= 2(\mathcal{N}n)^2 + 3(\mathcal{N}n)^2 \cdot \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i \\
&= 2(\mathcal{N}n)^2 + (\mathcal{N}n)^2
\end{aligned}$$

$$= 3(\mathcal{N}n)^2 \tag{3.13}$$

In the second phase of the algorithm (the call to fib-help) the various values of the two sequences are multiplied together to get f_n. In this phase the worst case will be when the numbers being multiplied together are about the same size, or about one half the length of f_n. The number of bit operations for the second phase is at most

$$\begin{aligned}
T(n) &= T(n/2) + 2M(\mathcal{N}n/2) \\
&= 2M(\mathcal{N}n/2) + 2M(\mathcal{N}n/4) + 2M(\mathcal{N}n/8) + \cdots \\
&= \frac{2(\mathcal{N}n)^2}{4} + \frac{2(\mathcal{N}n)^2}{16} + \frac{2(\mathcal{N}n)^2}{64} + \cdots \\
&= \sum_{i=1}^{\infty} \frac{2(\mathcal{N}n)^2}{2^{2i}} \\
&= 2(\mathcal{N}n)^2 \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i \\
&= 2(\mathcal{N}n)^2 \cdot \frac{1}{3} \\
&= \frac{2(\mathcal{N}n)^2}{3} \qquad (3.14)
\end{aligned}$$

So, the worst case number of bit operations for the algorithm presented in figure 2.13 is certainly less than $\frac{11(\mathcal{N}n)^2}{3}$.

3.8 Matrix Method of Gries and Levin

The algorithm of Gries and Levin presented in section 2.4 exponentiates the $k \times k$ matrix A using only k^2 multiplications at each step. Each iteration of the algorithm doubles n so only $\log n$ iterations are required to compute f_n^k. This leads to the equation

$$\begin{aligned}
T_k(n) &= T_k(n/2) + k^2 M(\mathcal{N}n/2) \\
&= k^2 M(\mathcal{N}n/2) + k^2 M(\mathcal{N}n/4) + k^2 M(\mathcal{N}n/8) + \cdots \\
&= \frac{k^2(\mathcal{N}n)^2}{4} + \frac{k^2(\mathcal{N}n)^2}{16} + \frac{k^2(\mathcal{N}n)^2}{64} + \cdots \\
&= k^2(\mathcal{N}n)^2 \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i
\end{aligned}$$

$$= \frac{k^2(\mathcal{N}n)^2}{3} \tag{3.15}$$

For $k = 2$ (the fibonacci sequence)

$$T_2(n) = \frac{4(\mathcal{N}n)^2}{3} \tag{3.16}$$

or $\frac{4 \cdot M(\mathcal{N}n)}{3}$.

3.9 Binomial Coefficients

Computing f_{n+1} by taking the sum of the nth diagonal of Pascal's triangle requires computing the $n+1$ binomial coefficients in the nth diagonal. The two methods presented in section 2.7 will be analyized, but since the optimal method of computing binomial coefficients is not known there may be better algorithms.

Using the standard method of computing binomial coefficients, see equation 2.16, requires computing $k(n-k) - \frac{k^2}{2}$ elements of Pascal's triangle. There are $k(n-k)$ elements added to compute $\begin{pmatrix} n \\ k \end{pmatrix}$ but since Pascal's triangle is symmetrical, Equation 2.15 only the elements on one side of the center of Pascal's triangle need to be computed. From the diagram in figure 3.1 it can be seen that the largest number of elements will need to be computed when $k = \frac{n}{2}$. These elements also have the largest values since they are closest to the center of Pascal's triangle. The total number of elements of Pascal's triangle that need to be computed in order to compute $\begin{pmatrix} n \\ k \end{pmatrix}$ is

$$n(n-k) - \frac{(n - |n/2 - k|)^2}{4}$$

and when $k = \frac{n}{2}$

$$\frac{n^2}{2} - \frac{\left(n - \left|\frac{n}{2} - \frac{n}{2}\right|\right)^2}{4}$$

$$= \frac{n^2}{2} - \frac{n^2}{4}$$

$$= \frac{n^2}{4}$$

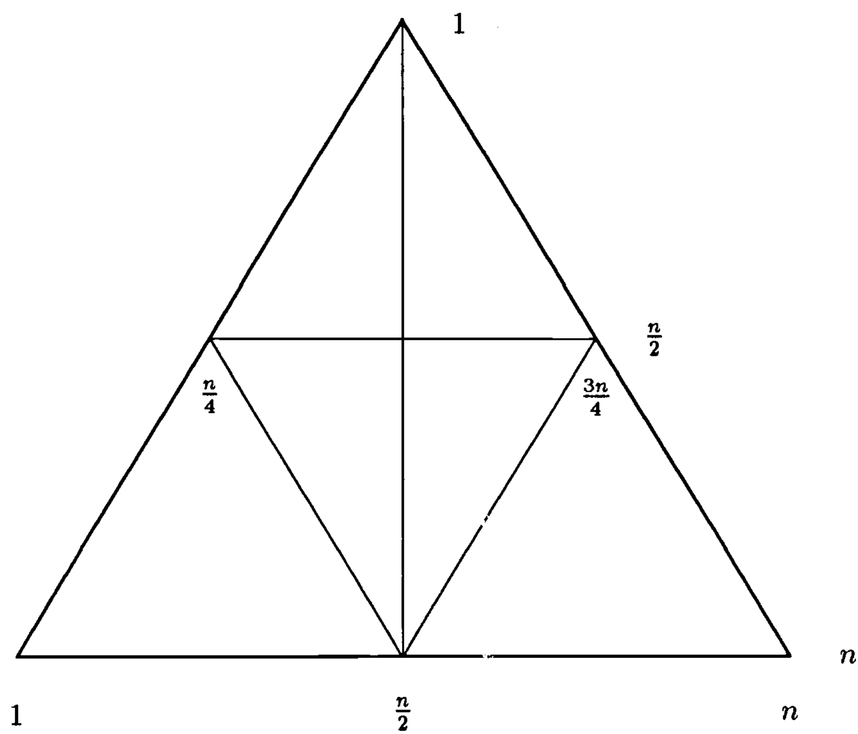

Figure 3.1: Elements of Pascal's triangle used to compute a binomial coefficient

So, in the worst case, there are $\frac{n^2}{4}$ elements to be added and each of these elements must be less than n bits long since the sum of the nth row of Pascal's triangle is equal to 2^n or

$$\sum_{i=1}^{n} \binom{n}{k} = 2^n,$$

therefore at most $\frac{n^3}{8}$ bit operations need to be done to compute $\binom{n}{k}$.

To compute f_n a complete diagonal of Pascal's triangle must be computed. Since the diagonal rises to the right all of the elements in the nth diagonal will be computed if $\binom{2n}{n}$ is computed. This will take, at most, $\frac{(2n)^3}{8} = n^3$ bit operations to compute. Clearly, this is a gross overestimate of the time required to compute f_{n+1} by taking the sum of the nth diagonal of Pascal's triangle.

A different way to compute binomial coefficients is with Goetgheluck's algorithm. Each call to the function given in figure 2.10 performs $\log_p n$ iterations. During each iteration two divisions of the same size are done, the size decreasing by a factor of p each iterations. The number of bit operations done by one call is less than

$$\begin{aligned}\sum_{i=1}^{\log_p n} 2 \cdot \frac{n}{p^i} \cdot p &= \sum_{i=1}^{\log_p n} 2 \cdot \frac{n}{p^{i-1}} \\ &= n + \frac{n}{p^1} + \frac{n}{p^2} + \frac{n}{p^3} + \cdots + 1\end{aligned}$$

Since the sum will be the largest when the prime number, $p = 2$, a call to the function in figure 2.10 will take no more than

$$\begin{aligned}& n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 1 \\ =\ & 2n\end{aligned}$$

bit operations. There will be at most n [1] calls made to the function, so the resulting number of bit operations is $\leq n^2$.

[1] The actual number is the number of primes $\leq n$ or about $\frac{n}{\log n}$ calls, but this is a slight underestimate and using n will not change the order.

Once the factors are computed they must be multiplied together to form the binomial coefficient. In the worst case we would have to multiply two half size numbers to get the result, and four quarter size numbers to get the two half size numbers and so on.

$$\begin{aligned}
T(n) &= 2T(n/2) + M(n/2) \\
&= M(n/2) + 2M(n/4) + 4M(n/8) + \cdots \\
&= \frac{n^2}{4} + \frac{n^2}{16} + \frac{n^2}{32} + \cdots \\
&= n^2 \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i \\
&= \frac{n^2}{3}
\end{aligned}$$

Adding the time to compute the factors and the time to multiply the factors, we get the time to compute a binomial coefficient $\leq \frac{4n^2}{3}$.

To compute f_{n+1}, the $\left\lceil \frac{n}{2} \right\rceil$ binomial coefficients of the dth diagonal need to be computed.

$$\begin{aligned}
T(n) &= \sum_{i=n/2}^{n} \frac{4i^2}{3} \\
&= \frac{4}{3} \sum_{i=1}^{n/2} \left(i + \frac{n}{2}\right)^2 \\
&= \frac{4}{3} \sum_{i=1}^{n/2} i^2 + n + \frac{n^2}{4} \\
&= \frac{4}{3} \left(\frac{\frac{n}{2}(\frac{n}{2}+1)(\frac{n}{2}+1)}{6} + \frac{n^2}{2} + \frac{n^3}{8} \right) \\
&= \frac{n^3}{12} + \frac{5n^2}{3} + n
\end{aligned}$$

Again, this is gross overestimate of the number of bit operations needed to compute f_{n+1} and is not useful to compare with number of bit operations needed to compute f_{n+1} using the standard method.

Algorithm	Constant
Repeated Addition	4.223 E -6
Product of Factors (any f_n)	3.273 E -8
Binet's Formula (any f_n)	1.06 E -1
Binet's formula (f_n, $n = 2^k$)	7.038 E - 5
Product of Factors (f_n, $n = 2^k$)	1.633 E - 8
3 Multiply Matrix	3.950 E - 8
2 Multiply Matrix	2.667 E - 8
Extended Vorob'ev	1.983 E - 8

Table 3.1: Constants for algorithms to compute f_n

3.10 Actual running times

All of the algorithms discussed in chapters two and three have been implemented in Ibuki common lisp, formerly Kyoto common lisp (KCL), on a Sequent Balance 21000. The functions are divided into two groups, those that compute f_n for any n and those that compute f_n, $n = 2^k$ where k is an integer. Each function computed f_n, where n was successive powers of two, until the function used more than five minutes of CPU time. The results for the functions that computed f_n for any n are given in Table 3.2 and for the functions that computed f_n, $n = 2^k$ where k is an integer, are given in Table 3.3 and Table 3.4.

A constant was computed for each algorithm that would make the actual running time of the algorithm equal to the computed running time derived earlier in this chapter. The constants were computed by dividing the actual running time of the largest Fibonacci number computed by the number of bit operations predicted by the analysis. Table 3.1 presents these constants.

$\log n$	Repeated Addition		Product of Factors		Binet's Formula	
	comp	act	comp	act	comp	act
2	0.00–0.00		0.00–0.00		3.82–0.08	
3	0.00–0.00		0.00–0.02		12.82–0.13	
4	0.00–0.02		0.00–0.02		42.40–1.60	
5	0.00–0.00		0.00–0.02		145.11–44.28	
6	0.02–0.03		0.00–0.03		519.43–519.43	
7	0.07–0.07		0.00–0.05		na–na	
8	0.28–0.22		0.00–0.05		na–na	
9	1.10–0.62		0.01–0.08		na–na	
10	4.42–5.17		0.03–0.17		na–na	
11	17.71–16.73		0.14–3.28		na–na	
12	70.84–71.30		0.55–6.97		na–na	
13	283.39–279.52		2.20–9.60		na–na	
14	1133.62–1133.62		8.79–17.45		na–na	
15	na–na		35.14–49.92		na–na	
16	na–na		140.57–154.40		na–na	
17	na–na		562.28–562.28		na–na	

Table 3.2: Running time to compute f_n for any n in CPU seconds

$\log n$	Binet's formula		Product of Factors	
	comp	act	comp	act
2	0.00–0.02		0.00–0.00	
3	0.01–0.02		0.00–0.02	
4	0.03–0.03		0.00–0.00	
5	0.10–0.05		0.00–0.00	
6	0.34–0.13		0.00–0.00	
7	1.28–0.75		0.00–0.00	
8	4.91–14.83		0.00–0.02	
9	19.10–11.10		0.00–0.02	
10	75.25–77.82		0.01–0.05	
11	298.38–287.70		0.07–0.12	
12	1187.73–1187.73		0.27–0.37	
13	na–na		1.10–1.23	
14	na–na		4.38–4.60	
15	na–na		17.53–17.77	
16	na–na		70.12–73.98	
17	na–na		280.47–280.22	
18	na–na		1121.88–1121.87	

Table 3.3: Running time to compute f_n, $n = 2^k$, in CPU seconds, part 1

<table>
<tr><th rowspan="2">log n</th><th colspan="2">3 Multiply Matrix</th><th colspan="2">2 Multiply Matrix</th><th colspan="2">Extended Vorob'ev</th></tr>
<tr><th>comp</th><th>act</th><th>comp</th><th>act</th><th>comp</th><th>act</th></tr>
<tr><td>2</td><td colspan="2">0.00–0.02</td><td colspan="2">0.00–0.02</td><td colspan="2">0.00–0.00</td></tr>
<tr><td>3</td><td colspan="2">0.00–0.02</td><td colspan="2">0.00–0.00</td><td colspan="2">0.00–0.00</td></tr>
<tr><td>4</td><td colspan="2">0.00–0.00</td><td colspan="2">0.00–0.02</td><td colspan="2">0.00–0.02</td></tr>
<tr><td>5</td><td colspan="2">0.00–0.00</td><td colspan="2">0.00–0.00</td><td colspan="2">0.00–0.00</td></tr>
<tr><td>6</td><td colspan="2">0.00–0.02</td><td colspan="2">0.00–0.02</td><td colspan="2">0.00–0.00</td></tr>
<tr><td>7</td><td colspan="2">0.00–0.03</td><td colspan="2">0.00–0.03</td><td colspan="2">0.00–0.02</td></tr>
<tr><td>8</td><td colspan="2">0.00–0.05</td><td colspan="2">0.00–0.05</td><td colspan="2">0.00–0.03</td></tr>
<tr><td>9</td><td colspan="2">0.01–0.05</td><td colspan="2">0.01–0.05</td><td colspan="2">0.01–0.03</td></tr>
<tr><td>10</td><td colspan="2">0.04–0.10</td><td colspan="2">0.03–0.10</td><td colspan="2">0.02–0.07</td></tr>
<tr><td>11</td><td colspan="2">0.17–0.28</td><td colspan="2">0.11–0.25</td><td colspan="2">0.08–0.18</td></tr>
<tr><td>12</td><td colspan="2">0.66–0.83</td><td colspan="2">0.45–0.70</td><td colspan="2">0.33–0.47</td></tr>
<tr><td>13</td><td colspan="2">2.65–2.92</td><td colspan="2">1.79–5.62</td><td colspan="2">1.33–1.58</td></tr>
<tr><td>14</td><td colspan="2">10.60–14.52</td><td colspan="2">7.16–7.77</td><td colspan="2">5.32–9.23</td></tr>
<tr><td>15</td><td colspan="2">42.41–42.85</td><td colspan="2">28.63–29.35</td><td colspan="2">21.29–21.63</td></tr>
<tr><td>16</td><td colspan="2">169.66–172.65</td><td colspan="2">114.54–117.97</td><td colspan="2">85.17–84.98</td></tr>
<tr><td>17</td><td colspan="2">678.63–678.63</td><td colspan="2">458.15–458.15</td><td colspan="2">340.68–340.68</td></tr>
</table>

Table 3.4: Running time to compute f_n, $n = 2^k$, in CPU seconds, part 2

Chapter 4

Redundant Information

Since so few of the integers are Fibonacci numbers we would expect redundant information in the integer representation of the Fibonacci numbers. If we could understand the structure of this redundancy, we could take advantage of it in computing f_n by using a representation using fewer bits than the integers. This chapter will present three compression methods that were run on a string of bits representing the sequence of ascending Fibonacci numbers. The results of compressing the Fibonacci sequence yielded no insights into any redundant structure that might exist in the Fibonacci sequence. The compression algorithms behaved as they would if given a random sequence of bits to compress, so the bits of the Fibonacci sequence were tested as a random sequence. The three tests for randomness that were performed indicated that the sequence of bits was random.

The final section of this chapter looks at the Fibonacci numbers in a different arrangement, instead of placing the binary representation of the Fibonacci numbers end to end, a binary sequence was constructed from the bth bit of each Fibonacci number. Arranging the Fibonacci numbers in this manner shows that the bth bit forms a cycle of length $3 \cdot 2^b$. Unfortunately, the cycle length is exponential in b, the bit position, and since the number of bits in f_n grows linearly with n, the cycles seem to only be useful in computing the lower bits in f_n. If we could compute the value of the mth bit in the bth cycle quickly (i.e., without computing the entire cycle) we might be able to compute f_n quickly. Redundancy in the bth cycle was investigated and found but efforts to explain it have not yet been successful. This is described in the final section of this chapter.

4.1 Compression

The sequence of bits that will be compressed is obtained by writing, in binary, each Fibonacci number end to end. The bits of each f_n are written in reverse order, that is the most significant bit is to the right. The start of the sequence would be

$$f_1\ f_2\ f_3\ f_4\ f_5\ f_6 \cdots$$

$$1\ 1\ 2\ 3\ 5\ 8 \cdots$$

$$1\ 1\ 01\ 11\ 101\ 0001 \cdots$$

$$1101111010001 \cdots$$

The bits of the individual f_n are in reverse order so that when the sequence is read from left to right the low order bit of f_n will be read first, but the order should not be important as long as it is consistent.

The sequence of Fibonacci numbers were made available to the data compression algorithms in two forms, one bit at a time or one byte at a time. When a compression function requests the next source character, the leftmost unused bit (byte) in the sequence is removed from the sequence and is returned to the calling function. The next Fibonacci number is computed when the current source of bits (bytes) have been exhausted. This allows the possibility of analyzing very long sequences.

Terry Welch [38] discusses four types of redundancy: character distribution, character repetition, high usage patterns, and positional. If some characters appear more frequently than others in a sequence that sequence is said to have character distribution redundancy. For example, in the English language the characters 'e' and ' ' occur with greater frequency than the character 'z'. Character repetition redundancy occurs when the same character appears several times in succession. Frequently, files with fixed size records will use a blank, or some other distinguished character, to fill the records; these filler blanks are frequently a source of character repetition redundancy. High usage pattern redundancy is an extension of character distribution redundancy, instead of noting that some individual characters appear

	Compression Ratio			
k	integer	2^n	Fibonacci	random
128	0.508	0.821	0.533	0.533
256	0.577	2.667	0.520	0.561
512	0.569	2.510	0.520	0.547
1024	0.580	2.081	0.527	0.537
2048	0.616	2.216	0.524	0.532
4096	0.639	2.306	0.531	0.527
8192	0.680	2.421	0.530	0.534
16384	0.712	2.447	0.531	0.530
32768	0.747	2.496	0.532	0.531

Table 4.1: Compression ratios of Ziv–Lempel for various sources

more frequently than others, it is noted that some groups of characters appears more frequently than other groups of characters. A sequence is positionally redundant when the same character appears at the same position in each block of data. An example of positional redundancy would be a file containing a list of numbers of the form "XXXX.YY" since the '.' always appears as the fifth character. Positional redundancy will be important in section 4.3.

4.1.1 Ziv Lempel Compression

The Ziv Lempel (ZL) compression algorithm, [40,41,42], along with the LZW compression algorithm, [38], will be considered and the results of applying them to the sequence of Fibonacci bits will be discussed in this section.

A formal description and analysis of the Ziv Lempel compression algorithm is in [41], what follows is an informal presentation of the algorithm. If we have a array of characters, S, of length k and an integer j, $1 \leq j < k$, then we can match, character by character, the string at S[1] with the string starting at S[j]. Let $L(i)$

	Compression Ratio			
k	integer	2^n	Fibonacci	random
128	0.731	1.438	0.677	0.703
256	0.776	2.393	0.753	0.755
512	0.806	3.483	0.791	0.782
1024	0.861	4.697	0.811	0.811
2048	0.838	6.564	0.841	0.830
4096	0.848	8.480	0.842	0.842
8192	0.854	11.977	0.851	0.849
16384	0.866	15.326	0.859	0.858
32768	0.870	21.530	0.868	0.867

Table 4.2: Compression ratios of the modified Lempel–Ziv–Welch algorithm

	Compression Ratio			
k	integer	2^n	Fibonacci	random
128	0.871	1.707	0.871	0.871
256	0.880	3.413	0.880	0.880
512	0.838	5.069	0.839	0.845
1024	0.815	7.758	0.809	0.805
2048	0.783	11.253	0.763	0.765
4096	0.749	16.190	0.725	0.725
8192	0.721	22.506	0.698	0.694
16384	0.718	30.972	0.678	0.678
32768	na	43.691	na	na

Table 4.3: Compression ratios for Unix Ziv-Lempel implementation

1. Initilize the array of characters, B
2. Compute the largest reproducible extension
3. Generate the code word
4. If not done update the array B and goto step 2

Figure 4.1: Pseudo-code for ZL compression algorithm

be the length of this match where the strings being matched start at location i and $j+1$. For example, let S contain 00101011 and let $j = 3$ so we have

$$001 \cdot 01011$$

So $L(1) = 1$ since S[1] = S[4] and S[2] $\neq$ S[5]; $L(2) = 4$ since S[2] = S[4], S[3] = S[5], S[4] = S[6], S[5] = S[7], and S[6] $\neq$ S[8]; $L[3] = 0$ since S[3] $\neq$ S[4]. The largest reproducible extension is the longest substring of S starting at $j+1$ that matches with some substring of S starting between 1 and j. That is, the string starting at location p such that $L(p) = \max_{1 \leq i \leq p} L(i)$. Knowing p, called the pointer of the reproduction, and the length of the reproduction, l, we can store just p, l and S[j] instead of the substring S[j]...S[j+l].

An outline of the ZL algorithm is given in figure 4.1. The first step is to initilize the array B with zeros followed by the first characters from the source where j is the position of the last 0, see [41] for details about the position of j in B. The second step is to compute the largest reproducible extension as discussed above.

The third step in the algorithm is to compute the code word C_i composed of three parts C_{i1}, C_{i2}, and C_{i3}. C_{i1} will represent p_i, C_{i2} will represent the length of the longest reproducible extension, l, and C_{i3} will be the character immediately following the reproducible extension. Again, representation details may be found in [41].

The final step is to shift out the first l characters in the buffer B and add l characters from the source to the end of the buffer. The cycle of executing steps

two, three, and four continues until there are no more characters from the source.

Various sequences were compressed: the sequence formed by concatenating the binary representation of the non-negative integers, the sequence formed by concatenating the binary representations of 2^i for $i = 0, i = 1, i = 2, \ldots$, the sequence formed by concatenating the binary representation of f_n for $n = 0, n = 1, n = 2, \ldots$, and the sequence formed by concatenating the binary representation of the numbers generated by the random number generator in Ibuki common lisp. In Table 4.1 through Table 4.4, k is the number of bits in the sequence that were compressed. The compression ratio is the size of the input file divided by the size of the resulting compressed file. A compression ratio that is greater than one indicates the compressed file is smaller than the original file. A compression ration that is less than one indicates that the compressed file is larger than the original file.

Three versions of the Ziv–Lempel algorithm we used to compress the four sequences. The first version is a direct implementation of the Ziv–Lempel algorithm presented in [41]. There are several parameters that need to be set for this algorithm, two that have a large effect on the compression ration are the size of the array B, and j, the position in B to start looking for the reproducible extension. The size of B was 96 and j was 32. Since a pointer into the first $|B| - j$ elements of B must be stored in binary $|B| - j$ should be a power of two and since the length of the longest possible extension, l, must be less than or equal to j, j should also be a power of two. The values of 96 and 32 were selected from all of the reasonable combinations of $j = 2^x$ and $|B| - j = 2^y$ for $2 \leq x, y \leq 12$, $x < y$. The results of running the Ziv–Lempel compression program using other values for x and y are available upon request.

The compression ratios for the direct implementation of the Ziv–Lempel algorithm are presented in figure 4.1. The integers do not compress well, they actually expand, but as k increases the compression ratio improves. This is because the leading bits of the successive numbers remain the same for some time and the algorithm exploits this high usage redundancy. Since the sequence of 2^n is mostly zeros, it should be easily compressible. The results show that the compression ratio

	Compression Ratio			
k	integer	2^n	Fibonacci	random
128	0.908	2.723	0.914	0.914
256	0.911	4.491	0.921	0.911
512	0.961	5.447	0.919	0.919
1024	0.985	6.206	0.923	0.921
2048	1.009	6.759	0.938	0.941
4096	0.998	7.186	0.957	0.958
8192	1.008	7.441	0.972	0.972
16384	1.026	7.620	0.983	0.984
32768	na	7.743	na	na

Table 4.4: Compression ratios of the dynamic Huffman code algorithm

does not get much higher than 2.5 due to the choice of the parameters s and y that was made above. The Fibonacci sequence and random number sequence behave remarkably similarly, neither compressing well.

The modified Lempel–Ziv–Welch implementation and the Unix Ziv–Lempel implementation both showed similar compression ratio patterns as above. The results are presented in table 4.2 and table 4.3. Since neither of these algorithms use a static buffer they are capable of taking better advantage of the long strings of zeros in the sequence of 2^n. Again, note the similarity of compression ratios of the Fibonacci sequence and the random sequence.

4.1.2 Dynamic Huffman Codes

The standard Huffman encoding algorithm scans the data twice, the first time to find the frequencies of each of the characters in the message, and a second time to encode the message. First, a tree is constructed with each of the characters in the message assigned to a leaf with the leaf is assigned a weight proportional to the

store the k leaves in a list L

While L contains at least two nodes do

remove the two nodes x and y of smallest weight from l

create a new node p and make p the parent of x and y

p's weight $\leftarrow$ x's weight + y's weight

insert p into L

end

Figure 4.2: Huffman's algorithm to compute letter frequencies

frequency that character appeared in the message. An algorithm to construct this Huffman tree from [35] is given in figure 4.2. The result of this algorithm is a single node that is a root of a binary tree with minimum weighted external path length among all binary trees for the given leaves [35].

When traversing the Huffman tree choosing a branch to the left is represented by a zero and choosing a branch to the right is represented by a one. The second pass over the data creates the coded message by emitting the path that would be traversed in the Huffman tree from the root to the leaf representing the character.

To use this algorithm the encoder first transmits the Huffman tree and then the encoded message. The decoder, or receiver, will construct the tree sent to it and then follow the string of zeros and ones going left or right in the Huffman tree as appropriate until reaching a leaf and emitting the character associated with that leaf. The decoder then starts at the root of the tree again to decode the next character and so on until the entire message is decoded.

A disadvantage to this two pass Huffman compression algorithm is the necessity to first transmit the Huffman tree and then encode the text. A dynamic Huffman compression algorithm will create the Huffman tree on the fly as the message is being encoded. The frequencys used to encode the kth character are the frequencies of the previous $k-1$ characters. The encoder and decoder of the mes-

sage both build the tree at the same time, as the text is sent. A disadvantage of using the one pass Huffman compression algorithm is that the beginning of the message is not encoded efficiently since there is little information about the distribution of the characters in the text. For long sequences of uniformly distributed characters this is not a problem since the algorithm relatively quickly "learns" the distribution of the characters, but, if the message is short or the character distribution changes in the message then the compression will be poor since the algorithm never has an accurate Huffman tree to work from.

For the purpose of extracting redundant information in a long sequence, either method will be satisfactory. The **compact** command in the BSD 4.2 Unix operating system compresses file using the dynamic Huffman compression algorithm briefly described above. The sequences were written to a file and then used as input to **compact**. The results of this type of compression on the sequences are presented in table 4.4. Again, as with the various version of Ziv–Lempel compression, the compression ratios of the Fibonacci numbers and the random numbers are very close.

4.2 Testing for Randomness

Compression of the Fibonacci sequence using the standard methods that were tried did not reveal any redundant information in the binary representation of the Fibonacci numbers. Since the methods of compression that were used were designed to remove redundant information from *non*-random sequences (there is no redundant information in a random sequence) and no compression was found, the sequence of Fibonacci numbers was tested for randomness. Three tests were chosen: the equidistribution test, the serial test, and the Wald-Wolfowitz test. Each method was implemented and at least three sequences were tested: a sequence of pseudo-random numbers generated by the **random** function of Ibuki Common Lisp using the linear congruential method, a sequence consisting of a single character, and a sequence of Fibonacci numbers. The results are presented in table 4.5, table 4.6,

Equidistribution χ^2/df			
k	Fibonacci	Random	Non-Random
1000	0.995	0.901	1000
1500	1.115	0.846	1500
2000	1.033	1.049	2000
2500	0.893	0.985	2500
3000	0.979	1.048	3000
3500	0.953	0.903	3500
4000	1.034	0.974	4000
4500	0.974	0.914	4500
5000	0.968	0.980	5000
5500	1.010	1.194	5500
6000	1.027	1.002	6000

Table 4.5: Equidistribution test for randomness

and table 4.7.

4.2.1 Equidistribution

The equidistribution test or frequency test, tests the sequence of numbers for uniform distribution between 0 and d. For each number r, $0 \leq r \leq d$, a count, C_r, of the number of occurrences of r in the sequence is kept and should be evenly distributed. To test the evenness of the distribution, apply the chi-square test with the probability of $\frac{1}{d}$ for each r.

Table 4.5 gives the value of χ^2/df where df equals the degrees of freedom for the number of occurrences each of the 256 possible eight bit words using n, from 1000 to 6000. The hypothesis that we are testing is that each of the 256 possible eight bit words is equally likely to occur. For n = 1000 if $0.899 < \chi^2/df < 1.11$ then we can

Serial test χ^2/df			
k	Fibonacci	Random	Non-Random
300	0.952	0.921	90820
600	0.921	0.905	big
900	0.937	0.952	big
1200	0.937	0.889	big
1500	0.921	0.937	big
1800	0.937	0.905	big
2100	0.952	0.937	big
2400	0.873	0.857	big
2700	0.921	0.937	big
3000	0.937	0.921	big
3300	0.921	0.921	big

Table 4.6: Serial test for randomness

Wald-Wolfowitz test Z			
k	Fibonacci	Random	Non-Random
1000	-9.237	-7.907	22327
2000	-12.419	-15.353	big
3000	-16.457	-14.433	big
4000	-20.429	-20.986	big
5000	-22.163	-22.643	big
6000	-22.691	-25.529	big
7000	-26.864	-26.742	big
8000	-27.650	-28.411	big
9000	-30.172	-29.052	big
10000	-29.988	-30.278	big
11000	-32.031	-30.988	big

Table 4.7: Wald-Wolfowitz test for randomness

not reject the hypothesis at the 99% level that the words are equally distributed. The hypothesis can be rejected, that is, the words are not equally distributed, for several entries in the table at the 99% level. The Fibonacci sequence when n = 1500 and n = 2500 and the random sequence when n = 1500, n = 3500, n = 4500, and n = 5500.

4.2.2 Serial Test

Equal distribution is not enough to be reasonably sure that a sequence of numbers is random. The sequence of numbers 1, 2, 3, 1, 2, 3, 1, 2, 3, $\cdots$ is equally distributed but not random. The serial test will measure the frequency of pairs of numbers following one another. For each pair of numbers r_1 and r_2, $0 \leq r_1, r_2 \leq d$ a count C_{r_1,r_2} of the number of occurrences of r_1 immediately followed by r_2 is kept and should be evenly distributed over the d^2 pairs of r_1 and r_2. To test the evenness of the distribution, the chi-square test was applied with the probability of $\frac{1}{d^2}$ for each category. Similar tests can be conducted using triples or quadruples of numbers instead of pairs, but the number of categories quickly becomes unmanageable.

Table 4.6 gives the value of χ^2/df for the number of occurrences of each pair of words that appear in order. The number of words examined ranged from 300 to 3300 in increments of 300. Again, the hypothesis that the number of occurrences of word pairs is equally distributed, can be rejected at the 99% level in only a few entries of table 4.6. The hypothesis can be rejected in the Fibonacci column when n = 2400 and in the random column when n = 1200 and n = 2400.

4.2.3 Wald-Wolfowitz Total Number of Runs Test

The Wald-Wolfowitz total number of runs test, [28], examines the grouping of similar objects in the sequence. Let a run be defined as a consecutive group of characters from the sequence,

$$\ldots, s_{k-1}, s_k, s_{k+1}, \ldots, s_{k+n-2}, s_{k+n-1}, s_{k+n}, \ldots$$

such that $s_{k-1} \neq s_k$, $s_k = s_{k+1} = \cdots = s_{k+n-2} = s_{k+n-1}$, and $s_{k+n-1} \neq s_{k+n}$. For example, the bits 001111001 have four runs, two runs of zeros of length two each and two runs of ones of lengths four and one. A random sequence of characters will likely not have each character grouped together, nor is it likely to have the character perfectly shuffled, the Wald-Wolfowitz will measure this grouping.

Table 4.7 gives the value of z for the Wald-Wolfowitz test with the number of bits examined equal to n. The values for z indicate that the number of runs is different than what we should expect if they were random. At the 99% level the values should be between -2.326 and 2.326. The values of z for the Fibonacci sequence of bits is similar to the values of z for the random sequence and different from the non-random sequence, a sequence of all ones.

4.3 Cycles in the b Bit of f_n

Let the sequence $\mathcal{B}_i$ be the sequence formed by concatenating the ith bit of f_n for n from 0 to ∞. The bits to the left of the left most one bit of f_n are assumed to be zeros. If the ith bit of f_n is examined independently of the the other bits a cycle is apparent. For $i = 0$ the sequence $\mathcal{B}_0$ is

$$011011011011011...$$

Similar sequences appear for other i as well.

Theorem 4.1 *The cycle length of the sequence $\mathcal{B}_n$ is $3 \cdot 2^n$.*

Proof. Since $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{3\cdot 2^n} \cdot \begin{pmatrix} f_1 = 1 \\ f_0 = 0 \end{pmatrix} = \begin{pmatrix} f_{3\cdot 2^n+1} \\ f_{3\cdot 2^n} \end{pmatrix}$ if $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{3\cdot 2^n} \bmod 2^{n+1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$ then $f_{3\cdot 2^n+1} \bmod 2^{n+1} = f_1$ and $f_{3\cdot 2^n} \bmod 2^{n+1} = f_0$.

It will be shown that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{3\cdot 2^n} \bmod 2^{n+1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I.$$

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^3 = \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}$$
$$= 2\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$
$$= 2A + I$$

so

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{3\cdot 2^n} = (2A+I)^{2^n}$$
$$= (2A)^0 \begin{pmatrix} 2^n \\ 0 \end{pmatrix} + (2A)^1 \begin{pmatrix} 2^n \\ 1 \end{pmatrix} + \cdots +$$
$$(2A)^{2^n-1} \begin{pmatrix} 2^n \\ 2^n-1 \end{pmatrix} + (2A)^{2^n} \begin{pmatrix} 2^n \\ 2^n \end{pmatrix}$$
$$= I + (2A)2^n + \cdots + (2A)^{2^n-1}2^n + (2A)^{2^n}$$

and this mod 2^{n+1} is I. ∎

These cycles also appear in the generalized Fibonacci numbers. Let $\mathcal{B}_i^k$ be the sequence formed by concatenating the ith bit of f_n^k for n from 0 to ∞.

Theorem 4.2 *The cycle length of the sequence $\mathcal{B}_n^k$ is $(k+1)\cdot 2^n$.*

Proof. Let $L = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & & & & \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}$.

It will be shown that

$$L^{(K+1)\cdot 2^n} \bmod 2^{n+1} = I.$$

since

$$f_n^k = 2f_{n-1}^k - f_{n-k-1}^k$$

$$
\begin{aligned}
L^n &= 2L^{n-1} - L^{n-k-1} \\
\frac{L^n}{L^{n-k-1}} &= 2\frac{L^{n-1}}{L^{n-k-1}} - \frac{L^{n-k-1}}{L^{n-k-1}} \\
L^{k+1} &= 2L^k - I \\
(L^{k+1})^{2^n} &= (2L^k - I)^{2^n} \\
&= (2L^k)^0 \begin{pmatrix} 2^n \\ 0 \end{pmatrix} + (2L^k)^1 \begin{pmatrix} 2^n \\ 1 \end{pmatrix} + \cdots + \\
& \quad (2L^k)^{2^n-1} \begin{pmatrix} 2^n \\ 2^n - 1 \end{pmatrix} + (2L^k)^{2^n} \begin{pmatrix} 2^n \\ 2^n \end{pmatrix} \\
&= I + (2L^k)2^n + \cdots + (2L^k)^{2^n-1}2^n + (2L^k)^{2^n}
\end{aligned}
$$

and this mod 2^{n+1} is I. Therefore

$$
\begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & \cdots & 0 \\
\vdots & & & & \\
0 & 0 & \cdots & 1 & 0
\end{pmatrix}^{(K+1)\cdot 2^n} \mod 2^{n+1} = I.
$$

∎

If the entire cycle must be computed to find the nth element the cycle this method will be useful only in computing the low order bits since the cycle lengths grow exponentially with the bit position. If there were a way to directly compute the n_{th} bit of the cycle $\mathcal{B}_i$ we might be able to construct an algorithm to compute f_n using fewer bit operations than is currently known.

There are patterns in the $\mathcal{B}_i$ sequeneces, but they are not well understood yet. The observations of these patterns is presented in the limited detail that they are known. The patterns that are discussed break the cycle into six bit groups that place one on top of the other with the least significant bit in the upper left corner of the pile. Table 4.8 shows the bit patterns for $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4$, and $\mathcal{B}_5$ where j is the jth group of six bits in the cycle.

j	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_3$	$\mathcal{B}_4$	$\mathcal{B}_5$
0	000110	000001	000000	000000	000000
1		011010			000110
2					011101
3				001011	001011
4				101011	101011
5					100110
6					111101
7			110001	110001	110001
8			011011	011011	011011
9					011101
10					000110
11				010000	010000
12				110000	110000
13					111101
14					100110
15			101010	101010	101010

Table 4.8: Repeated patterns in $\mathcal{B}_i$

Let $\mathcal{B}_i(j)$ be the jth group of 6 bits of the sequence $\mathcal{B}$. From table 4.8 it can be seen that

$$\begin{aligned}
\mathcal{B}_3(0) &= \mathcal{B}_4(0) = \mathcal{B}_5(0) = 000000 \\
\mathcal{B}_3(1) &= \mathcal{B}_4(3) = \mathcal{B}_5(7) = 110001 \\
\mathcal{B}_3(2) &= \mathcal{B}_4(4) = \mathcal{B}_5(8) = 011011 \\
\mathcal{B}_3(3) &= \mathcal{B}_4(7) = \mathcal{B}_5(15) = 101010
\end{aligned}$$

and also

$$\begin{aligned}
\mathcal{B}_4(1) &= \mathcal{B}_5(3) = 001011 \\
\mathcal{B}_4(2) &= \mathcal{B}_5(4) = 101011 \\
\mathcal{B}_4(5) &= \mathcal{B}_5(11) = 010000 \\
\mathcal{B}_4(6) &= \mathcal{B}_5(12) = 110000
\end{aligned}$$

If this pattern continued then

$$\begin{array}{ccc}
\mathcal{B}_6(0) = \mathcal{B}_5(0) & \mathcal{B}_6(3) = \mathcal{B}_5(1) & \mathcal{B}_6(4) = \mathcal{B}_5(2) \\
\mathcal{B}_6(7) = \mathcal{B}_5(3) & \mathcal{B}_6(8) = \mathcal{B}_5(4) & \mathcal{B}_6(11) = \mathcal{B}_5(5) \\
\mathcal{B}_6(12) = \mathcal{B}_5(6) & \mathcal{B}_6(15) = \mathcal{B}_5(7) & \mathcal{B}_6(13) = \mathcal{B}_5(8) \\
 & \vdots &
\end{array}$$

but actually

$$\begin{array}{ccc}
\mathcal{B}_6(0) = \mathcal{B}_5(0) & \mathcal{B}_6(4) = \mathcal{B}_5(1) & \mathcal{B}_6(3) = \mathcal{B}_5(2) \\
\mathcal{B}_6(8) = \mathcal{B}_5(3) & \mathcal{B}_6(7) = \mathcal{B}_5(4) & \mathcal{B}_6(12) = \mathcal{B}_5(5) \\
\mathcal{B}_6(11) = \mathcal{B}_5(6) & \mathcal{B}_6(16) = \mathcal{B}_5(7) & \mathcal{B}_6(15) = \mathcal{B}_5(8) \\
 & \vdots &
\end{array}$$

The six bit segments $\mathcal{B}_6(4 \cdot i - 1)$ and $\mathcal{B}_6(4 \cdot i)$ are switched in positions relative to what was predicted from earlier observations for $1 \leq i \leq 7$. When $\mathcal{B}_7$ was examined it also exhibited the same patterns, including the switch, along with one higher level

switch. Presumably, $\mathcal{B}_8$ shows that same patterns as $\mathcal{B}_7$ with one further switch and so on.

If this pattern of switching could be understood we would have a method of computing the nth bit of $\mathcal{B}_i$ for each bit i in f_n. This appears promising since the cycle $\mathcal{B}_i$ can be entered, at most, six bits from the bit we want, and starting with the low order bits we can use addition to compute those missing six bit positions. This method could be applied iteratively to compute each bit in f_n.

The incentive to investigate this method is that it appears that it may be possible to compute the ith bit of f_n in constant time. If this were the case we could compute f_n in linear time. This is still very speculative and unsuccessful to date.

Chapter 5

Conclusions and Future Research

This thesis has presented several algorithms for computing Fibonacci numbers. Table 5.1 lists the algorithms and the number of bit operations the algorithm uses to compute f_n.

A few of the algorithms developed led naturally to computing f_n for any positive n, but most algorithms more naturally compute f_n for $n = 2^i$. The fastest algorithm for computing f_n, $n = 2^i$, the product of factors algorithm, was used to develope a fast algorithm to compute f_n for any n.

The additive algorithms must add n numbers, each $\mathcal{N}n$ bits long, using about $\mathcal{N}n^2$ bit operations. The multiplicative algorithms usually multiply $\frac{\mathcal{N}n}{2}$ bit numbers to compute the nth Fibonacci number using about, with our assumption that multiplication uses n^2 bit operations, $(\mathcal{N}n)^2$ bit operations. Since $\mathcal{N} < 1$ the multiplicative algorithms will, when n is big enough, be faster than the additive algorithms.

Several versions of the Ziv Lempel compression algorithm and dynamic Huffman code compression algorithm failed to compress the sequence of bits generated by concatenating the binary representation of the Fibonacci numbers. The sequence of bits behaved similarly to the random bits generated using the linear congruential method in three tests of randomness.

Further research needs to be done on the patterns that appear in the bth bit of f_n. Understanding of these patters may lead to an improvement in the order of the number of bit operations used to compute f_n. Another method of computing the Fibonacci numbers that was not discussed here but may lead to a fast algorithm

Algorithm Name	number of bit-operations	f_n for any n or $n = 2^i$	section	new
Natural Recursive	$\Theta(\lambda_1^n)$	any n	2.1	no
Repeated addition	$\Theta\left(\mathcal{N}\left(\frac{n^2}{2} - \frac{n}{2}\right)\right)$	any n	2.2	no
Binet's Formula	$\Theta\left((\mathcal{N}n + \log n)^2\right)$	$n = 2^i$	2.3	no
Extending Vorob'ev	$\Theta\left(\frac{(\mathcal{N}n)^2}{2}\right)$	$n = 2^i$	2.4	yes
3 Multiply Matrix	$\Theta\left((\mathcal{N}n)^2\right)$	$n = 2^i$	2.6.1	no
2 Multiply Matrix	$\Theta\left(\frac{2(\mathcal{N}n)^2}{3}\right)$	$n = 2^i$	2.6.2	yes
Binomial Coefficients	$O(n^3)$	any n	2.8	no
Product of Factors	$\Theta\left(\frac{5(\mathcal{N}n)^2}{12}\right)$	$n = 2^i$	2.10	yes
Product of Factors	$\Theta\left(3(\mathcal{N}n)^2\right)$	any n	2.10	yes
Gries and Levin f_n^k	$\Theta\left(\frac{k^2(\mathcal{N}n)^2}{3}\right)$	$n = 2^i$	2.5	no
Generalized addition f_n^k	$\Theta\left(\mathcal{N}(n^2 - n)\right)$	any n	2.2	yes

Table 5.1: Summary of bit operation complexity using n^2 multiply

is computing the number of nodes in a complete AVL tree. It might be possible to recursively remove the top part of the AVL tree, the part that is a binary tree, leaving several smaller AVL trees.

Finally, the applicability of these methods to computing f_n^k and eventually to more general recurrence relations should be investigated.

Bibliography

[1] Aho, A. V., J. E. Hopcroft and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading MA; 1974.

[2] Anderson, P. G., "Fibonaccene," in *Fibonacci Numbers and Their Applications,* A. N. Philippou et al. (eds.), D. Reidel Publishing Company, Boston; 1986.

[3] Avriel, M. and D. J. Wilde, "Optimality proof for the symetric fibonacci search technique," in *Fibonacci Quarterly,***4**, 265-269; 1966.

[4] Beyer, W. H., Ed., "CRC Standard Mathematical Tables", 27th edition, CRC Press Inc, Boca Ratan FL; 1984.

[5] Bird, R. S., "Tabulation techniques for recursive programs," in *Computing Surveys,***12**, 403-413; 1980.

[6] Brousseau, A., "Relation of zeros to periods in the Fibonacci sequence modulo a prime," in *American Math Monthly,***71**, 897-899; 1964.

[7] Brousseau, A., "Fibonacci and Related Number Theoretic tables," The Fibonacci Association, San Jose; 1972.

[8] Capocelli, R. M., and P. Cull, "Generalized Fibonacci numbers are rounded powers," Technical report no. 87-50-1, Oregon State University, Department of Computer Science; 1987.

[9] Dickson, L. E., "History of the Theory of Numbers," Vol. 1, Carnegie Institution of Washington, Washington D. C., 393-411; 1919.

[10] Er, M. C., "A fast algorithm for computing order-k Fibonacci numbers," in *The Computer Journal,***26**, 224-227; 1983.

[11] Er, M. C., "Computing sums of order-k Fibonacci numbers in log time," in *Information Processing Letters,***17**, 1-5; 1983.

[12] Fiduccia, C. M., "An efficient formula for linear recurrences," in *SIAM Journal of Computing,***14**, 106-112; 1985.

[13] Gerald, C. F., and P. O. Wheatley, "Applied Numberical Analysis," Addison Wesley, Reading MA; 1984.

[14] Goetgheluck, P., "Computing binomial coefficients," in *American Math Monthly,* **94**, 360-365; 1987.

[15] Gries, D. and G. Levin, "Computing Fibonacci numbers (and similarly defined functions) in log time," in *Information Processing Letters,* **11**, 68-69; 1980.

[16] Horowitz, E. and S. Sahni, "Fundamentals of Data Structures," Computer science press, Rockville MD; 1982.

[17] Jean, R. V., "Mathematical Approach to Pattern and Form in Plant Growth," John Wiley and Sons, New York NY; 1984.

[18] Kiefer, J., "Sequential minimax search for a maximum," in *Procedings of the American Math Society,* **4**, 502-506; 1953.

[19] Knuth, D. E., "The Art of Computer Programming," Vol. 1, Addison-Wesley, Reading MA; 1973.

[20] Knuth, D. E., "The Art of Computer Programming," Vol. 2, Addison-Wesley, Reading MA; 1981.

[21] Lahr, J., "Fibonacci and Lucas numbers and the Morgan-Voyce polynomials in ladder networks and in electric line theory," in *Fibonacci Numbers and Their Applications,* A. N. Philippou et al. (eds.), D. Reidel Publishing Company, Boston; 1986.

[22] Lewis, T. G. and M. Z. Smith, "Applying Data Structures," Houghton Mifflin, Bostion; 1982.

[23] Lynch, W. C., The t-Fibonacci numbers and polyphase sorting," in *Fibonacci Quarterly* **8**, 6-22; 1970.

[24] Oliver, L. T., and D. J. Wilde, "Symmetric sequential minimax search for a maximum," in *Fibonacci Quarterly,* **2**, 169-175; 1964.

[25] Pettorossi, A., "Derivation of an $O(k^2 \log n)$ algorithm for computing order-k Fibonacci numbers from the $O(k^3 \log n)$ matrix multiplication method," in *Information Processing Letters,* **11**, 172-179; 1980.

[26] Pettorossi, A. and R. M. Burstall, "Deriving very efficient algorithms for evaluatinb linear recurrence relations using the program transformation technique," in *Acta Infomatica,* **18**, 181-206; 1982.

[27] Reingold, E. M. and W. J. Hansen, "Data Structures," Little, Brown and Company; 1983.

[28] Romano, A., "Applied statistics for science and industry", Allyn and Bacon, Boston; 1977.

[29] Rosen, K. H., "Elementary Number Theory and Its Applications," pp 60-61, Addison Wesley, Reading MA; 1984.

[30] Rotkieicz, A., "Problems on the Fibonacci numbers and their generalizations," in *Fibonacci Numbers and Their Applications,* A. N. Philippou et al. (eds.), D. Reidel Publishing Company; 1986.

[31] St. John, P. H., "On the asymptotic proportions of zeros and ones in Fibonacci sequences," in *Fibonacci Quarterly,* **22**, 144-145; 1984.

[32] Shortt J., "An iterative program to calculate Fibonacci Numbers in $O(logn)$ arithmetic operations," in *Information Processing Letters,***7**, 299-303; 1977.

[33] Urbanek, F. J., "An $O(\log n)$ algorithm for computing the nth element of the solution of a difference equation," in *Information Processing Letters,***11**, 66-67; 1980.

[34] Vinson J., "The relation of a period modulo to the rank of apparition of m in the Fibonacci sequence," in *Fibonacci Quarterly,***2**, 37-45; 1963.

[35] Vitter, J. S. "Design and analysis of dynamic Huffman codes," in *Journal of the Association for Computing Machinery,* **34**, 825-845; 1987.

[36] Vorob'ev, N.N., "Fibonacci Numbers," Pergamon press, New York; 1961.

[37] Wall, D. D., "Fibonacci series modulo m," in *American Math Monthly,***67**, 525-532; 1960.

[38] Welch, T. A., "A technique for high-perfomance data compression," in *Computer,***17**, 8-19; 1984.

[39] Witten, I. H., R. M. Neal and J. G. Cleary, "Arithmetic coding for data compression," in *Communications of the ACM,***30**, 520-540; 1987.

[40] Ziv, J. and A. Lempel, "On the complexity of finite sequences," in *IEEE Transactions on Information Theory,***IT-22**, 75-81; 1976.

[41] Ziv, J. and A. Lempel, "A universal algorithm for sequential data compression," in *IEEE Transactions on Information Theory,***IT-23**, 337-343; 1977.

[42] Ziv, J. and A. Lempel, "Compression of individual sequences via variable-rate coding," in *IEEE Transactions on Information Theory,* **IT-24**, 530-536; 1978.

Appendices

Appendix A

List of Fibonacci Relations

$f_{n+2} = f_{n+1} + f_n,\ n \geq 0,\ f_0 = 0,\ f_1 = 1$	1.1
$f_n^k = 2f_{n-1}^k - f_{n-k-1}^k$	2.3
$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}\right)$	2.5
$f_n = \left\lfloor \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n + 0.5 \right\rfloor,\ n \geq 0$	2.6
$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$	
$f_{2n+1} = f_n^2 + f_{n+1}^2$	2.7
$f_{n+1}(f_{n+1} + 2f_n) = f_{2n+2}$	2.8
$f_n(2f_{n+1} - f_n) = f_{2n}$	2.9
$f_{n+k} = f_{n-1}f_k + f_nf_{k+1}$	2.10
$f_{2n} = f_{n+1}^2 - f_{n-1}^2$	2.12
$f_{2n+1} = f_{n-1}f_{n+1} + f_n(f_{n+1} + f_n)$	2.13
$G(z) = z/(1 - z - z^2)$	2.18
$\left(\frac{f_{2n}}{f_n}\right)^2 - 2(-1)^n = \frac{f_{4n}}{f_{2n}}$	2.20
$f_{2^i+k} = f_{2^{i-1}+k}\beta_i - f_k$	2.21
$\mathcal{N}n = \log_2 \lambda_1^n = \frac{\log_{\lambda_1} \lambda_1^n}{\log_{\lambda_1} 2} = \frac{n}{log_{\lambda_1} 2}$	3.1
$\lvert\mathcal{B}_i^k\rvert = (k+1) \cdot 2^n$	

Appendix B

Lisp Functions

B.1 Natural Recursion

```
(defun rec-fib (n)
   (cond ((= n 1) 1)
         ((= n 0) 0)
         (t (+ (rec-fib (- n 1)) (rec-fib (- n 2))))))
```

B.2 Repeated Addition

```
(defun add-fib (n)
   (cond ((eq n 1) 1)
         ((eq n 0) 0)
         (t
            (prog (prev1 prev2 next)
                  (setq prev1 0)
                  (setq prev2 1)
                  (return (do ((i n (- i 1))) ((= i 1) prev2)
                              (setq next (+ prev1 prev2))
                              (setq prev1 prev2)
                              (setq prev2 next))))))
```

B.3 Binet's Formula

```
(defun binet-fib-help (i)
   (* (- (expt (/ (+ 1 sqrt5) 2) i)
         (expt (/ (- 1 sqrt5) 2) i))
      inverted_sqrt5))

(defun binet-fib (i)
```

```
   (setq sqrt5 (my-sqrt 5 (/ 1 (ash 1 i)))) ; Get the i most significant
   (setq inverted_sqrt5 (/ 1 sqrt5)) ; bits of \( sqrt{5} \)
   (binet-fib-help i))

(defun fib (n)
  (cond ((eql 4 n) 3)
        (t (ceiling (* (square (fib (/ n 2)))  (sqrt 5))))))
```

B.4 N. N. Vorob'ev's Methods

```
(defun nnv2-fib (i)
   (cond ((= 0 i) 0)
         ((= 2 i) 1)
         ((oddp i) (add-fib i))
         (t (- (square (nnv2-fib (+ (/ i 2) 1)))
               (square (nnv2-fib (- (/ i 2) 1)))))))

(defun nnv3-fib (i)
   (cond ((= 0 i) 0)
         ((not (= 0 (mod i 3))) (add-fib i))
         (t (- (+ (expt (nnv3-fib (+ (/ i 3) 1)) 3)
                  (expt (nnv3-fib (/ i 3)) 3))
               (expt (nnv3-fib (- (/ i 3) 1)) 3)))))

(defun nnv4-fib (i)
  (cond ((eql i 1) '(1 (0 1)))
        (t (let* ((fiba (nnv4-fib (/ i 2)))
                  (fibb (let* ((lst (cadr fiba))
                               (n (car lst))
                               (np1 (cadr lst))
                               (nm1 (- np1 n)))
                          (+ (* nm1 np1)
                             (* n
                                (+ np1 n))))))
             (list (* (car fiba) (+ fibb (- fibb (car fiba))))
                   (list (car fiba) fibb))))))
```

B.5 Matrix Methods

B.5.1 Three Multiply Matrix Method

```
(defun mat-fib (i)
   (cond ((eq 1 i) '((1 1) (1 0)))
         (t (mat-square (mat-fib (/ i 2))))))

(defun mat-square (mat)
   (prog (n+1 n n-1)
      (setq n+1 (square (car (car mat))))
      (setq n (square (car (car (cdr mat)))))
      (setq n-1 (square (car (cdr (car (cdr mat))))))
      (return (list (list (+ n+1 n) (setq temp (- n+1 n-1)))
                    (list temp (+ n n-1))))))
```

B.5.2 Two Multiply Matrix Method

```
(defun mat2-fib (i)
   (cond ((eq 1 i) '((1 1) (1 0)))
         (t (mat2-square (mat2-fib (ash i -1))))))

(defun mat2-square (mat)
   (prog (alpha beta
          (old-alpha (caar mat))
          (old-beta (cadar mat)))
         (setq alpha (* old-alpha
                        (+ old-alpha
                           (ash old-beta 1))))
         (setq beta (* old-beta
                       (- (ash old-alpha 1)
                          old-beta)))
         (return (list (list (setq temp (- alpha beta))
                             beta)
                       (list beta
                             (- temp beta))))))
```

B.6 Generating Function

```
(defun beta (i)
   (cond ((eq 1 i) '(3))
```

```
        (t (prog (temp)
                 (setq temp (beta (- i 1)))
                 (return (push (- (* (car temp) (car temp)) 2) temp))))))

(defun gf2-fib (i)
   (cond ((= 0 i) 1)
         ((= 1 i) 1)
         (t (apply '* (beta (- i 1))))))

(defun gf4-fib (n) (gf4-fib-help (round (log n 2))))

(defun gf4-fib-help (n m o)
   (let ((b 3)
         (limit (- n 1)))
     (do ((i 1 (+ i 1))) ((>= i limit) m)
       (setf b (- (* b b) 2))
       (setf m (- (* m b) o)))))

(defun anyf (n)
   (let ((a0 3) (s0 0)
         (a1 2) (s1 1)
         (b 3)
         (alpha0-list '(3 1)) (alpha1-list '(2 1))
         (limit (- (integer-length n) 1)))

     (do ((i 2 (+ i 1))) ((>= i limit) t)
       (setf b (- (* b b) 2))
       (push (setf a0 (- (* a0 b) s0)) alpha0-list)
       (push (setf a1 (- (* a1 b) s1)) alpha1-list))

     (anyf-help n
                limit
                (reverse alpha0-list)
                (reverse alpha1-list))))

(defun anyf-help (n q alpha0-list alpha1-list)
  (cond ((eql n 0) 0)
        ((eql n 1) 1)
        ((eql n 2) 1)
        (t (+ (* (nth (- q 1) alpha1-list)
                 (anyf-help (setf n (- n (ash 1 q)))
                            (- (integer-length n) 1)
```

```
                       alpha0-list
                       alpha1-list))
          (* (nth (- q 1) alpha0-list)
             (anyf-help (incf n)
                       (- (integer-length n) 1)
                       alpha0-list
                       alpha1-list))))))
```

B.7 Binomial Coefficients

```
(defun std2-binomial (n k)
   (prog ((binom 1))
      (return (do ((i n (- i 1)) (j k (- j 1))) ((= j 0) binom)
                (setq binom (* (/ i j) binom))))))

(defun mult2-binomial (n k)
   (cond ((= k 0) 1)
         ((= k 1) n)
         (t (/ (apply '* (gen-values n (- n k)))
            (apply '* (gen-values k 1))))))
```

B.8 Generalized Fibonacci Numbers

```
(defun kfib (n k)
  (prog ((mem (append '(1 1) (make-list (- k 1) :initial-element '0)))
         (limit (- n k)))
    (return (do ((i 1 (+ i 1))) ((> i limit) (car mem))
              (setq mem (nbutlast (nconc (list (- (ash (car mem) 1)
                                                   (car (last mem))))
                                       mem)))))))

(defun gries-expo (mat n)
  (do ((i 0 (+ i 1))) ((>= i n) mat)
    (gries-square mat)))

(defun gries-square (mat)
  (let ((new-bottom (vector-mat-mult (car mat) (cdr mat))))
    (rplca mat new-bottom)
    (mapcar #'(lambda (elem lst)
```

```
    (nsubstitute-if elem #'t lst :count 1 :from-end t))
  new-bottom
  (cdr mat))))
```