ALGORITHMS AND COMBINATORICS OF MAXIMAL COMPACT CODES

by

CHRISTOPHER JORDAN DEUGAU
B.Sc., Malaspina University-College, 2003

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

ALGORITHMS AND COMBINATORICS OF MAXIMAL COMPACT CODES

by

CHRISTOPHER JORDAN DEUGAU
B.Sc., Malaspina-University College, 2003

Examiners:

Dr. F. Ruskey, (Department of Computer Science)

Supervisor

Dr. D. Roelants Van Baronaigien, (Department of Computer Science)

Supervisor

Dr. G. MacGillivray, (Department of Mathematics & Statistics)

Member

Dr. B. Jackson, (Department of Mathematics, San Jose State University)

External Examiner

Examiners:

Dr. F. Ruskey, (Department of Computer Science)

Supervisor

Dr. D. Roelants Van Baronaigien, (Department of Computer Science)

Supervisor

Dr. G. MacGillivray, (Department of Mathematics & Statistics)

Member

Dr. B. Jackson, (Department of Mathematics, San Jose State University)

External Examiner

ABSTRACT

The implementation of two different algorithms for generating compact codes of some size N are presented. An analysis of both algorithms is given, in an attempt to prove whether or not the algorithms run in constant amortized time. Meta-Fibonacci sequences are also investigated in this paper. Using a particular numbering on k-ary trees, we find that a group of meta-Fibonacci sequences count the number of nodes at the bottom level of these k-ary trees. These meta-Fibonacci sequences are also related to compact codes. Finally, generating functions are proved for the meta-Fibonacci sequences discussed.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I want to thank Dr. Frank Ruskey for his support, both financially through my funding, and also through our weekly meetings, and also for the guidance necessary to complete this thesis. I also thank him for pointing me towards the topics I discuss in my thesis. I also would like to thank Dr. Dominique Roelants for his support, and helping point me towards the University of Victoria. Finally, I would like to thank Dr. Gary MacGillivray for helping make me feel welcome at the University of Victoria during my entire time on campus.

Dedication

In loving memory of my grandfather, William McFarlane. I wish you were here for this.

Chapter 1 — Introduction and Definitions

This thesis deals with two different problems about compact codes. The first problem is related to the generation of compact codes and compact code trees. Research has previously been done on an algorithm for generating all possible compact codes [8], as well as simply trying to count the number of compact codes [11]. For this thesis, I implemented two similar algorithms and tried to determine whether the algorithms run in constant amortized time.

The second part of this thesis deals with meta-Fibonacci sequences. Meta-Fibonacci sequences have been looked at in many different papers; they were first mentioned in [5], and were also mentioned in [2] and [7]. While research has been done on these sequences, there has not been a lot proven about these sequences. Jackson and Ruskey [7] were able to prove that for certain initial conditions, a combinatorial meaning involving binary trees could be given to a particular set of these sequences. In this thesis, I prove that using a similar set of initial conditions, the same combinatorial meaning can be extended to k-ary trees. These sequences also prove to be useful in one of the algorithms I mentioned above.

In Chapter 2, I will discuss the algorithms for generating all possible compact codes. We will look at the previous work, as well as give a brief discussion on my implementations of the algorithms, and also the work done to try and prove whether or not the algorithms run in constant amortized time. In Chapter 3, I will review previous work done on the meta-Fibonacci sequence, including the paper by Jackson and Ruskey [7]. After that, I will provide proofs to prove a recurrence relation similar to Jackson and Ruskey's recurrence for binary trees that will extend the combinatorial meaning to k-ary trees.

1.1 Definitions

According to Donald Knuth [9], a *k-ary tree* is a finite set T of *nodes* that is either empty, or it consists of some positive number of nodes where one node is called the *root node* and the remainin nodes are partitioned into a sequence of k disjoint k-ary trees, $T_1, T_2, \ldots, T_k$, which are called the *subtrees* of the root. Take some node x in a k-ary tree. A node y is a *child* of x if y is the root node of one of the subtrees of x. If y is a child of x, then x is called the *parent* of y. The *degree* of a node is the number of children that particular node has. If a node has a degree of 0, the node is referred to as a *leaf* node. If a node has a degree $m \geq 1$, the node is called an *inner* or *internal* node. A *binary tree* is a k-ary tree where $k = 2$. An *extended binary tree* is a binary tree where every internal node has exactly 2 children.

Knuth [9] defines the *level* of a node in some tree T recursively; the level of the root is 0, and the level of a child is one higher than the level of its parent. *Traversing* a tree is a method of visiting every single node in a tree, one by one. A *preorder traversal* of some k-ary tree T is a traversal where you first visit the root node of T, and then do a preorder traversal on each of the subtrees $T_1, T_2, \ldots, T_k$ in order.

Chapter 2 — Compact Codes

Assume that we have N different symbols which we want to encode. For example, say we wanted to encode the four symbols in Table 2.1. A *binary code* is a way to encode the symbols using just the binary numbers 0 and 1, so our code is a binary code. A *symbol* is the smallest unit of data to be compressed [13]. The smallest units of data we are trying to compress are the characters a, b, c, and d. A *code word* represents one symbol we are trying to encode. So, the code word for c is 110. Let $(l_1, l_2, \ldots, l_N)$ represent some binary code with N code words, where l_i is the length of the code word for the symbol i in our language. A *compact code* [8] is a sequence $(l_1, l_2, \ldots, l_N) \in \mathbb{N}^N$ that satisfies the expressions

$$1 \leq l_1 \leq l_2 \leq \cdots \leq l_{N-1} = l_N, \tag{2.1}$$

and

$$\sum_{j=1}^{N} 2^{-l_j} = 1. \tag{2.2}$$

For example, $(2, 2, 3, 4, 4, 4, 4, 4, 4)$ represents a compact code of *size* 9 because

$$\frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{16} + \frac{1}{16} + \frac{1}{16} + \frac{1}{16} + \frac{1}{16} = 1,$$

and there are 9 numbers in the sequence. Therefore, our compact code would consist of 9 code words; two of length two, one of length three, and six of length four. The

Symbol	Code Word
a	0
b	10
c	110
d	111

Table 2.1. *Symbols and code words for a code with lengths* $(1, 2, 3, 3)$.

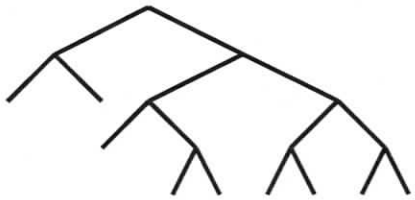

Figure 2.1. *Compact code tree for* $(2, 2, 3, 4, 4, 4, 4, 4, 4)$.

existance of these codes can be seen by looking at the later description of compact code trees. Table 2.2 on page 8 provides a listing of all of the compact codes of size nine.

Compact codes can also be represented as a binary tree [4]. In the binary tree representation, we create a tree T which has N leaf nodes. Assume that we number each leaf node $n_1, n_2, \ldots, n_N$ from left to right, top to bottom. We create our tree by placing the leaf node n_i at level l_i of our tree. This is always possible based on our bounds, and the fact that the lengths add up to one. Figure 2.1 shows the compact code tree representation of the compact code of size nine given above. You can see that the first two leaf nodes are at level two, the next leaf node is at level three, and the final six leaf nodes are at level four of the tree. Note that binary compact code trees are always *extended binary trees*, meaning that every internal node has exactly two children. Using these compact code trees, one can also generate code words for the given lengths. By tracing the path from the root node to each leaf node, and representing the left branch of a node as 0 and the right branch of a node as 1, you can create code words. For example, in Figure 2.1, the left most leaf node would be represented by the code 00, while the node directly to its right would be represented by the code 01.

2.1 Previous Work

Compact codes are useful because they provide a binary code which is uniquely decodable [11]. If each symbol has a probability p_i, Abramson [1] shows that we can use compact codes to create a code with the minimum possible average word length

L, where

$$L = \sum_{i=1}^{N} p_i l_i. \tag{2.3}$$

Related to the problem of finding the right compact code for some particular probability distribution is the idea of finding all possible compact codes of a certain size. While algorithms exist for finding the optimal codes (based on (2.3)), they rely on the fact that we can determine the probabilities for each symbol. For example, Huffman's algorithm [6] can create a minimal code if we have a probability for each symbol before we start the algorithm. Without these probabilities, we are unable to use Huffman's algorithm. One possible use of an algorithm for finding all possible compact codes would be the ability to try and minimize some function f; so, while we may not have particular values that we can use for the probabilities of symbols, it is possible that we could try and determine which tree works best based on several different constraints.

I worked on implementations of two different algorithms for finding all possible compact codes. The first algorithm comes from a paper by Khosravifard, Esmaeili, Saidi, and Gulliver [8]. This paper proves new constraints on the l_i values which can be used to generate all of the possible compact codes of size N. The algorithm also proves difficult to prove bounds on due to the nature of the code generated. The second algorithm is based on a paper by Emily Norwood [11]. Norwood proves a recurrence relation that counts the number of different possible compact codes of size N, but does not directly generate the actual codes. Using the same recurrence relation, we can also generate all of the compact code trees.

2.1.1 A Tree Based Algorithm

In [8], the authors prove that we can replace the previous constraints from (2.1) and (2.2) on the lengths $(l_1, l_2, \ldots, l_N)$ with the following constraints

$$\max\left\{l_{i-1}, \left\lfloor \lg \frac{2}{1 - \sum_{j=1}^{i-1} 2^{-l_j}} \right\rfloor\right\} \leq l_i \leq \left\lfloor \lg \frac{N-i+1}{1 - \sum_{j=1}^{i-1} 2^{-l_j}} \right\rfloor \tag{2.4}$$

for $1 \leq i \leq N-1$, and also show that $l_N = l_{N-1}$. The same paper also provides an algorithm for generating a tree structure which has a leaf node for every possible

compact code. The tree generated in [8] can be viewed as a computational tree for the version of the algorithm that I implement in Section 2.2.1.2.

2.1.2 Number of Different Compact Codes

Norwood's paper [11] was written approximately thirty-six years prior to [8]. While [8] looked at generating the compact codes for some size N, Norwood only looked at finding the number of compact codes for size N. Norwood's paper also differed in the fact that she grouped the compact codes based on the number of nodes at the bottom level of the tree, as opposed to simply the number of leaf nodes.

Let $\mathbf{T}(p, q)$ be the set of all the compact code trees that are valid by (2.1) and (2.2), and that have q leaf nodes with p of those leaf nodes at the bottom level of the tree. Also, let $T(p, q) = |\mathbf{T}(p, q)|$. Norwood proves that

$$|\mathbf{T}(2s, q+s)| = |\bigcup_{p \geq s} \mathbf{T}(p, q)|, \tag{2.5}$$

and it follows that

$$T(2s, q+s) = \sum_{p \geq s} T(p, q), \tag{2.6}$$

which is obvious given the definition of cardinality, and the fact that all of the sets are disjoint.

Equation (2.5) is easier understood if you substitute q for $q+s$, and get

$$|\mathbf{T}(2s, q)| = |\bigcup_{p=s}^{q-s} \mathbf{T}(p, q-s)|. \tag{2.7}$$

The following is a brief summary of the proof of (2.7). Take some tree that has q leaf nodes, with $2s$ of those leaf nodes at the bottom level of the tree. If you delete the $2s$ nodes at the bottom level, you will be taking $2s$ leaf nodes away; but the s parent nodes of those leaf nodes will have no children, and by definition, they will become leaf nodes. That means that we now have $q - 2s + s = q - s$ leaf nodes. This raises the question of how many of the leaf nodes are at the bottom level of the tree? We don't know exactly how many leaf nodes p will be at the bottom level of the tree, but we do know that there must be at least s nodes at the bottom level; the s nodes

that were the parents of the nodes at the bottom level before our deletion step must now be at the bottom level of our new tree. We also know that at most $q - s$ of the leaf nodes are at the bottom level, since there are only $q - s$ leaf nodes remaining after the deletion step. It is now clear that there is a one-to-one correspondence between the trees in $\mathbf{T}(s, q - s), \mathbf{T}(s + 1, q - s), \ldots, \mathbf{T}(q - s, q - s)$ and the trees in $\mathbf{T}(2s, q)$ through this deletion step.

Norwood gives a table that shows the values of $T(p, q)$ for $p = 2, 4, 6, \ldots, 20$ and $q = 2, 3, 4, \ldots, 23$. The p value must always be even; we are dealing with compact codes, and due to the fact that all compact code trees are extended binary trees, we must have an even number of nodes at the bottom level. Norwood also points out that you can determine the values of the entire table from just the fact that $T(2, 2) = 1$. A condensed version of this table is provided in Table 2.3.

The last thing Norwood discusses is where rows begin in the table. For example, the row for $p = 2$ has its first value when $q = 2$, and $p = 4$ has its first value when $q = 4$, but the first value in the row representing $p = 6$ occurs when $q = 7$. Norwood gave several explanations as to how one could determine which column q a particular row p started; the simplest way was to use the fact that all the branchings are binary, so the total number of nodes at each level must be even (except the top level, which will have simply the root node). So, when $p = 6$, the bottom level will have 6 nodes. The previous level must have at least 4 nodes instead of $6/2 = 3$, and the extra node will be a leaf node (since it has no children). The level above that must have at least 2 nodes, since $4/2 = 2$ and 2 is even. The top level would then have the root node. Therefore, at a minimum, a tree with 6 nodes at the bottom level must have at least 7 leaf nodes. Thus, row $p = 6$ starts at column $q = 7$. We will look at this sequence in more detail in Section 3.1.1.

2.2 Implementation and Analysis

In this section I will discuss my implementation of the algorithms from the papers by Khosravifard, Esmaeili, Saidi, and Gulliver [8] (which can be found on the COS website) and Norwood [11].

(1,2,3,4,5,6,7,8,8)	(1,2,3,4,5,7,7,7,7)	(1,2,3,4,6,6,6,7,7)	(1,2,3,5,5,5,6,7,7)
(1,2,3,5,5,6,6,6,6)	(1,2,4,4,4,5,6,7,7)	(1,2,4,4,4,6,6,6,6)	(1,2,4,4,5,5,5,6,6)
(1,2,4,5,5,5,5,5,5)	(1,3,3,3,4,5,6,7,7)	(1,3,3,3,4,6,6,6,6)	(1,3,3,3,5,5,5,6,6)
(1,3,3,4,4,4,5,6,6)	(1,3,3,4,4,5,5,5,5)	(1,3,4,4,4,4,4,5,5)	(1,4,4,4,4,4,4,4,4)
(2,2,2,3,4,5,6,7,7)	(2,2,2,3,4,6,6,6,6)	(2,2,2,3,5,5,5,6,6)	(2,2,2,4,4,4,5,6,6)
(2,2,2,4,4,5,5,5,5)	(2,2,3,3,3,4,5,6,6)	(2,2,3,3,3,5,5,5,5)	(2,2,3,3,4,4,4,5,5)
(2,2,3,4,4,4,4,4,4)	(2,3,3,3,3,3,4,5,5)	(2,3,3,3,3,4,4,4,4)	(3,3,3,3,3,3,3,4,4)

Table 2.2. *All possible compact codes of size 9.*

	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
2	1	1	1	2	3	5	9	16	28	50	89	159	285	510	914
4			1	1	2	3	5	9	16	28	50	89	159	285	510
6						1	1	2	4	7	12	22	39	70	126
8							1	1	2	4	7	12	22	39	70
10											1	2	3	6	11
12												1	2	3	6
14														1	1
16															1

Table 2.3. $T(p, q)$ *for* $p = 2, 4, \ldots, 16$ *and* $2 \leq q \leq 16$.

2.2.1 Khosravifard Algorithm

While working as a research assistant for Frank Ruskey, I was given the opportunity to work on the Combinatorial Object Server (COS)[1], a website which lets users automatically create a limited number of various combinatorial objects. One of the first programs I worked on for COS was a program for calculating all possible compact codes for some n. This program was written in C, and prints the compact codes in lex order. Pseudocode for this algorithm can be found in Figure 2.2 on page 10.

2.2.1.1 Explanation of Pseudocode for Khosravifard Algorithm

This algorithm makes use of a global varaiable `N`, which is an integer variable representing the number of code words in our code. This parameter `N` is given to the program by the user. The GENTREE method is recursive, and takes three arguments. The `l` argument is an integer variable, and represents the length $\mathtt{l}_{\mathtt{N}-\mathtt{n}}$. The `n` argument is also an integer variable, and represents how many lengths we have left to find. When $\mathtt{n} = 1$, we only have one length left to find, and this is our base case. The `S` argument is a floating point number, and it represents $\sum_{\mathtt{i}=1}^{\mathtt{N}-\mathtt{n}} 2^{-\mathtt{l}_\mathtt{i}}$.

The array `P` is a global array of integers indexed from 1 to `N`. We use it to store all of the lengths before we print them out. The local variables `L` and `U` are integers used to represent the upper and lower limits for some $\mathtt{l}_\mathtt{i}$, and the local variable `s` is a float which will represent the value of $2^{\mathtt{l}_\mathtt{i}}$ for updating the `S` value on recursive calls.

In (1), we check to see if $\mathtt{n} = 1$. If it does, then from the fact that $\mathtt{l}_{\mathtt{N}-1} = \mathtt{l}_\mathtt{N}$, we can set the value of `P[N]`. Now that our array is filled we can print it. and return to our previous method call.

After setting the upper and lower bounds for the for loop (2), we set `s` to be $2^{-\mathtt{L}}$. As we go through the foor loop and divide `s` by 2, we will be giving it the values $2^{-\mathtt{L}}, 2^{-(\mathtt{L}+1)}, 2^{-(\mathtt{L}+2)}, \ldots, 2^{-\mathtt{U}}$. Therefore, since we are going through all the possible values for $\mathtt{l}_{\mathtt{N}-\mathtt{n}+1}$, we will make all of the recursive calls to GENTREE with the parameters set correctly.

The main method's initial call will be to GENTREE(0, 0, N), so the first value will be put in $\mathtt{P}[\mathtt{N} - \mathtt{N} + 1] = P[1]$.

[1] http://www.theory.cs.uvic.ca/~cos/

Algorithm 2.2.1: GENTREE(l, S, n)

comment: Generate next length

if n = 1

then { P[N] ← P[N − 1] (1)
print P
return

L ← max(l, ⌊lg(2/(1 − S))⌋)

U ← ⌊lg(n/(1 − S))⌋

s ← 2^{-L}

for i ← L **to** U

do { P[N − n + 1] ← i (2)
GENTREE(i, S + s, n − 1)
s ← s/2

Figure 2.2. *Pseudocode for the* GENTREE(l, S, n) *method*

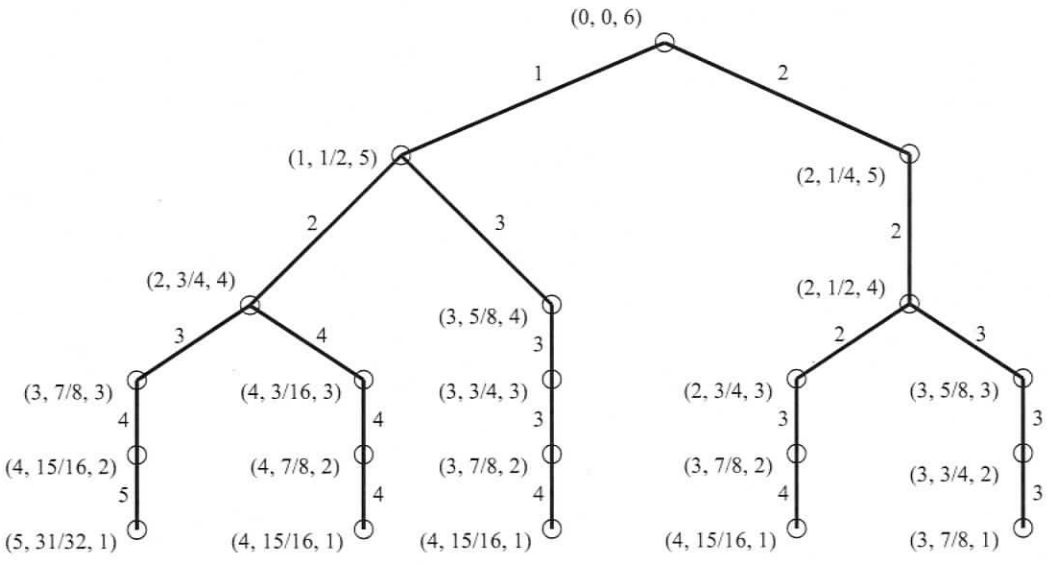

Figure 2.3. *Computation Tree for* $N = 6$

N	$\frac{\text{Calls}}{\text{Outputs}}$	N	$\frac{\text{Calls}}{\text{Outputs}}$	N	$\frac{\text{Calls}}{\text{Outputs}}$	N	$\frac{\text{Calls}}{\text{Outputs}}$
2	2.000	**10**	4.920	**18**	5.175	**26**	5.196
3	3.000	**11**	4.978	**19**	5.181	**27**	5.197
4	3.500	**12**	5.031	**20**	5.186	**28**	5.197
5	4.000	**13**	5.084	**21**	5.189	**29**	5.198
6	4.200	**14**	5.116	**22**	5.192	**30**	5.198
7	4.444	**15**	5.139	**23**	5.194	**31**	5.198
8	4.750	**16**	5.158	**24**	5.195	**32**	5.198
9	4.857	**17**	5.168	**25**	5.196	**33**	5.198

Table 2.4. *Number of calls divided by number of outputs for* $2 \leq N \leq 33$

2.2.1.2 Analysis of Khosravifard Algorithm

After successfully writing the program and testing it against various small test cases, as well as comparing the number of trees generated by the program for the number of trees expected from Norwood's paper [11], and the fact that there are 565,168 compact codes for $N = 26$ [8], we concluded the program was running correctly. At this point, we wanted to try and analyze the algorithm to see if it ran in *Constant Amortized Time* (CAT). An algorithm which runs in Constant Amortized Time is also known as a *CAT Algorithm*. An algorithm is CAT if the total amount of work done divided by the number of objects generated is constant; in other words, only a constant amount of work is done for each object on average. For the purposes of this program, the amount of work done is counted by the number of method calls made by the program.

For the purposes of this analysis, I am also assuming that the lg operation is running in the same amount of time as the other operations. In the actual implementation of the programs, the lg operation is actually run by making two calls to C's library method `double log10(double x)`. By this assumption, we can now see that the running time for each call to the GENTREE(0,0,N) method is based on the recursive calls.

The first check to determine if this algorithm is CAT was done by tracking the number of calls to the GENTREE method, which would be the amount of work done

for some N. This is true because, as you can see by the code provided in Figure 2.2, the work done in GENTREE is all done in a single for loop. By taking the number of calls and dividing by the number of objects generated, we are able to see if the ratio of work done to outputs is moving towards some sort of constant. On the machine used to serve COS, we were able to efficiently run this algorithm for $2 \leq N \leq 33$. The results are listed in Table 2.4.

As one can see by looking at Table 2.4, it seems that the algorithm is at least experimentally a CAT Algorithm, with a constant somewhere around 5.2. I did not go any further with the CAT analysis on this algorithm, however. Instead, I tried focusing on the second algorithm which seemed in the experiments to be more efficient, and it also appeared that it would be easier to prove that the second algorithm was a CAT algorithm.

2.2.2 Norwood's Algorithm

The second algorithm I implemented was based on Emily Norwood's formula (2.7) for counting the number of different possible compact codes [11]. The theory behind this algorithm is as follows. At each level of the recursion we are generating trees with p bottom level nodes, so if at each step we create these p nodes, when we recurse to the level above we can create the nodes at the penultimate level, and start to form our tree recursively. When we get to the top level of our recursive calls, which is $T(1,1)$, we will be generating the root node. We simply have to set the left and right subtrees to the root node and return the tree.

We will now go through a short example of how the algorithm works. I will be using Figure 2.4 on page 13 to show graphically the process of what is happening in each step. The white nodes will represent nodes created in the current step, while the black nodes will represent nodes created and joined in previous steps.

For this example, I will show the first few steps of finding $T(4,8)$ (refer to Figure 2.4(a)). Since we know there are four leaf nodes, we create four new nodes and check our list to see if any nodes are in the list. There are not, so these four nodes will be leaf nodes. We add the four nodes to the list, and determine what our recursive calls will be. From (2.7) we know that

$$T(4,8) = T(2,6) + T(4,6), \tag{2.8}$$

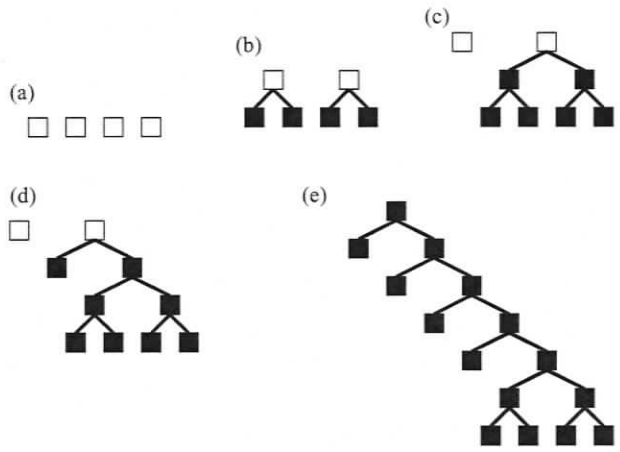

Figure 2.4. *Examples for Norwood's Algorithm*

so our first recursive call will be to $T(2,6)$, with a linked list of four nodes.

When we enter $T(2,6)$ (refer to Figure 2.4(b)), we first create two new nodes. We again check our list to see if there are any nodes in the list. This time there are four in the list. We delete the four nodes from the list, and add them as children to the two nodes which we just created. After adding them as children to the new nodes, we add the two new nodes to our list and check our recursive calls. According to (2.7), our recursive calls are

$$T(2,6) = T(2,5) + T(4,5), \tag{2.9}$$

so we make a call to $T(2,5)$ with a list of two nodes (refer to Figure 2.4(c)). We create two new nodes again and check our linked list. Our linked list only contains two nodes this time, so we delete them and make them the children of the right most node (to keep the property given in (2.1). We add both of our new nodes to the linked list, and check the recursive calls again.

We will continue this recurrence with $T(2,4)$ (refer to Figure 2.4(d)), where the same step we've described above will occur again (create two new nodes, add the two nodes in our list as children, add our new nodes to the list). After a couple more recursive calls, we will eventually reach the base case of $T(1,1)$ (refer to Figure 2.4(e)). When we reach the base case, we will always have two trees in our linked list, because the only way to get to $T(1,1)$ is from $T(2,2)$. We simply create one new node, which will be our root node, and set the left and right child of the node. Once we've

created the root node, we can print the tree and/or a compact code representation of the tree, and then return and go back in our recursive calls. Pseudocode for this algorithm can be found in Figure 2.5.

To speed up this program, the creation of nodes occurs at the beginning of the program. Since we know that there are N bottom level nodes, there must be $N-1$ internal nodes, and therefore $2N-1$ nodes altogether in each tree. So, at start up we create $2N-1$ nodes and place them in an array for future use. When we call the CREATENODE method, it returns the first node which is currently unused, and also makes sure to reset the node to default values. When exiting a call to T, a call is made to FREENODES, which updates the indexes of the free list, so that the nodes we allocaated in our current call can be used again in the next recursive call.

Another feature added to speed up the program is a check to see which values of q a certain p is valid for. This check is done in Step 4(a) in the pseudocode listing in Figure 2.5. As mentioned in Section 2.1.2, there are several methods for determining which column q some row p starts having non-zero values in. For this program, I precompute the $p(n)$ numbers in the program, and then check the numbers before issuing a new method call. These $p(n)$ numbers will be discussed in Section 3.1.1.

The final feature used to reduce the number of method calls is the fact that there can not be a compact code tree which has an odd number of nodes at any level, other than at the top level. Our for loop conditions deal with this by ignoring the odd numbers for p.

Since Norwood's algorithm deals with $T(p,q)$, our main method uses a for loop that makes calls to the methods $T(i,q)$ for $2 \leq i \leq q$ where i is even.

2.2.2.1 Explanation of Pseudocode

In this section, I will give a brief explanation of the pseudocode in Figure 2.5.

The variables `r` and `curr` refer to variables of type `node`; these variables have two attributes, `right` and `left`. Both `right` and `left` are pointers to node variables. The variable `list` is a linked list of `nodes`. `nodesCreated`, `loopP`, `p`, and `q` are all integer variables.

In the statement labelled (1), the CREATENODE method call returns a new node which has no children. The node `r` will be our root node. We set the right child of

Algorithm 2.2.2: T(p, q)

comment: Generate all trees with p bottom level nodes and q leaf nodes

```
nodesCreated ← 0
if p = 1 and q = 1
  then { r ← CREATENODE()                                   (1)
         r.right ← list.HEAD()
         r.left ← list.TAIL()
         print r
         return
while list.head() is old
  do { curr ← CREATENODE()                                  (2)
       curr.right ← list.DELHEAD()
       curr.left ← list.DELHEAD()
       list.ADD(curr)
       nodesCreated ← nodesCreated + 1
while nodesCreated < p
  do { curr ← CREATENODE()                                  (3)
       list.ADD(curr)
       nodesCreated ← nodesCreated + 1
if p/2 is even
  then loopP ← p/2
if p/2 is odd
  then loopP ← p/2 + 1
while loopP ≤ q − p/2
  do { if ISVALID(loopP, q − p/2)
         then T(loopP, q − p/2)                             (4)
       loopP ← loopP + 2
FREENODES()
```

Figure 2.5. *Pseudocode for the* T(p, q) *method*

our root node to be the first tree in our list, and the left child of our root node to be the second tree in our list. We then print the tree that we have now represented, and return to our previous recursive step.

If we reach (2), we know that we have $\mathtt{p} \geq 2$ and $\mathtt{q} \geq 2$. In (2), we check to see if our list contains any trees from a previous level. If it does, we first create a new node. After creating the new node, we set the right and left children of the node to be the first two trees in our list. The DELHEAD method deletes the list node at the head of the list, and returns the tree node that it contained. Since we always set the right child first, we will be sure that our tree is always built in increasing order from left to right (since the left subtrees will always have the same number or fewer nodes than right subtrees). We then add the current node to the end of our list, and continue. Eventually, we will have gone through all of the items in our list that were there from the previous method call, and the only trees in our list will be the new ones that we've created in our current method call. When we reach this point, we exit the loop and move on to (3).

When we reach (3), we have either created some number of nodes in (2) or we have not created any new nodes; either way, the number of nodes that we have created so far is stored in the variable `nodesCreated`. In (3), we create leaf nodes one by one and add them to the list, until we have created a total of `p` nodes in the two loops.

We then set `loopP` based on whether $\mathtt{p}/2$ is even or odd. If $\mathtt{p}/2$ is even, `loopP` will start at $\mathtt{p}/2$. If $\mathtt{p}/2$ is odd, we need to set `loopP` to $\mathtt{p}/2 + 1$, since all of our recursive calls must have an even value for `p`. Finally, in (4), we do our recursive calls based on (2.7). The method ISVALID is a boolean function, which uses the $p(n)$ numbers to determine whether a recursive call to $\mathrm{T}(p, q)$ will return any trees. If ISVALID returns true, we know that we need to make a recursive call to generate more trees. If it returns false, we do not need to make a recursive call.

2.2.2.2 Analysis of Norwood's Algorithm

As discussed in Section 2.2.1.2, the main goal of implementing this algorithm was to try and determine whether or not this algorithm was a CAT algorithm. Similar to what was done above, the first test was to try and determine experimentally whether or not this algorithm is CAT, and if so, approximately what the constant would be.

This constant was easier to find in Norwood's algorithm then it was in Khosravifard's algorithm.

Let $T(q)$ be the number of compact code trees which have q leaf nodes. In other words,

$$T(q) = \sum_{p=2}^{q} T(p,q) \tag{2.10}$$

Similarly, let $M(p,q)$ be the number of method calls the program makes to generate all of the possible compact code trees for $T(p,q)$. Also, let $M(q)$ be the number of method calls the program makes to generate all of the compact code trees for $T(q)$. Note that both $M(p,q)$ and $M(q)$ will not make any calls to any $M(p,q) = 0$, which only occurs when $q < p(n)$, and which will be discussed later. This fact is ignored in proofs to try and make the proofs easier to read.

Theorem 2.2.1 *If $p = q = 1$, then $M(p,q) = 1$. If $p \geq 2$ and $q \geq 2$, and $q \geq p(n)$, then*

$$M(p,q) = 1 + \sum_{i=p/2}^{q-p/2} M(i, q - p/2) \tag{2.11}$$

Proof. If $p = q = 1$, then the call will be caught by the base case and creates one tree of size 1.

If $p \geq 2$ and $q \geq 2$, then the case will not be caught by the base case and will instead fall into the main part of the algorithm. Our method T only makes additional method calls to T inside a for loop which goes from $p/2$ to $q - p/2$. Therefore, the number of calls being made to T is given by adding 1 for the initial call to T, and $\sum_{i=p/2}^{q-p/2} M(i, q - p/2)$ for the additional calls from inside of our initial call, and that gives us the equation above. ■

Since we now have recurrence relations for both $T(p,q)$ and $M(p,q)$, it was easy to create a table to try and determine whether this algorithm is CAT or not. By creating this table, it quickly became evident that while the algorithm is not CAT for some particular combination of $\frac{M(p,q)}{T(p,q)}$, it may be CAT for $\frac{M(q)}{T(q)}$.

Theorem 2.2.2 *The given algorithm is not a CAT algorithm for the generation of the trees $T(p,q)$.*

Proof. The algorithm is not CAT because the expression $\frac{M(p,q)}{T(p,q)}$ can not be bound by a constant. For example, take $p = q = 2^h$ for some positive integer h. It is obvious that in this case, $T(p,q) = 1$, since there is only one tree which can have 2^h leaf nodes with 2^h of those leaf nodes at the bottom most level. On the other hand, one can see by induction that since $M(2^0, 2^0) = M(1,1) = 1$, then for any h

$$\begin{aligned} M(2^h, 2^h) &= 1 + \sum_{i=2^{h-1}}^{2^h - 2^{h-1}} M(i, 2^h - 2^{h-1}) \\ &= 1 + \sum_{i=2^{h-1}}^{2^{h-1}} M(i, 2^{h-1}) \\ &= 1 + M(2^{h-1}, 2^{h-1}) \\ &= h + 1 \end{aligned}$$

Thus, as h goes towards infinity, so will $\frac{M(p,q)}{T(p,q)}$, and therefore we can not bound the expression by a constant. ■

Note that while this proof uses the powers of 2 to prove it, that doesn't necessarily mean that the bound is $\lg n$. For instance, while $\frac{M(32,32)}{T(32,32)} = 6$, it turns out that $\frac{M(30,32)}{T(30,32)} = 7$.

This not withstanding, a large majority of the values in the table of $\frac{M(p,q)}{T(p,q)}$ values seem to be going towards a constant value. As the number of outputs increase, the values of this ratio are approximately 4.577. For $p = 2$, $\frac{M(2,q)}{T(2,q)} \approx 4.577$ for $25 \leq q \leq 76$, and there are similar results for $p = 4, 6, 8, 10$. By the time we get to $q = 76$, $\frac{M(p,q)}{T(p,q)} \approx 4.577$ for $p = 2, 4, 6, \ldots, 48$.

This is especially interesting considering our next two lemmas.

Lemma 2.2.3 *For $q \geq 2$, $T(2, q+1) = T(q)$.*

Proof. Using (2.7), we see $T(2, q+1) = \sum_{p=1}^{q} T(p,q) = T(q)$. ■

Lemma 2.2.4 *For $q \geq 2$, $M(2, q+1) = 1 + M(q)$.*

Proof. By Theorem 2.2.1, we have $M(2, q+1) = 1 + \sum_{i=1}^{q} M(i,q) = 1 + M(q)$. ■

Thus, for large values of q, we can say that $M(2, q+1) = M(q)$. Below are some additional facts relating to the performance of this algorithm.

Lemma 2.2.5 *If $k \geq 1$ is odd, then if $n \geq 4$,*

$$T(2(k+1), n) = T(2k, n-1). \tag{2.12}$$

Proof.

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

The second equality is true because $T(p, q) = 0$ when p is odd, and the third equality is simply updating the summation notation. The final equality comes from (2.6). ■

Lemma 2.2.6 *If $k \geq 2$ is even and $T(2k, n-1) > 0$,*

$$T(2(k+1), n) \leq T(2k, n-1). \tag{2.13}$$

Proof.

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

The second step comes from the fact that we are subtracting something which is not negative from a positive value. The third step is simply rewriting the summation, and the fourth step comes from (2.6). ■

Corollary 2.2.7 *For $p \geq 4$ and $q \geq 4$, $T(p,q) \leq T(p-2,q-1)$.*

Proof. Follows directly from previous two lemmas. ■

Lemma 2.2.8 *For $q \geq 2$, $T(2,q) \geq \sum_{i=4}^{q} T(i,q)$.*

Proof. If q is odd, then $T(q,q) = 0$, since we can not have any trees with an odd number of nodes at the bottom level of the tree. Therefore, the last term of the summation which might be non-zero is $T(q-1,q)$.

$$\begin{aligned} T(2,q) &= T(2,q-1) + T(4,q-1) + \cdots + T(q-3,q-1) + T(q-1,q-1) \\ &= T(4,q) + T(4,q-1) + \cdots + T(q-3,q-1) + T(q-1,q-1) \\ &\geq T(4,q) + T(6,q) + \cdots + T(q-1,q) \\ &\geq \sum_{i=4}^{q} T(i,q), \end{aligned}$$

since, by Corollary 2.2.7, $T(4,q) \leq T(2,q-1)$, $T(6,q) \leq T(4,q-1)$, and so on until $T(q-1,q) \leq T(q-3,q-1)$.

If q is even, then $T(q,q)$ might be non-zero, so the last term of the summation which might be non-zero is $T(q,q)$.

$$\begin{aligned} T(2,q) &= T(2,q-1) + T(4,q-1) + \cdots + T(q-4,q-1) + T(q-2,q-1) \\ &\geq T(4,q) + T(6,q) + \cdots + T(q-2,q) + T(q,q) \end{aligned}$$

■

Lemma 2.2.9 *For $n \geq 4$,*

$$2T(4,n) \geq T(2,n). \tag{2.14}$$

Proof. By Lemma 2.2.5, we know that $T(4,n) = T(2,n-1)$. Therefore,

$$\begin{aligned} 2T(4,n) &= 2T(2,n-1) = T(2,n-1) + T(2,n-1) \\ &\geq T(2,n-1) + T(4,n-1) + \cdots + T(n-1,n-1) \\ &\geq T(2,n), \end{aligned}$$

by using Lemma 2.2.8 to get the second step, and 2.6 for the final step. ■

Unfortunately, I was unable to prove the following inequality

$$\frac{M(q)}{T(q)} \leq 5. \tag{2.15}$$

The problems arose due to the lack of a suitable bound for the $M(p, q)$ numbers. Also, trying to represent the numbers $M(q)$ and $T(q)$ in terms of $q - 1$ proved messy, making an inductive proof difficult.

2.2.3 Final Thoughts

Experimentally, it seems evident that both of these algorithms are CAT algorithms. Due to the nature of the Khosravifard algorithm, the only way to experiment was through executions of the program. At least for $1 \leq N \leq 33$, there seemed to be a bound somewhere around 5.198. For the Norwood algorithm, it's recursive nature allowed formulas to be created in Maple and spreadsheets that show a bound of 4.577 for $M(q)$ where $1 \leq q \leq 76$. Unfortunately, I was unable to finalize proofs that would prove these bounds further than the experimental evidence. One possible area of future work would be to try and prove that one or both of these algorithms in fact run in CAT.

One interesting thing to mention is the different nature of these algorithms. As described in the respective sections, the algorithm based on the paper by Khosravifard, Esmaeili, Saidi, and Gulliver creates just the code lengths of the form $(l_1, l_2, \ldots, l_N)$. On the other hand, Norwood's algorithm directly creates only the tree structure itself. While one can convert the code lengths to a tree and the tree to code lengths, it still takes added time. This means that having implementations of both algorithms could provide helpful if one wanted to generate either just trees or just the code lengths.

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	2	3	4	4.0	4.33	4.40	4.44	4.50	4.54	4.54	4.55	4.56	4.56	4.57	4.57	4.57	4.57	4.57	4.58
4			3	4.0	4.00	4.33	4.40	4.44	4.50	4.54	4.54	4.55	4.56	4.56	4.57	4.57	4.57	4.57	4.57
6						4.00	5.00	4.50	4.50	4.57	4.58	4.55	4.56	4.57	4.57	4.57	4.57	4.58	4.58
8							4.00	5.00	4.50	4.50	4.57	4.58	4.55	4.56	4.57	4.57	4.57	4.57	4.58
10											5.00	5.00	5.00	4.67	4.64	4.65	4.62	4.61	4.60
12												5.00	5.00	5.00	4.67	4.64	4.65	4.62	4.61
14														5.00	6.00	5.00	4.75	4.75	4.73
16															5.00	6.00	5.00	4.75	4.75

Table 2.5. *$M(p,q)/T(p,q)$; for $p = 2, 4, 6, \ldots, 16$ and $2 \leq q \leq 20$*

Chapter 3 — Meta-Fibonacci Sequences and Complete Binary Trees

A *meta-Fibonacci* recurrence relation is a recurrence relation defined by

$$a(n) = a(x(n) - a(n-1)) + a(y(n) - a(n-2))$$

where $x(n)$ and $y(n)$ are linear functions. Jackson and Ruskey [7] looked at the sequences where $x(n) = n - s$ and $y(n) = n - (s+1)$, where $s \geq 0$. These sequences had been studied before; Tanny [14] looked at the case where $s = 1$ while Conolly studied the case of $s = 0$. Jackson and Ruskey [7] were able to provide a combinatorial meaning for the sequence.

In this chapter, we will be looking at *generalized meta-Fibonacci* sequences, which are the sequences $(a(1), a(2), \ldots)$ of solutions to the recurrence

$$a(n) = \sum_{i=1}^{k} a(n - i - (s-1) - a(n-i)). \tag{3.1}$$

This sequence was studied recently by Callaghan, Chow, and Tanny [2]. In [2], it was shown that these meta-Fibonacci sequences can change dramatically based on the initial conditions.

In this chapter, we will show that for a specific and slightly unusual way of numbering the nodes in k-ary trees, (3.1) will count the number of nodes at the bottom level of the tree. In Section 3.1, I will give a brief summary of previous results related to meta-Fibonacci sequences, in particular results from the paper by Jackson and Ruskey [7]. In Section 3.2, I extend the work done by Jackson and Ruskey, which was based on binary trees, to k-ary trees. Finally, in Section 3.2.2 I will look at the generating functions for the sequences from Section 3.2; and in Section 3.2.3, I will use these generating functions to show that the meta-Fibonacci sequences can count certain restricted compositions of integers.

3.1 Previous Work

Most of the previous work on these meta-Fibonacci sequences was based around the behaviours of these sequences as related to the $x(n)$ and $y(n)$ functions and the initial conditions. For instance, Callaghan, Chow, and Tanny [2] looked at the sequence

$$T_{a,k}(n) = \sum_{i=0}^{k-1} T_{a,k}(n-i-a-T_{a,k}(n-i-1)) \tag{3.2}$$

where $n > a+k$, $k \geq 2$. This is the same as our recurrence given in (3.1). In the paper, they show that depending on the choice of initial conditions, the sequences can look radically different. For example, take $T_{0,3}(n)$. Using the initial conditions $(1,2,3)$ gives us the sequence

$$(1,2,3,3,4,5,6,6,7,8,9,9,9,10,11,12,12,13,14,15,15,16,\ldots),$$

which we will see later is the $a_{0,3}(n)$ sequence. However, with initial conditions $(-1,0,1)$ we get the sequence

$$(-1,0,1,3,2,4,4,7,4,7,7,9,8,9,11,10,10,13,15,13,13,13,18,15,\ldots),$$

which is not monotone.

3.1.1 Jackson-Ruskey

The paper by Jackson and Ruskey [7] was the first paper that gave a combinatorial meaning to a meta-Fibonacci sequence. They proved interesting properties based on binary trees for the recurrence

$$a_{s,2}(n) = a_{s,2}(n-s-a_{s,2}(n-1)) + a_{s,2}(n-s-1-a_{s,2}(n-2)). \tag{3.3}$$

Figure 3.1 is the start of an infinite binary tree called $\mathcal{F}_{0,2}$. There is a sequence of smaller binary trees of size $1,1,3,7,\ldots,2^h-1,\ldots$ These smaller trees are numbered with a preorder traversal. Joining these smaller trees is what they [7] called *super nodes*, but could also be thought of as a *delay path*. The super nodes are numbered after its left subtree has been labelled, but before the right subtree has been labelled. The right subtree is labelled using a preorder traversal. Think of the delay path as

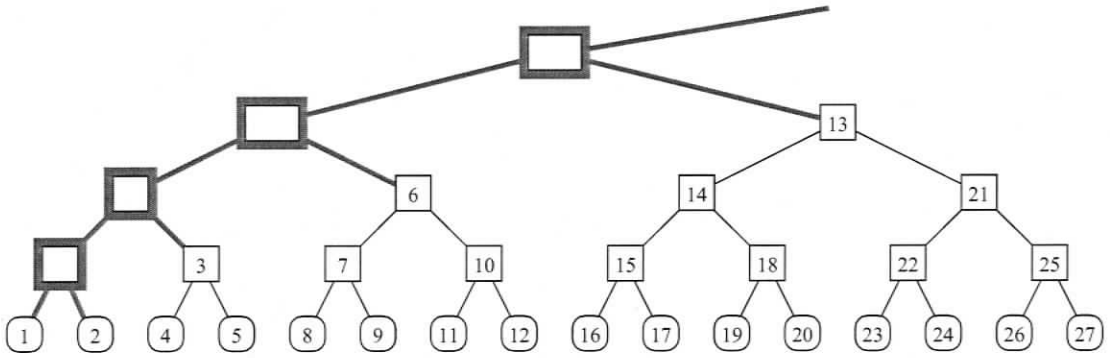

Figure 3.1. $\mathcal{F}_{0,2}$

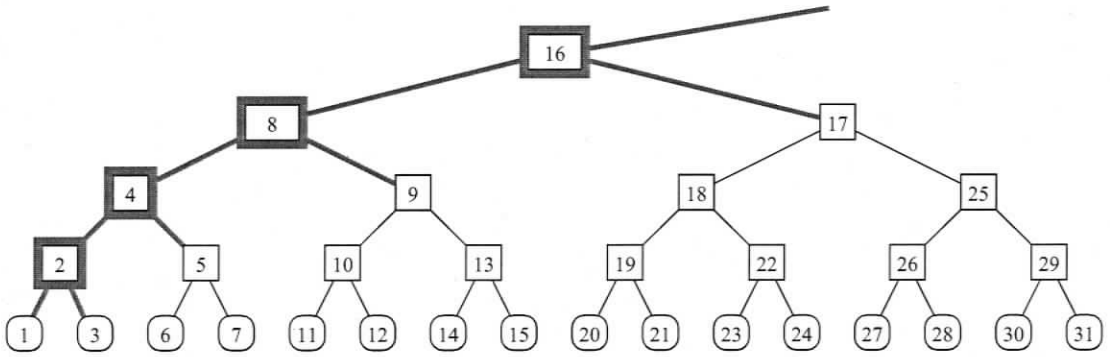

Figure 3.2. $\mathcal{F}_{1,2}$

being associated with the parameter s, which is the number of ordinary nodes that each super node represents. For example, in Figure 3.1 each super node represents 0 ordinary nodes, and is not numbered. In Figure 3.2 each super node represents 1 ordinary node and receives one number, while Figure 3.3 shows $\mathcal{F}_{2,2}$ where each super node receives two numbers.

Let $\mathcal{T}_{s,2}(n)$ be the tree which uses the first n labelled nodes of the infinite tree $\mathcal{F}_{s,2}$. Jackson and Ruskey [7] found that, with the correct initial conditions, $a_{s,2}(n)$ counts the number of nodes at the bottom level of $\mathcal{T}_{s,2}(n)$. The initial conditions they used were: if $0 \leq n \leq s+1$, then $a_{s,2}(n) = 1$, and if $n = s+2$, then $a_{s,2}(n) = 2$. A table of these numbers can be found in Table 3.1. They also defined two other sequences. Let $d_{s,2}(n)$ be 1 if the n-th node is a leaf, and 0 if the n-th node is not a leaf. They also discussed what they call the $p_{s,2}(n)$ numbers, which are the positions occupied

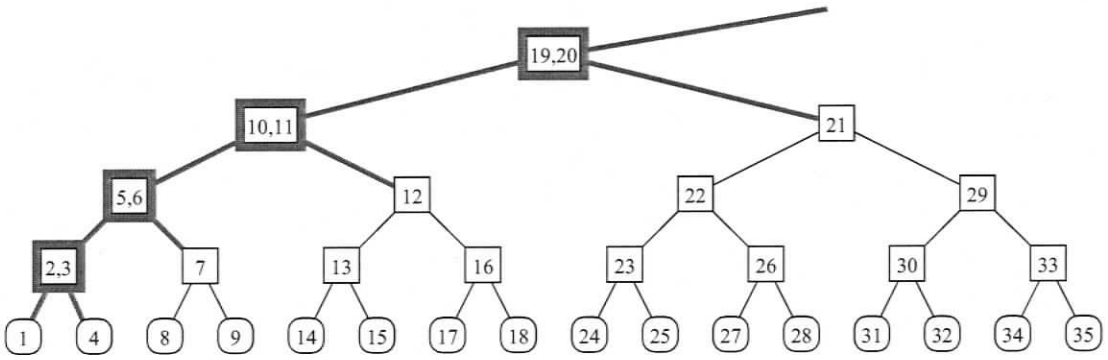

Figure 3.3. $\mathcal{F}_{2,2}$

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$a_{0,2}(n)$	1	2	2	3	4	4	4	5	6	6	7	8	8	8	8	9
$a_{1,2}(n)$	1	1	2	2	2	3	4	4	4	4	5	6	6	7	8	8
$a_{2,2}(n)$	1	1	1	2	2	2	2	3	4	4	4	4	4	5	6	6
$d_{0,2}(n)$	1	1	0	1	1	0	0	1	1	0	1	1	0	0	0	1
$d_{1,2}(n)$	1	0	1	0	0	1	1	0	0	0	1	1	0	1	1	0
$d_{2,2}(n)$	1	0	0	1	0	0	0	1	1	0	0	0	0	1	1	0
$p_{0,2}(n)$	1	2	4	5	8	9	11	12	16	17	19	20	23	24	26	27
$p_{1,2}(n)$	1	3	6	7	11	12	14	15	20	21	23	24	27	28	30	31
$p_{2,2}(n)$	1	4	8	9	14	15	17	18	24	25	27	28	31	32	34	35

Table 3.1. $a_{s,2}(n)$, $d_{s,2}(n)$ *and* $p_{s,2}(n)$ *for* $s = 0, 1, 2$, *and* $1 \leq n \leq 16$.

by 1's in the $d_{s,2}$ sequence. The $p_{s,2}(n)$ numbers are defined as

$$p_{s,2}(n) = \min\{j \ : \ a_{s,2}(j) = n\}. \tag{3.4}$$

These $p_{s,2}(n)$ numbers are related to the table of values in Norwood's paper (Table 2.3 on page 8). As mentioned above, the numbers $p_{1,2}(n)$ (see Table 3.1) are one less than q, where q is which column some row p starts in for Norwood's sequence. From Norwood's table, one can see that row $p = 2$ starts at column $q = 2 = p_{1,2}(1) + 1$, row $p = 4$ starts at column $q = 4 = p_{1,2}(2) + 1$, row $p = 6$ at column $q = 7 = p_{1,2}(3) + 1$, row $p = 8$ at column $q = 8 = p_{1,2}(4) + 1$, and so forth. In my implementation of Norwood's algorithm, I generate the $p_{1,2}(n)$ numbers to work as a boundary for generating the trees $T(p, q)$.

Finally, Jackson and Ruskey also provided generating functions for the sequences $a_{s,2}(n)$, $p_{s,2}(n)$, and $d_{s,2}(n)$. Jon Perry [12] discussed a form of compositions that, at least experimentally, the sequence $a_{1,2}(n)$ seemed to count. These compositions were the compositions of n such that, for a positive integer m,

$$x_0 + x_1 + \cdots + x_m = n \text{ where } x_i \in 1, 2^i \text{ for } i = 0, 1, \ldots, m. \tag{3.5}$$

In this thesis, I will be using the same notation as Jackson and Ruskey used to denote these compositions, $1 + \langle 1, 2\rangle + \langle 1, 4\rangle + \cdots$.

Using the generating functions for $a_{s,2}(n)$, Jackson and Ruskey were also able to prove a set of specifications similar to those mentioned by Perry. They proved that for $s \geq 1$, $a_{s,2}(n)$ is equal to the number of compositions of n with specification

$$\langle 1, 2, \ldots, s\rangle + \langle s, 2 + s - 1\rangle + \langle s, 4 + s - 1\rangle + \langle s, 8 + s - 1\rangle + \cdots . \tag{3.6}$$

As one can see, when $s = 1$, these specifications are exactly the same as Perry's.

3.2 New Work

While Jackson and Ruskey had worked on the problem for binary trees, there was an open question as to whether a similar equation could be used to extend these results to k-ary trees. The idea was to try and use definitions and initial conditions similar to the ones used by Jackson and Ruskey, and extend them using the generalized meta-Fibonacci sequence for $k \geq 2$. In this thesis, I was able to find equations that either take the same form as those from Jackson and Ruskey's paper [7], or are very similar. We will also be able to see by the combinatorial interpretation that $a_{s,k}(n)$ is monotone, that its consecutive terms increase by 0 or 1, and therefore, that the sequences hit every positive integer. I have provided two tables of values, Table 3.2 for the $a_{s,k}(n)$ where $k = 3$ and Table 3.3 for $k = 4$, as well as some example tree images to show how they were created.

Figure 3.4 shows the infinite tree $\mathcal{F}_{0,3}$ and Figure 3.7 shows the infinite tree $\mathcal{F}_{0,4}$. Similar to the definition given before for the tree $\mathcal{F}_{0,2}$, we start with k complete trees of size 1, then we continue with $k-1$ complete trees of sizes $\frac{k^2-1}{k-1}, \frac{k^3-1}{k-1}, \ldots, \frac{k^h-1}{k-1}, \ldots$. We refer to this collection of the $k-1$ k-trees as the h *subforest*. As above, these

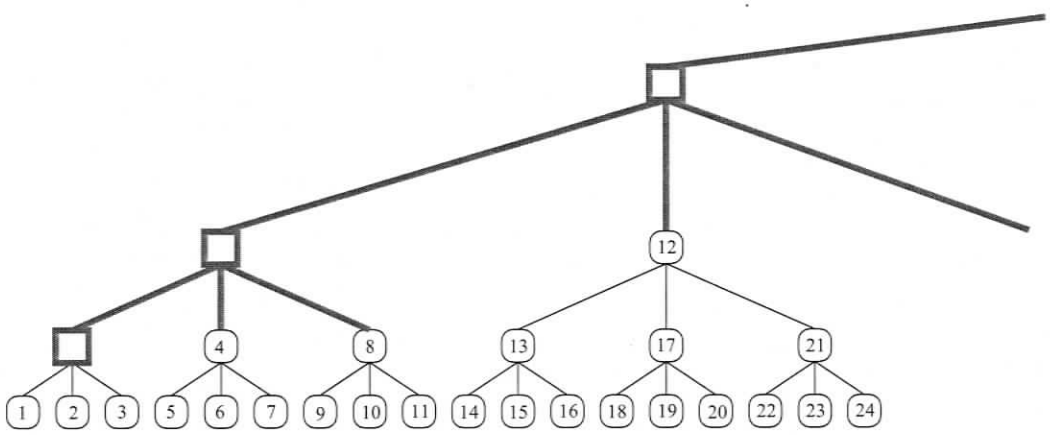

Figure 3.4. $\mathcal{F}_{0,3}$

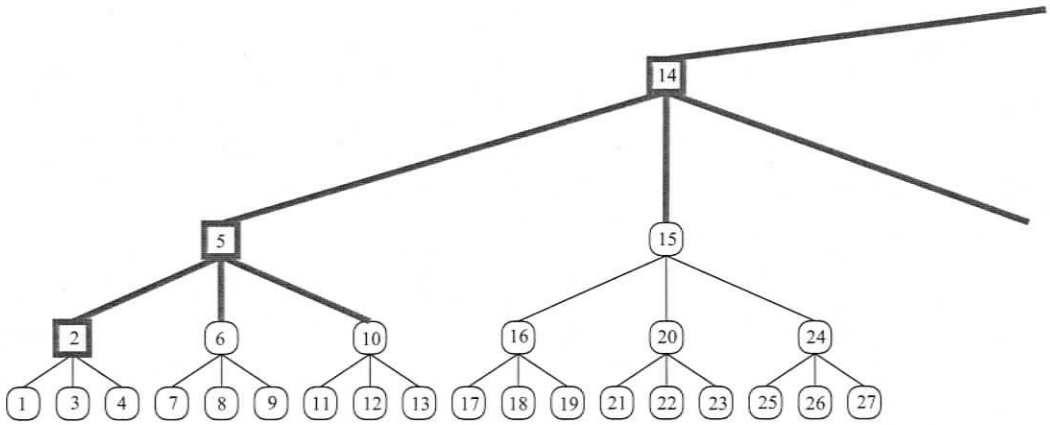

Figure 3.5. $\mathcal{F}_{1,3}$

subtrees will be labelled using a preorder traversal. Now, to obtain a completed $\mathcal{F}_{0,k}$, add a path of delay nodes connecting the subtrees from left-to-right. The delay nodes will be labelled immediately after their leftmost subtree has been numbered. The parameter s will determine how many regular nodes each delay node represents; so if $s = 1$ label the delay nodes with one number, if $s = 2$ label the delay nodes with two numbers, and so forth. The trees $\mathcal{F}_{0,3}, \mathcal{F}_{1,3}, \mathcal{F}_{2,3}, \mathcal{F}_{0,4}, \mathcal{F}_{1,4}$ are shown in Figures 3.4-3.8.

For the purposes of the remainder of this thesis, define $\mathcal{T}_{s,k}(n)$ to be the tree induced by the first n nodes of $\mathcal{F}_{s,k}$ and let $a_{s,k}(n)$ be the number of nodes at the

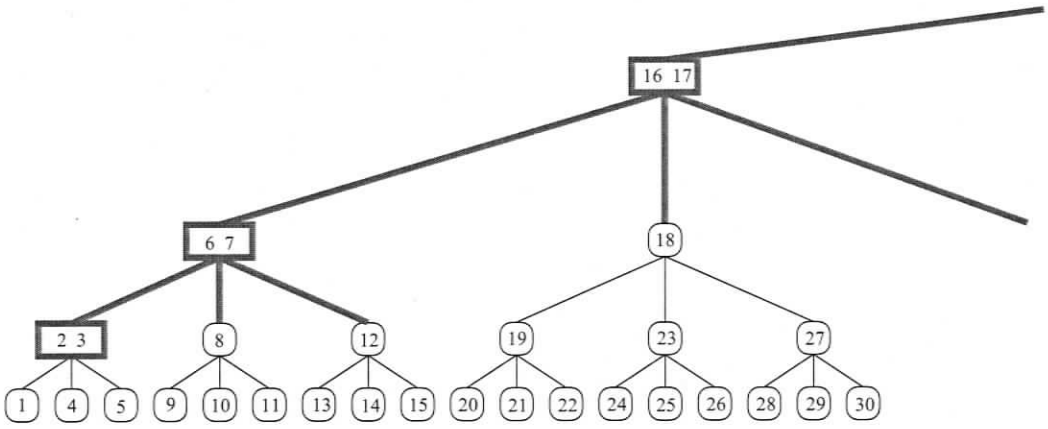

Figure 3.6. $\mathcal{F}_{2,3}$

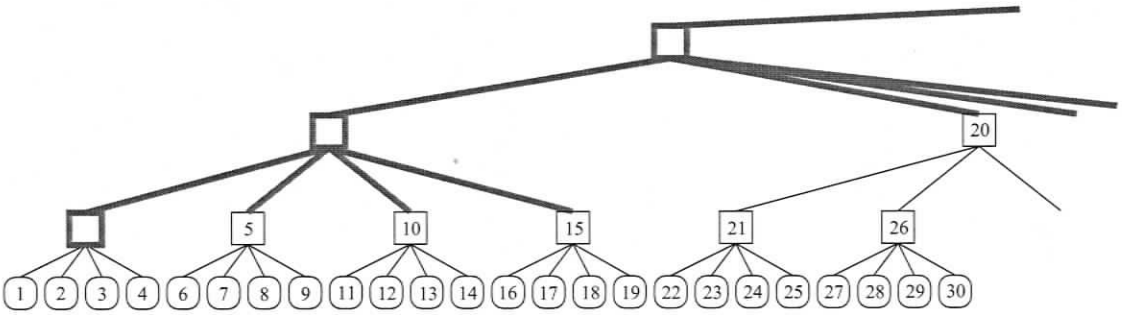

Figure 3.7. $\mathcal{F}_{0,4}$

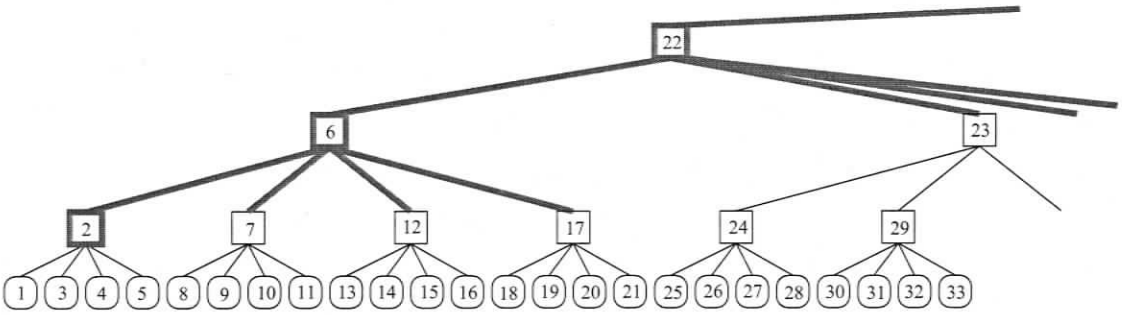

Figure 3.8. $\mathcal{F}_{1,4}$

$n=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$a_{0,3}(n)$	1	2	3	3	4	5	6	6	7	8	9	9	9	10	11	12
$a_{1,3}(n)$	1	1	2	3	3	3	4	5	6	6	7	8	9	9	9	9
$a_{2,3}(n)$	1	1	1	2	3	3	3	3	4	5	6	6	7	8	9	9
$d_{0,3}(n)$	1	1	1	0	1	1	1	0	1	1	1	0	0	1	1	1
$d_{1,3}(n)$	1	0	1	1	0	0	1	1	1	0	1	1	1	0	0	0
$d_{2,3}(n)$	1	0	0	1	1	0	0	0	1	1	1	0	1	1	1	0
$p_{0,3}(n)$	1	2	3	5	6	7	9	10	11	14	15	16	18	19	20	22
$p_{1,3}(n)$	1	3	4	7	8	9	11	12	13	17	18	19	21	22	23	25
$p_{2,3}(n)$	1	4	5	9	10	11	13	14	15	20	21	22	24	25	26	28

Table 3.2. *$a_{s,3}(n)$, $d_{s,3}(n)$ and $p_{s,3}(n)$ for $s = 0, 1, 2$ and $1 \leq n \leq 16$.*

bottom-most level of $\mathcal{T}_{s,k}(n)$. Also, just as above, $d_{s,k}(n)$ is 1 if the nth node in $\mathcal{T}_{s,k}(n)$ is a leaf node and 0 otherwise, and let $p_{s,k}(n)$ be the positions of the 1's in the sequence $d_{s,k}$. By these definitions, the sequences $a_{s,2}(n), d_{s,2}(n)$, and $p_{s,2}(n)$ are all the same as the sequences from [7]. For the sake of readability, the subscripts s and/or k will be dropped when no confusion can arise.

Since a tree $\mathcal{T}_{0,k}(0)$ has no nodes at the bottom level, $a_{s,k}(0) = p_{s,k}(0) = d_{s,k}(0) = 0$.

3.2.1 Determining the Sequences

Theorem 3.2.1 *If $1 \leq n \leq s+1$, then $a_{s,k}(n) = 1$. If $n = s+i$ and $2 \leq i \leq k$ then $a_{s,k}(n) = i$. If $n > s+k$, then*

$$a_{s,k}(n) = \sum_{i=1}^{k} a_{s,k}(n - i - (s-1) - a_{s,k}(n-i)). \tag{3.7}$$

Proof. First observe that if all the leaves at the last level are removed from $\mathcal{F}_{s,k}$, then the same structure remains, except that the leftmost super-node needs to be made into an ordinary node (by subtracting $s-1$). We will refer to this process as *chopping* the last level. The number of nodes at the penultimate level of $\mathcal{T}_{s,k}(n)$ can be found by chopping the last level of the tree, and then counting how many nodes are

$n=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$a_{0,4}(n)$	1	2	3	4	4	5	6	7	8	8	9	10	11	12	12	13
$a_{1,4}(n)$	1	1	2	3	4	4	4	5	6	7	8	8	9	10	11	12
$a_{2,4}(n)$	1	1	1	2	3	4	4	4	4	5	6	7	8	8	9	10
$d_{0,4}(n)$	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1
$d_{1,4}(n)$	1	0	1	1	1	0	0	1	1	1	1	0	1	1	1	1
$d_{2,4}(n)$	1	0	0	1	1	1	0	0	0	1	1	1	1	0	1	1
$p_{0,4}(n)$	1	2	3	4	6	7	8	9	11	12	13	14	16	17	18	19
$p_{1,4}(n)$	1	3	4	5	8	9	10	11	13	14	15	16	18	19	20	21
$p_{2,4}(n)$	1	4	5	6	10	11	12	13	15	16	17	18	20	21	22	23

Table 3.3. *$a_{s,4}(n)$, $d_{s,4}(n)$ and $p_{s,4}(n)$ for $s = 0, 1, 2$ and $1 \leq n \leq 16$.*

at the bottom level of a tree containing that same number of nodes $(n-(s-1)-a(n).)$ Therefore, the number of nodes at the penultimate level of $\mathcal{T}_{s,k}(n)$ is

$$a(n - (s-1) - a(n)). \tag{3.8}$$

Also observe that if each node at the penultimate level of $\mathcal{T}_{s,k}(n)$ has k children, then $a(n)/k$ counts the number of nodes at the penultimate level, since each node at the penultimate level would have k children at the bottom level. However, the rightmost node on the penultimate level will not necessarily have k children. Assume that the rightmost node on the penultimate level has r children. If we added $k - r$ nodes to the bottom level of $\mathcal{T}_{s,k}(n)$, the rightmost node would have k children, and we divide by k to determine the number of nodes at the penultimate level. Therefore, the number of nodes at the penultimate level of $\mathcal{T}_{s,k}(n)$ is

$$(a(n) + k - r)/k = \lceil a(n)/k \rceil. \tag{3.9}$$

Finally, observe that if the rightmost node at the penultimate level had r children, and we subtracted r nodes from the bottom level of $\mathcal{T}_{s,k}(n)$, the rightmost node at the penultimate level would have 0 children. If we divided by k, we would be counting every node at the penultimate level other than the rightmost node. Therefore, the number of nodes at the penultimate level, other than the rightmost node, of $\mathcal{T}_{s,k}(n)$ is

$$(a(n) - r)/k = \lfloor a(n)/k \rfloor. \tag{3.10}$$

We split the proof into two broad cases depending on whether n or not is a leaf or not; i.e., whether $d_{s,k}(n) = 1$ (Case 1) or $d_{s,k}(n) = 0$ (Case 2). In either case, we will be computing $a_{s,k}(n)$, the number of nodes at the bottom level of our tree $\mathcal{T}_{s,k}(n)$, by counting each node p that is at the penultimate level of our tree p_c times, where p_c is the number of children of node p.

Case 1: If $d(n) = 1$, then node n is the rth child of node $n - r$. The r trees $\mathcal{T}_{s,k}(n-1)$, $\mathcal{T}_{s,k}(n-2)$, ..., $\mathcal{T}_{s,k}(n-r)$ have node $n-r$ at the penultimate level, and therefore have $(a(n) + k - r)/k$ nodes at the penultimate level. The $k - r$ trees $\mathcal{T}_{s,k}(n-r-1)$, $\mathcal{T}_{s,k}(n-r-2)$, ..., $\mathcal{T}_{s,k}(n-k)$ do not have $n-r$ at the penultimate level, and therefore have $(a(n) - r)/k$ nodes at the penultimate level. Remember that for any m, $\mathcal{T}_{s,k}(m)$ has $a(m - (s-1) - a(m))$ nodes at the penultimate level. Thus

$$\begin{aligned}
&\sum_{i=1}^{k} a(n - i - (s-1) - a(n-i)) \\
&= \sum_{i=1}^{r} a(n - i - (s-1) - a(n-i)) + \sum_{i=r+1}^{k} a(n - i - (s-1) - a(n-i)) \\
&= \sum_{m=n-r}^{n-1} a(m - (s-1) - a(m)) + \sum_{m=n-k}^{n-r-1} a(m - (s-1) - a(m)) \\
&= \sum_{m=n-r}^{n-1} (a(n) + k - r)/k + \sum_{m=n-k}^{n-r-1} (a(n) - r)/k \\
&= r(a(n) + k - r)/k + (k-r)(a(n) - r)/k \\
&= (ra(n) + rk - r^2 + ka(n) - kr - ra(n) + r^2)/k \\
&= ka(n)/k \\
&= a(n).
\end{aligned}$$

For example, look at node 19 in $\mathcal{F}_{2,5}$ ($\mathcal{T}_{2,5}(33)$ is given in Figure 3.9 on page 33). By looking at the diagram, we can see that $a(19) = 13$. Node 19 is a leaf node, and its parent is node 16. Therefore, $r = 19-16 = 3$. The 3 trees $\mathcal{T}_{2,5}(18)$, $\mathcal{T}_{2,5}(17)$, and $\mathcal{T}_{2,5}(16)$ have three nodes at the penultimate level: the supernode $2, 3$, 10, and 16. Notice that $a(18 - (2-1) - a(18)) = a(5) = 3$, and the same holds true for 16 and 17. The 2 trees $\mathcal{T}_{2,5}(15)$ and $\mathcal{T}_{2,5}(14)$ have two nodes at the penultimate level: the supernode $2, 3$ and 10. Notice again that $a(15 - (2-1) - a(15)) = a(4) = 2$, and the same holds true with 14. Counting the nodes at the penultimate level for these five

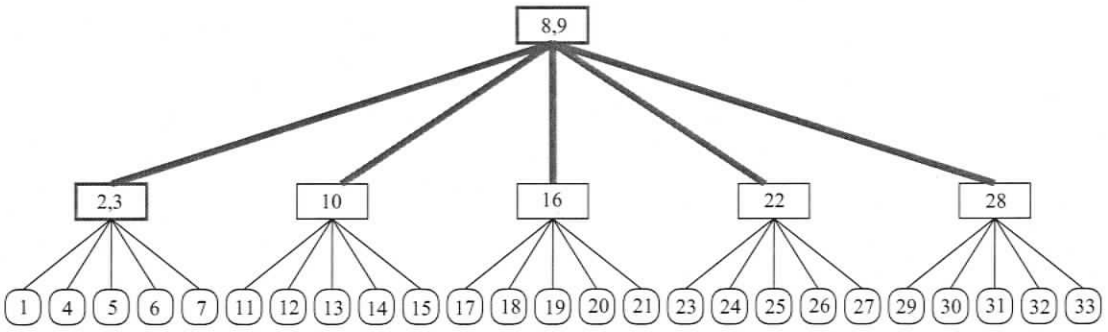

Figure 3.9. $\mathcal{T}_{2,5}(33)$

trees gives us $3+3+3+2+2=13$.

Case 2: If $d(n)=0$, node n is an internal node which might be at the penultimate level of $\mathcal{T}_{s,k}(n)$, but is not at the penultimate level of any tree $\mathcal{T}_{s,k}(n-i)$ where $0<i<n$. Therefore, the k trees $\mathcal{T}_{s,k}(n-1), \mathcal{T}_{s,k}(n-2), \cdots, \mathcal{T}_{s,k}(n-k)$ have $a(n)/k$ nodes at the penultimate level. Remember that for any m, $\mathcal{T}_{s,k}(m)$ has $a(m-(s-1)-a(m))$ nodes at the penultimate level. Thus

$$\begin{aligned}
&\sum_{i=1}^{k} a(n-i-(s-1)-a(n-i)) \\
&\quad = \sum_{m=n-k}^{n-1} a(m-(s-1)-a(m)) \\
&\quad = \sum_{m=n-k}^{n-1} a(n)/k \\
&\quad = k(a(n))/k \\
&\quad = a(n).
\end{aligned}$$

For example, consider node 24 in $\mathcal{F}_{1,3}$ in Figure 3.5 on page 28. For all three of the trees $\mathcal{T}_{1,3}(23)$, $\mathcal{T}_{1,3}(22)$, and $\mathcal{T}_{1,3}(21)$, the nodes at the penultimate level are 2, 6, 10, 16, and 20. Summing the number of nodes at the penultimate level of those trees gives us $5+5+5=15$, which is equal to $a(24)$. ■

Define $\mathcal{D}_{s,k}$ to be the infinite string $d_{s,k}(1)d_{s,k}(2)d_{s,k}(3)\cdots$. Let $D_{n,k}$ be the finite string defined by $D_{0,k}=1$ and $D_{n+1,k}=0(D_{n,k})^k$. Let $E_{n,k}$ be the finite string defined by $E_{0,k}=1$ and $E_{n+1,k}=(E_{n,k})^k 0$. As above, we will omit the subscript k when no

confusion can arise.

Lemma 3.2.2 *For all $n \geq 0$, we have $0^n E_n = D_n 0^n$.*

Proof. [Proof by induction] Base Case: For $n = 0$, $E_0 = D_0 = 1$. Assuming that our formula is true for n, for $n+1$ we have

$$\begin{aligned} 0^{n+1}E_{n+1} &= 0\ 0^n E_{n+1} \\ &= 0\ 0^n (E_n)^k 0 \\ &= 0\ (D_n)^k\ 0^n\ 0 \\ &= D_{n+1}\ 0^n\ 0 \\ &= D_{n+1} 0^{n+1}, \end{aligned}$$

where the first equality comes from spliting 0^{n+1} into a single 0 and a string of n 0's, the second equality comes from our definition of E_n, the third equality comes from our assumption, the fourth equality is from the definition of D_n, and the final step is from combining a single 0 with a string of n 0's into a string of $n+1$ 0's. ■

Lemma 3.2.3 *For $n \geq 0$ and $k \geq 2$,*

$$D_0(D_0)^{k-1}(D_1)^{k-1}\cdots(D_n)^{k-1} = (E_n)^{k-1}(E_{n-1})^{k-1}\cdots(E_1)^{k-1}(E_0)^{k-1}E_0. \quad (3.11)$$

Proof. [Proof by induction] For the general case we will first prove, by induction on n, that

$$(D_n)^{k-1} = 0^n\ (E_n)^{k-2}(E_{n-1})^{k-1}\cdots(E_1)^{k-1}(E_0)^{k-1}E_0. \quad (3.12)$$

Equation (3.12) is true if $n = 1$. For $n+1$ we have

$$\begin{aligned} (D_{n+1})^{k-1} &= (D_{n+1})^{k-2} D_{n+1} \\ &= (D_{n+1})^{k-2} 0 (D_n)^k \\ &= (D_{n+1})^{k-2} 0 D_n (D_n)^{k-1} \\ &= (D_{n+1})^{k-2} 0 D_n 0^n\ (E_n)^{k-2}(E_{n-1})^{k-1}\cdots(E_1)^{k-1}(E_0)^{k-1}E_0 \\ &= (D_{n+1})^{k-2} 0^{n+1}\ E_n (E_n)^{k-2}(E_{n-1})^{k-1}\cdots(E_1)^{k-1}(E_0)^{k-1}E_0 \\ &= 0^{n+1}\ (E_{n+1})^{k-2}(E_n)^{k-1}(E_{n-1})^{k-1}\cdots(E_1)^{k-1}(E_0)^{k-1}E_0. \end{aligned}$$

Similarly, we can prove that

$$(E_n)^{k-1}(E_{n-1})^{k-1}\cdots(E_1)^{k-1}(E_0)^{k-1}E_0 0^{n+1} = E_{n+1}. \tag{3.13}$$

Now to our main proof. If $n = 0$, then $(D_0)^k = (E_0)^k = 1^k$. Assuming that it is true for some n, for $n+1$ we have

$$\begin{aligned}
& D_0(D_0)^{k-1}(D_1)^{k-1}\cdots(D_n)^{k-1}(D_{n+1})^{k-1} \\
= \; & (E_n)^{k-1}(E_{n-1})^{k-1}\cdots(E_1)^{k-1}(E_0)^{k-1}E_0(D_{n+1})^{k-1} \\
= \; & (E_n)^{k-1}(E_{n-1})^{k-1}\cdots(E_1)^{k-1}(E_0)^{k}0^{n+1}\,(E_{n+1})^{k-2}(E_n)^{k-1}\cdots(E_1)^{k-1}(E_0)^{k-1}E_0 \\
= \; & E_{n+1}(E_{n+1})^{k-2}(E_n)^{k-1}\cdots(E_1)^{k-1}(E_0)^{k-1}E_0 \\
= \; & (E_{n+1})^{k-1}(E_n)^{k-1}\cdots(E_1)^{k-1}(E_0)^{k}.
\end{aligned}$$

The first equality follows from the induction hypothesis, the second equality comes from (3.12) and the third equality comes from (3.13). ■

Lemma 3.2.4 *For $k \geq 2$,*

$$\mathcal{D}_{0,k} = D_0(D_0)^{k-1}(D_1)^{k-1}(D_2)^{k-1}(D_3)^{k-1}\cdots = E_\infty. \tag{3.14}$$

Proof. The first equality in (3.14) is implied immediately by the definition of $\mathcal{F}_{0,k}$; i.e., in $0D_nD_n\cdots$ the 0 is from the root (which is listed first in preorder) and $D_nD_n\cdots$ are the k subtrees of height n. Since we will add $k-1$ extra subtrees of height n, (one subtree has already been defined), we need to make sure to repeat D_n $k-1$ times.

The second equality comes from Lemma 3.2.3. Since E_n is a prefix of E_{n+1}, the expression E_∞ is well-defined. Hence $\mathcal{D}_0 = E_\infty$. ■

Using the notation for the q-binomial coefficients [3], we have $\begin{bmatrix} h \\ 1 \end{bmatrix}_k = \frac{k^h-1}{k-1} = 1 + k + \cdots + k^{h-1}$. In this thesis, the bottom term of the coefficient will always be one, so we will use the notation $[h]_k$ to represent $\begin{bmatrix} h \\ 1 \end{bmatrix}_k$. As above, we will drop the subscript k when there can be no confusion. Also, let $\#_1(x)$ represent the number of 1's in some string x.

Lemma 3.2.5 *For $n \geq 0$, we have $|D_n| = |E_n| = [n+1]$.*

Proof. By the definitions, we know $|D_n| = |E_n|$. We also know $D_0 = 1$, so $|D_0| = 1 = [0+1]$. Since $D_{n+1} = 0(D_n)^k$, we can show by induction

$$|D_{n+1}| = 1 + k[n+1] = 1 + k(1 + k + \cdots + k^n) = [n+2] \tag{3.15}$$

■

Corollary 3.2.6 *The numbers $a_{0,k}(n)$ satisfy the following recurrence relation for $0 \leq m < k^h$.*

$$a_0([h] + m) = k^{h-1} + a_0(m).$$

Proof. Since $\mathcal{D}_0 = E_h E_h \cdots 0 = E_{h-1}E_{h-1}E_{h-1}\cdots 0$ and $|E_{h-1}| = [h]$, the value of $d_0([h] + m) = d_0(m)$ for $1 \leq m \leq k^h - 1$. This is true due to the fact that E_{h-1} will be repeated $k-1$ times, and $(k-1)[h] = k^h - 1$.

Since we defined $d_0(0) = 0$ it also holds when $m = 0$. The number of 1's in E_{h-1} is $\#_1(E_{h-1}) = k^{h-1}$. Thus

$$\begin{aligned} a([h] + m) &= \sum_{p=0}^{[h]} d(p) + \sum_{p=0}^{m} d([h] + p) \\ &= \#_1(E_{h-1}) + \sum_{p=0}^{m} d(p) \\ &= k^{h-1} + a(m). \end{aligned}$$

■

Lemma 3.2.7 *For $s \geq 1$ and $k \geq 2$,*

$$a_{s,k}(n) = \begin{cases} a_0(n - sh) & \text{if } [h] + (s-1)h + 2 \leq n \leq [h+1] + (s-1)h, \\ k^{h-1} & \text{if } [h] + (s-1)h - s + 2 \leq n \leq [h] + (s-1)h + 1. \end{cases}$$

Proof. We show that the labels on the nodes in subforest h in $\mathcal{F}_{s,k}$ are exactly the values of n lying in the first range above. The number of regular nodes left of h can be found by adding the number of nodes in all of the subtrees. The first subtree has only one node. The remaining subtrees are $k$ary trees of height $j = 1, 2, \ldots, h-1$.

These k-ary trees each have $1 + k + \cdots + k^{j-1} = [j]$ nodes. By the construction of $\mathcal{F}_{s,k}$, we have $k-1$ of each subtree of height j (except, as previously mentioned, the extra tree of height 1). Summing the number of nodes in all of the subtrees gives us

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

The number of super-nodes that come before a super-node h is $s(h-1)$. Thus, the super-node h consists of the labels $[h]-h+1+(h-1)s+1 = [h]+(s-1)h-s+2$ to $[h]-h+1+(h-1)s+s = [h]+(s-1)h+1$. Therefore, the lowest label of a node in subforest h of our tree is $[h]+(s-1)h+1+1 = [h]+(s-1)h+2$ and the highest label is $[h]+(s-1)h+1+(k-1)[h] = [h+1]+(s-1)h$. ■

Corollary 3.2.8 $a_{1,k}(n) = a_{0,k}(n - \lfloor \log_k((n-1)(k-1)+1) \rfloor)$

Proof. If $s = 1$, we know from Lemma 3.2.7 that the supernodes of $\mathcal{F}_{1,k}$ will be numbered $[h]+1$. So, for example, in $\mathcal{F}_{1,4}$, the first supernode will be $[1]+1 = 2$, the second supernode will be $[2]+1 = 6$, and so on. Using this fact, we know that for some node n, we can determine which subforest h it is in by $\lfloor log_k((n-1)(k-1)+1) \rfloor$.

Taking $s = 1$ in Lemma 3.2.7 we obtain $a_1(n) = a_0(n-h)$ in the range $[h]+2 \leq n \leq [h+1]$. In that range $h = \lfloor log_k((n-1)(k-1)+1) \rfloor$. We need to check what happens when $n = [h]+1$. By the lemma $a_1([h]+1) = k^{h-1}$. In $\mathcal{F}_{0,k}$, the node $[h]+1-h$ is the rightmost node in subforest $h-1$, and thus $a_0([h]+1-h) = k^{h-1}$. ■

Theorem 3.2.9 *If $1 \leq n \leq s+1$, then $a_{s,k}(n) = 1$. If $n = s+i$ and $2 \leq i \leq k$ then $a_{s,k}(n) = i$. If $n > s+k$, then,*

If $[h]+(s-1)h-s+2 \leq n \leq [h]+(s-1)h+2$ then

$$a(n) = k^{h-1}$$

If $1 \leq m \leq [h-1]$ *then*

$$a([h] + (s-1)h + 2 + m) = k^{h-2} + a([h] + 1 + (s-1)h + m - [h-1] - s)$$

If $1 \leq m \leq k^{h-1} - 1$ *then*

$$a([h] + [h-1] + (s-1)h + 2 + m) = k^{h-2} + a([h] + (s-1)h + 2 + m)$$

If $1 \leq m \leq (k-2)[h]$ *then*

$$a(2[h] + (s-1)h + 1 + m) = k^{h-1} + a([h] + (s-1)h + 1 + m).$$

Proof. Let the node n be in the subforest h or in the super-node, call it y, that is the parent of the subforest h. Let $x_1, \ldots, x_{k-1}$ be the $k-1$ children of y which are not the left-most child of y, and denote the k subtrees of some x_i to be $T_{i,1}, T_{i,2}, \ldots, T_{i,k}$. We will establish the following recurrence relation.

$$a(n) = \begin{cases} k^{h-1} & \text{if } n = x_1 \text{ or } n \in y \\ k^{h-2} + a(n - [h-1] - s - 1) & \text{if } n \in T_{1,1} \\ k^{h-2} + a(n - [h-1]) & \text{if } n \in T_{1,i} \text{ where } i = 2, 3, ..., k \\ k^{h-1} + a(n - [h]) & \text{otherwise} \end{cases} \tag{3.16}$$

Clearly, if $n = x_1$ or $n \in y$, $a(n) = k^{h-1}$. Let T be the subtree whose root is the rightmost child of the leftmost child of y. In the second case above, we are relating the subtree $T_{1,1}$ to T. In this case we skip over k^{h-2} leaves and $[h-1] + s + 1$ nodes. In the third case above, we are relating the remaining subtree of $T_{1,i}$ to $T_{1,i-1}$. In this case we skip over k^{h-2} leaves and $[h-1]$ nodes. The final case relates the subtree rooted at x_i to the subtree rooted at x_{i-1}. In this case we are skipping over k^{h-1} leaves and $[h]$ nodes.

From the proof Lemma 3.2.7 we know that the labels in the super-node y are the labels in the range $[h] + (s-1)h - s + 2$ to $[h] + (s-1)h + 1$. Therefore, since x_1 is the next node that is labelled, $x_1 = [h] + (s-1)h + 2$. Thus, the leftmost child of x_1, which is the root of T_L, is $x_1 + 1 = [h] + (s-1)h + 3$. The second child of x_1 would be $[h] + [h-1] + (s-1)h + 3$, and x_2 would be $2[h] + (s-1)h + 2$. Thus, we know the range of values for each of the cases in (3.16), and we can see that the theorem statement is another way of writing (3.16). ■

Let $r_1, r_2, r_3, \ldots$ be the transition sequence of the k-ary reflected Gray codes. For the case where $k = 2$, this is commonly referred to as the "ruler function". For the purposes of this thesis we will be using the generalized ruler function, which is defined as the limit R_∞ where $R_1 = 1^{k-1}$ and $R_{n+1} = (R_n, n+1)^{k-1}, R_n$. R_∞ is well-defined because R_n is well-defined, and is the prefix of R_{n+1}. If the 0's are removed from the sequence $r_1 - 1, r_2 - 1, \ldots$ the ruler function is obtained again. The non-zero values in this sequence only occur in all of the positions divisible by k.

Lemma 3.2.10 *The generating function, $R(z)$ for the generalized ruler function is*

$$R(z) = \sum_{k \geq 1} r_k z^k = \sum_{n \geq 0} \frac{z^{k^n}}{1 - z^{k^n}}. \tag{3.17}$$

Proof. As mentioned above, by subtracting 1 from each r_k and removing the 0's, we obtain the ruler function again (see above), and the non-zero values in the sequence are in the positions divisible by k. This implies that if we again removed 1, but just from the positions divisible by k, and removed the 0's we would also obtain the ruler function again. Continuing this shows that the ruler function satisfies

$$R(z) = \frac{z}{1 - z} + R(z^k). \tag{3.18}$$

(3.17) is obtained by iterating this equation. ■

Lemma 3.2.11

$$\begin{aligned} \mathcal{D}_0 &= 1^k 0^{r_1} 1^k 0^{r_2} 1^k 0^{r_3} 1^k 0^{r_4} \cdots \\ &= 10^{r_1 - 1} 10^{r_2 - 1} 10^{r_3 - 1} 10^{r_4 - 1} \cdots . \end{aligned}$$

Proof. The ruler sequence is R_∞ where $R_1 = 1$ and $R_{n+1} = (R_n, n+1)^{k-1}, R_n$. Since $|R_n| = k^n - 1$, we have $r_{k^n + i} = r_i$ for $1 \leq i \leq k^{n+1} - k^n - 1$ and $r_{k^n} = n + 1$. We will show that

$$E_n = 1^k 0^{r_1} 1^k 0^{r_2} \cdots 1^k 0^{r_{k^n - 1}} \tag{3.19}$$

which will finish the proof of the first equality since $\mathcal{D}_0 = E_\infty$. For $n = 0$, we know that $E_1 = (E_0)^k 0$. By induction

$$\begin{aligned} E_{n+1} &= E_n E_n \cdots 0 \\ &= 1^k 0^{r_1} 1^k 0^{r_2} \cdots 1^k 0^{r_{k^n-1}} 1^k 0^{r_1} 1^k 0^{r_2} \cdots 1^k 0^{r_{k^n-1}} 0 \\ &= 1^k 0^{r_1} 1^k 0^{r_2} \cdots 1^k 0^{r_{k^n-1}} 1^k 0^{r_{k^n-1+1}} 1^k 0^{r_{k^n-1+2}} \cdots 1^k 0^{r_{k^n-1}} 1^k 0^{r_{k^n-1}} 0 \\ &= 1^k 0^{r_1} 1^k 0^{r_2} \cdots 1^k 0^{r_{k^n-1}} 1^k 0^{r_{k^n-1+1}} 1^k 0^{r_{k^n-1+2}} \cdots 1^k 0^{r_{k^n-1}} 1^k 0^{n+1}. \end{aligned}$$

The second equality comes from the fact mentioned above about the ruler function.

■

We can extend some of the previous results about $\mathcal{D}_{0,k}$ to $\mathcal{D}_{s,k}$. For proposition P the ntation $[\![P]\!]$ means 1 if P is true and 0 if P is false.

Lemma 3.2.12 *Let $t_j = r_j + s[\![n \text{ is a power of } k]\!]$. Then,*

$$\mathcal{D}_{s,k} = D_{0,k} 0^s (D_{0,k})^{k-1} 0^s (D_{1,k})^{k-1} 0^s (D_{2,k})^{k-1} \cdots \tag{3.20}$$

$$\mathcal{D}_{s,k} = 10^{t_1-1} 10^{t_2-1} 10^{t_3-1} 10^{t_4-1} \cdots \tag{3.21}$$

Proof. Equation (3.20) comes from our construction of $\mathcal{F}_{s,k}$. The 0^s terms represent where the supernodes go in our construction, and since $d(n) = 0$ when n is an internal node, we will have s 0's.

Equation (3.21) comes from the second equality in Lemma 3.2.11, and the fact that a new supernode will be added after we have seen a complete left subtree, which will have k^i leaf nodes, where i is an integer. Therefore, we need to add s 0's after every k^ith leaf node, where i is an integer. This gives us

$$\begin{aligned} \mathcal{D}_{s,k} &= 10^s 0^{r_1-1} 10^{r_2-1} \cdots 10^s 0^{r_k-1} 1 \cdots 10^s 0^{r_{k^2}-1} \cdots \\ &= 10^{s+r_1-1} 10^{r_2-1} \cdots 10^{s+r_k-1} 1 \cdots 10^{s+r_{k^2}-1} \cdots \\ &= 10^{t_1-1} 10^{t_2-1} 10^{t_3-1} 10^{t_4-1} \cdots \end{aligned}$$

■

Since the numbers $p_{s,k}(n)$ give the positions of the 1's in $\mathcal{D}_{s,k}$, we have the following corollary.

Corollary 3.2.13 *For all* $n \geq 1$,

$$p_{s,k}(n+1) - p_{s,k}(n) = r_n + s[\![n \text{ is a power of } k]\!].$$

3.2.2 Generating Functions

If $S = s(1)s(2)\cdots s(m)$ is a string then we use $S(z)$ to denote the ordinary generating function $S(z) = \sum s(i)z^i$. Let $\mathcal{A}_{s,k}(z)$ and $\mathcal{D}_{s,k}(z)$ denote the ordinary generating functions of the $a_{s,k}(n)$ and $d_{s,k}(n)$ sequences, respectively. Directly from the definitions we get the equation shown below:

$$\mathcal{A}_{s,k}(z) = \frac{\mathcal{D}_{s,k}(z)}{1-z}.$$

Since $\mathcal{A}(z)$ is determined by $\mathcal{D}(z)$ and $\mathcal{D}(z)$ is easier to work with, we first concentrate our attention on $\mathcal{D}(z)$.

Lemma 3.2.14

$$\begin{aligned} D_n(z) &= z^{n+1}(1+z^{[1]}+z^{2[1]}+\cdots+z^{(k-1)[1]})\cdots(1+z^{[n]}+\cdots+z^{(k-1)[n]}) \\ &= z^{n+1}\prod_{i=1}^{n}\sum_{j=0}^{k-1} z^{j[i]} = z^{n+1}\prod_{i=1}^{n}\frac{1-z^{k[i]}}{1-z^{[i]}} \end{aligned}$$

$$\begin{aligned} E_n(z) &= z(1+z^{[1]}+z^{2[1]}+\cdots+z^{(k-1)[1]})\cdots(1+z^{[n]}+\cdots+z^{(k-1)[n]}) \\ &= z\prod_{i=1}^{n}\sum_{j=0}^{k-1} z^{j[i]} = z\prod_{i=1}^{n}\frac{1-z^{k[i]}}{1-z^{[i]}}. \end{aligned}$$

Proof. From the recurrence relation $D_0 = 1$ and $D_{n+1} = 0(D_n)^k$ we obtain $D_0(z) = z$, and

$$\begin{aligned} D_{n+1}(z) &= zD_n(z) + z^{|0D_n|}D_n(z) + z^{|0(D_n)^2|}D_n(z) + \cdots + z^{|0(D_n)^{k-1}|}D_n(z) \\ &= zD_n(z) + z^{[n+1]+1}D_n(z) + z^{2[n+1]+1}D_n(z) + \cdots + z^{(k-1)[n+1]+1}D_n(z) \\ &= z(1 + z^{[n+1]} + z^{2[n+1]} + \cdots + z^{(k-1)[n+1]})D_n(z). \end{aligned}$$

This is true because we will have k repetitions of the string D_n; the first D_n will occur shifted to the right one place (to make room for the starting 0), the second D_n will

occur shifted to the right $1+|D_n|$ places (to make room for $0D_n$), and so forth. The second equality comes from the fact that $|0|=1$ and $|D_n|=[n+1]$ by Lemma 3.2.5, and the final equality comes from factoring $zD_n(z)$ out of the equation.

Similarly, $E_0(z)=z$ and $E_{n+1}(z)=(1+z^{[n+1]}+z^{2[n+1]}+\cdots+z^{(k-1)[n+1]})E_n(z)$. The results now follow by induction. ■

Corollary 3.2.15

$$\mathcal{D}_{0,k}(z)=z\prod_{i\geq 1}\sum_{j=0}^{k-1}z^{j[i]}=z\prod_{i\geq 1}\frac{1-z^{k[i]}}{1-z^{[i]}}.$$

Proof. This follows from Lemma 3.2.14 and the equation $\mathcal{D}_0=E_\infty$ from Lemma 3.2.4. ■

Theorem 3.2.16 *The generating function $\mathcal{D}_{s,k}(z)$ is equal to*

$$z\left(1+z^{s+k^0}\left(\frac{1-z^{(k-1)[1]}}{1-z^{[1]}}+z^{s+k^1}\frac{1-z^{k[1]}}{1-z^{[1]}}\left(\frac{1-z^{(k-1)[2]}}{1-z^{[2]}}+z^{s+k^2}\frac{1-z^{k[2]}}{1-z^{[2]}}(\cdots\right.\right.\right. \tag{3.22}$$

Proof. We need to translate the string $D_00^s(D_0)^{k-1}0^s(D_1)^{k-1}0^s(D_2)^{k-1}0^s\cdots$ from Lemma 3.2.12 into its generating function. Since

$$|D_00^s(D_0)^{k-1}0^s\cdots(D_{n-1})^{k-1}0^s|=s+1+\sum_{i=0}^{n-1}((k-1)[i+1]+s)=s+n(s-1)+[n+1]$$

we can write

$$\begin{aligned}\mathcal{D}_s(z) &= z+\sum_{n\geq 0}z^{s+n(s-1)+[n+1]}D_n(z)(1+z^{|D_n|}+\cdots+z^{(k-2)|D_n|})\\ &= z+\sum_{n\geq 0}\sum_{i=1}^{k-1}z^{s+n(s-1)+i[n+1]}D_n(z)\\ &= z+\sum_{n\geq 0}\sum_{i=1}^{k-1}z^{s+n(s-1)+i[n+1]+1}x_1x_2\cdots x_n\end{aligned}$$

Where $x_i = z(1 + z^{[i]} + \cdots + z^{(k-1)[i]}) = z(1 - z^{k[i]})/(1 - z^{[i]})$, so that $D_n(z) = zx_1x_2\cdots x_n$. Now, if we look at the summation, we have

$$\begin{aligned}
\mathcal{D}_s(z) &= z+(z(z^{s+[1]}+\cdots+z^{s+(k-1)[1]})) + (zx_1(z^{2s-1+[2]}+\cdots+z^{2s-1+(k-1)[2]})) + \cdots \\
&= z(1+(z^{s+[1]}+\cdots+z^{s+(k-1)[1]}) + (x_1(z^{2s-1+[2]}+\cdots+z^{2s-1+(k-1)[2]})) + \cdots \\
&= z(1+z^{s+[1]}((1+\cdots+z^{(k-2)[1]}) + (z^{s-1}x_1(z^{k[1]}+\cdots+z^{(k-2)[2]+k[1]})) + \cdots \\
&= z(1+z^{s+k^0}((1+\cdots+z^{(k-2)[1]}) + z^{s-1+k^1}x_1((1+\cdots+z^{(k-2)[2]})+\cdots \\
&= z(1+z^{s+k^0}([k-1]_{z^{[1]}} + z^{s-1+k^1}x_1([k-1]_{z^{[2]}} + z^{s-1+k^2}x_2([k-1]_{z^{[3]}}\cdots \\
&= z\left(1+z^{s+k^0}\left(\frac{1-z^{(k-1)[1]}}{1-z^{[1]}} + z^{s+k^1}\frac{1-z^{k[1]}}{1-z^{[1]}}\left(\frac{1-z^{(k-1)[2]}}{1-z^{[2]}}+z^{s+k^2}\frac{1-z^{k[2]}}{1-z^{[2]}}(\cdots\right.\right.\right.
\end{aligned}$$

■

Theorem 3.2.17 *For all $s \geq 1$ and $k \geq 2$,*

$$\mathcal{A}_{s,k}(z) = z\frac{1-z^s}{1-z}\sum_{n\geq 0}\prod_{i=1}^{n} z^s\frac{1-z^{k[i]}}{1-z^{[i]}}.$$

Proof. Call the right side of the equation $R(z)$, and multiply it by $(1 - z)$. We get

$$
\begin{aligned}
(1-z)R(z) &= z(1-z^s)\sum_{n\geq 0}\prod_{i=1}^{n} z^s \frac{1-z^{k[i]}}{1-z^{[i]}} \\
&= z\sum_{n\geq 0}\prod_{i=1}^{n} z^s \frac{1-z^{k[i]}}{1-z^{[i]}} - z^{s+1}\sum_{n\geq 0}\prod_{i=1}^{n} z^s \frac{1-z^{k[i]}}{1-z^{[i]}} \\
&= z + z\sum_{n\geq 1} z^{sn}\prod_{i=1}^{n} \frac{1-z^{k[i]}}{1-z^{[i]}} - z^{s+1}\sum_{n\geq 0}\prod_{i=1}^{n} z^s \frac{1-z^{k[i]}}{1-z^{[i]}} \\
&= z + z\sum_{n\geq 1} z^{sn}\prod_{i=1}^{n} \frac{1-z^{k[i]}}{1-z^{[i]}} - z^{s+1}\sum_{n\geq 1}\prod_{i=1}^{n-1} z^s \frac{1-z^{k[i]}}{1-z^{[i]}} \\
&= z + z\sum_{n\geq 1} z^{sn}\prod_{i=1}^{n} \frac{1-z^{k[i]}}{1-z^{[i]}} - z^{s+1}\sum_{n\geq 1} z^{s(n-1)}\prod_{i=1}^{n-1} \frac{1-z^{k[i]}}{1-z^{[i]}} \\
&= z + z\sum_{n\geq 1} z^{sn}\prod_{i=1}^{n} \frac{1-z^{k[i]}}{1-z^{[i]}} - z\sum_{n\geq 1} z^{sn}\prod_{i=1}^{n-1} \frac{1-z^{k[i]}}{1-z^{[i]}} \\
&= z + z\sum_{n\geq 1} z^{sn}\left(\prod_{i=1}^{n} \frac{1-z^{k[i]}}{1-z^{[i]}} - \prod_{i=1}^{n-1} \frac{1-z^{k[i]}}{1-z^{[i]}}\right) \\
&= z + z\sum_{n\geq 1} z^{sn}\left(\frac{1-z^{n[i]}}{1-z^{[i]}} - 1\right)\left(\prod_{i=1}^{n-1} \frac{1-z^{k[i]}}{1-z^{[i]}}\right) \\
&= z + \sum_{n\geq 1} z^{sn+1}\sum_{i=1}^{k-1} z^{i[n]}\prod_{i=1}^{n-1} \frac{1-z^{k[i]}}{1-z^{[i]}} \\
&= z + \sum_{n\geq 0} z^{sn+s+1}\sum_{i=1}^{k-1} z^{i[n+1]}\prod_{i=1}^{n} \frac{1-z^{k[i]}}{1-z^{[i]}}
\end{aligned}
$$

where the third equality comes from taking the $n = 0$ term out of the first summation. the fourth equality comes from substituting $n-1$ for n, the seventh equality comes from collecting like terms, the eighth comes from factoring, the ninth comes from the fact that $\frac{k^h-1}{k-1} = 1 + k + \cdots + k^{h-1}$, and the final equality comes from substituting $n+1$ for n. The remaining equation after the last step is equal to $\mathcal{D}_s(z)$, and therefore we have the generating function for $\mathcal{A}_s(z)$ as determined above. ■

We finish this section by finding a generating function for the $p_{s,k}(n)$ sequences.

Lemma 3.2.18 *For all $s \geq 0$ and $k \geq 2$,*

$$\sum_{n\geq 0} p_s(n)z^n = \frac{1}{1-z}\left(z + z\sum_{m\geq 0} z^{k^m}\left(s + \frac{1}{1-z^{k^m}}\right)\right).$$

Proof. Let $\mathcal{P}_s(z)$ denote the ordinary generating function of the numbers $p_s(n)$. Then

$$\begin{aligned}
\sum_{n\geq 1}(p_s(n+1) - p_s(n))z^n &= \sum_{n\geq 1} p_s(n+1)z^n - \sum_{n\geq 1} p_s(n)z^n \\
&= \sum_{n\geq 1} p_s(n+1)z^n - z\sum_{n\geq 1} p_s(n+1)z^n - z \\
&= (1-z)\sum_{n\geq 1} p_s(n+1)z^n - z \\
&= \frac{1}{z}\left(z(1-z)\sum_{n\geq 1} p_s(n+1)z^n - z^2\right) \\
&= \frac{1}{z}\left((1-z)\sum_{n\geq 2} p_s(n)z^n + (1-z)z - z\right) \\
&= \frac{1}{z}((1-z)\mathcal{P}_s(z) - z)
\end{aligned}$$

By Corollary 3.2.13 this expression is equal to

$$\sum_{n\geq 1}(r_n + s[\![n \text{ is a power of k}]\!])z^n = \sum_{m\geq 0}\left(sz^{k^m} + \frac{z^{k^m}}{1-z^{k^m}}\right) \tag{3.23}$$

where the equality follows from the equality for the generating function. This equation is obtained using Lemma 3.2.10 for the ruler function. Solving for $\mathcal{P}_s(z)$ finishes the proof. ∎

3.2.3 Compositions of an Integer

Jon Perry [12] gave experimental evidence that $a_{1,2}(n)$ counts the number of compositions of n of the form, for some integer m,

$$x_0 + x_1 + \cdots + x_m = n \text{ where } x_i \in \{1, 2^i\} \text{ for } i = 0, 1, \ldots, m \tag{3.24}$$

which I will represent with the notation $1 + \langle 1,2\rangle + \langle 1,4\rangle + \langle 1,8\rangle + \cdots$. Perry also gives examples of other combinatorial objects that have a one-to-one correspondence

with similar composition rules. Jackson and Ruskey [7] proved that a form of these compositions could be counted by the $a_{s,2}(n)$ numbers for all s. The following proof shows that a similar conclusion can be drawn for all $k \geq 2$.

Corollary 3.2.19 *For $s \geq 1, k \geq 2$, the number of compositions of n with specification*

$$\langle 1, 2, \ldots, s\rangle + \langle s, s+[1], \cdots, s+(k-1)[1]\rangle + \langle s, s+[2], \cdots, s+(k-1)[2]\rangle + \cdots$$

is $a_{s,k}(n)$.

Proof. This is clear from the generating function of $\mathcal{A}_{s,k}(z)$ given in Theorem 3.2.17 by observing that $z(1-z^s)/(1-z) = (z+z^2+\cdots+z^s)$. Substituting into our summation gives

$$(z+z^2+\cdots+z^s)(1+z^s(1+z^{[1]}+\cdots+z^{k[1]})+z^s(1+\cdots+z^{k[1]})z^s(1+\cdots+z^{k[2]})+\cdots$$

■

For example, when $s=2$ and $k=4$, the specification would be $\langle 1,2\rangle+\langle 2,3,4,5\rangle+\langle 2,7,12,17\rangle+\cdots$ and the $a_{2,4}(10)=5$ compositions are

$$\begin{aligned} 10 &= 1+2+7 \\ &= 1+3+2+2+2 \\ &= 1+5+2+2 \\ &= 2+2+2+2+2 \\ &= 2+4+2+2. \end{aligned}$$

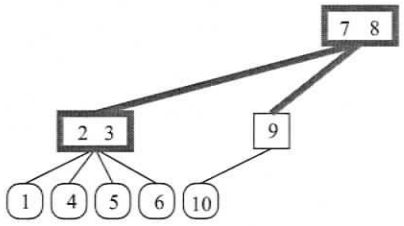

Figure 3.10. $\mathcal{T}_{2,4}(10)$

The tree $\mathcal{T}_{2,4}(10)$ is provided in Figure 3.10. It depicts the five bottom level nodes that are being counted by $a_{2,4}(10)$. Since the number of compositions is equal to the number of bottom level nodes, there would seem to be a one-to-one correspondence between a composition and a bottom level node. This is left as an open problem.

Chapter 4 — Summary and Conclusions

A new combinatorial interpretation has been given for some of the meta-Fibonacci sequences. Specifically, it deals with the number of nodes at the bottom level of a variation on k-ary trees. This extends work that was previously done by Jackson and Ruskey [7], who had similar results for binary trees. I was also able to provide information linking these same meta-Fibonacci sequences to a certain type of integer compositions, and provide generating functions for the sequences.

Work was also done on generating all possible compact codes of size n. While experimental evidence seems to show that the two algorithms given in this thesis run in constant amortized time, the proofs of this fact are not completed. Between the two algorithms, the one which is based on an algorithm by Norwood seems to be the most efficient, and also seems to be the easier one to prove out of the two. The second algorithm I discussed was a slight variation on the algorithm provided by Norwood.

There are several open problems that remain to be answered. Among these open problems is the task of finishing the proofs of whether or not the two given algorithms are actually CAT algorithms. In relation to the meta-Fibonacci sequence, an interesting open problem is what happens for $s < 0$. Could a similar interpretation be used for sequences of that form? Finally, there is also the correspondence between bottom level nodes in the tree $\mathcal{T}_{s,k}(n)$ and compositions as described in Corollary 3.2.19. Since the number of compositions is equal to the number of bottom level nodes, there should be a one-to-one correspondence between the two.

References

[1] N. Abramson, *Information Theory and Coding*, McGraw-Hill, 1963.

[2] J. Callaghan, Chow, and Tanny, *On the Behavior of a Family of Meta-Fibonacci Sequences*, SIAM J. Discrete Math, 18 (2005), 794-824.

[3] Peter J. Cameron, *Combinatorics: Topics, Techniques, Algorithms*, Cambridge University Press, 1994.

[4] R. Fano, *Transmission of Information*, Wiley, 1961.

[5] D. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, Basic Books, New York, 1979.

[6] D.A. Huffman, *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the Institute of Radio Engineers., Sept 1952, pp 1098-1101.

[7] B. Jackson and F. Ruskey, *Meta-Fibonacci Sequences, Binary Trees, and Extremal Compact Codes*, Electronic Journal of Combinatorics, 13 (2006), #R26, 13 pages.

[8] M. Khosravifard, M. Esmaeili, H. Saidi, and T. Aaron Gulliver, *A Tree Based Algorithm for Generating All Possible Binary Compact Codes with N Codewords*, IEICE Trans. Fundamentals, Vol.E86-A, No.10 October 2003.

[9] D.E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd edition, Addison-Wesley, 1997.

[10] D.E. Knuth, *Dynamic Huffman Coding*, Journal of Algorithms, 6(2):163-180, June 1985.

[11] E. Norwood, *The Number of Different Possible Compact Codes*, IEEE Transactions on Information Theory, 613-616, October 1967.

[12] J. Perry, *Symmetric Ferrar Diagrams*, website, `http://www.users.globalnet.co.uk/~perry/maths/symmetricferrars/symmetricferrars.htm`, May 2006

[13] D. Salomon, *Data Compression: The Complete Reference*, 3rd edition, Springer, 1998.

[14] S.M. Tanny, *A well-behaved cousin of the Hofstadter sequence*, Discrete Mathematics, 105 (1992) 227-239.

[15] J.S. Vitter, *ALGORITHM 673: dynamic Huffman coding*, ACM Transactions on Mathematical Software, 15(2):158-167, June 1989.