Amidakuji: Algorithms, Gray Codes and Equations for Listing Ladder Lotteries

by

Patrick Di Salvo

A Thesis

presented to

The University of Guelph

In partial fulfilment of requirements
for the degree of

MSc

in

Computer Science

Guelph, Ontario, Canada

ABSTRACT

Amidakuji: Algorithms, Gray Codes and Equations for Listing Ladder Lotteries

Patrick Di Salvo
University of Guelph, 2020

Advisor:
Dr. Joseph Sawada

We provide a Gray Code for listing ladder lotteries in which successive ladders differ by the addition/removal of a single bar or the relocation of a bar. Ladder lotteries are an abstract mathematical object which correspond to permutations. They are a network of vertical lines and horizontal bars, which induce transpositions on elements in a specific permutation. Ladder lotteries are of interest to the field of theoretical computer science because of their similarities to other mathematical objects such as primitive sorting networks. To list ladder lotteries, we define a function that calculates the location for any bar in the data structure in $O(1)$ time. We provide an $O(n^2)$ amortized algorithm for creating a specific ladder corresponding to a specific permutation. We provide an $O(1)$ amortized algorithm for listing all $n!$ canonical ladders by adding or removing a bar. We also provide a Gray code for listing all $n!$ ladders by $[0 \dots k \dots \binom{n}{2}]$ bars. Ladder lotteries are of interest to the field of theoretical computer science because of their similarities to other mathematical objects such as primitive sorting networks.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Amidakuji is a custom in Japan which allows for a pseudo-random assignment of children to prizes [28]. Usually done in Japanese schools, a teacher will draw n vertical lines, hereby known as *lines*, where n is the number of students in class. At the bottom of each line will be a unique prize. At the top of each line will be the name of one of the students. The teacher will then draw 0 or more horizontal lines, hereby known as *bars*, connecting two adjacent lines. No two endpoints of two bars can be touching. The more bars there are the more complicated (and fun) the Amidakuji is. Each student then traces their line, and whenever they encounter an end point of a bar along their line, they must cross the bar and continue going down the adjacent line. The student continues tracing down the lines and crossing bars until they get to the end of the ladder lottery. We call such a tracing for a given student a *path*. For example, the path of Ryu in the ladder lottery depicted in Figure 1.1 is highlighted in red.

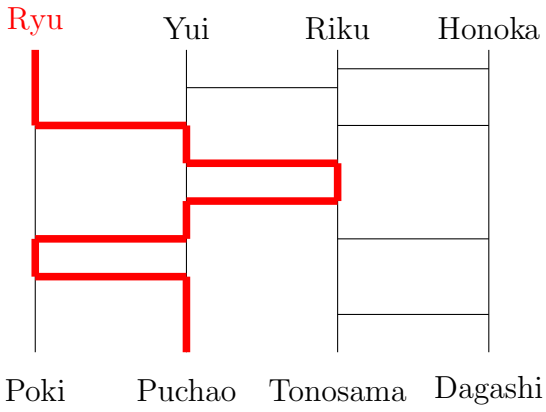


Figure 1.1: A ladder lottery where Ryu gets Puchao, Yui gets Dagashi, Riku gets Tonosama and Honoka gets Poki

The term "Amidakuji" has an interesting history. Amida is the Japanese name for Amithaba, the supreme Buddha of the Western Paradise. Amithaba was a Buddha from India and there was a cult based around him. The cult of Amida, otherwise known as Amidism, believed that by worshiping Amithaba, they would enter into his Western paradise [14]. Amidism began in India in the fourth century, made its way to China and Korea in the fifth century, and finally came to Japan in ninth century. Kuji is the Japanese word for lottery. Hence, the game was termed Amidakuji. In English, Amidakuji translates to ladder lottery.

The game Amidakuji began in Japan during the Muromachi period, which spanned from 1336 to 1573. During the Muromachi period, the game was played by having players draw their names at the top of the lines, and at the bottom of the lines were pieces of paper that had the amount the players were willing to bet. The pieces of paper were folded in the shape of Amithaba's halo [14].

Figure 1.2: Buddha Amida

An interesting property about a ladder lottery is that it is associated with a *permutation*. A *permutation* is a unique ordering of objects. For the purposes of this thesis, the objects of a permutation are $1, 2, \ldots, n$. A ladder lottery corresponds to a

unique permutation when:

1. The n elements of π are listed at the top of the ladder lottery in the order that they appear in π; one element per line of the ladder lottery.

2. At the bottom of the ladder lottery are the n elements of π in strictly ascending order. For each element x in π, x goes down its path and ends up in the bottom xth line.

1.1 Optimal Ladder Lotteries

Consider a permutation $\pi = (p_1, p_2, \ldots, p_n)$. A pair of elements, p_i and p_j, form an *inversion* if $p_i > p_j$ and $i < j$. For example, given $\pi = (4, 3, 5, 1, 2)$, its inversions are $(4, 3), (4, 2), (4, 1), (3, 2), (3, 1), (5, 1), (5, 2)$. A *transposition* is a swap of two elements in π. An *adjacent transposition* is defined as a swap of two adjacent elements in π. Given $\pi_1 = (4, 3, 5, 1, 2)$ and $\pi_2 = (3, 4, 5, 1, 2)$, they differ by the adjacent transposition of the elements $(4, 3)$. Let k denote the number of inversions for some permutation. An *optimal ladder lottery* is a special case of ladder lottery, in which the number of bars equals the number of inversions in π. An optimal ladder lottery sorts π in ascending order using exactly k bars by applying k adjacent transpositions on k inversions in π. When a permutation has k inversions, each bar transposes a single inversion in π exactly once [28]. For example, given $\pi = (4, 3, 5, 1, 2)$ an optimal ladder lottery associated with π would have seven bars; one for each inversion in π. For each bar, two elements in π that form an inversion cross the given bar. Once all elements have crossed their respective bars, π is sorted in ascending order. The number of bars in an optimal ladder lottery is the minimum number of bars in a ladder lottery required to sort π. To see an example of two ladder lotteries associated with $\pi = (4, 3, 5, 1, 2)$, one optimal and one non-optimal, please refer to Figure 1.3.

Let $OptL\{\pi\}$ be defined as the set of optimal ladder lotteries for a specific π. The first discussion of $OptL\{\pi\}$ is in the paper Efficient Enumeration of Ladder Lotteries

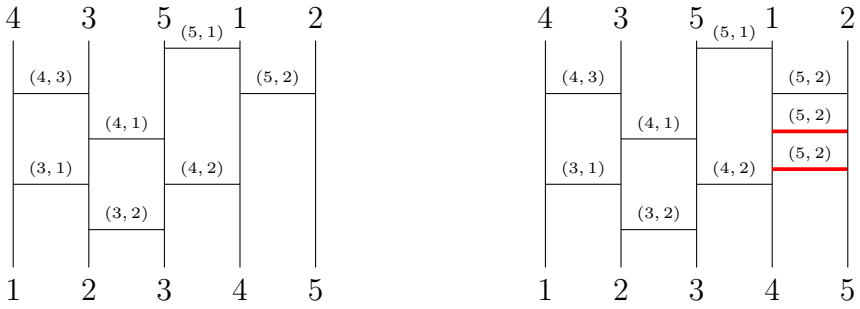


Figure 1.3: Two ladders for the permutation $(4, 3, 5, 1, 2)$. The left ladder is an optimal ladder and the right ladder is not. The bold bars in the right ladder are redundant, thus the right ladder is non optimal.

and its Application [28]. In this paper, the authors provide an algorithm which generates $OptL\{\pi\}$; the details of the algorithm are discussed in Chapter 2. To see $OptL\{(4, 3, 5, 1, 2)\}$ please refer to Figure 1.4. Given that there are $n!$ permutations of order n, each of them have their own $OptL\{\pi\}$. In Table A.1 found in the section A.1 of the Appendix, titled **Table of $OptL\{\pi\}$ for all π of order** 5, the number of ladders in each $OptL\{\pi\}$ of order 5 is presented. In general, $|OptL\{(n, n-1, \ldots, 2, 1)\}|$ is maximal. When $n = 5$, $|OptL\{(5, 4, 3, 2, 1)\} = 62|$.

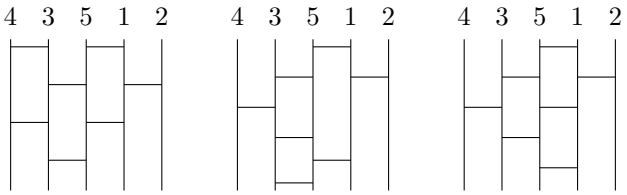


Figure 1.4: All the optimal ladders in $OptL\{(4, 3, 5, 1, 2)\}$

1.2 Combinatorial Generation

Our research on ladder lotteries pertains to research in *combinatorial generation*. Combinatorial generation is a subfield of theoretical computer science which lists all instances of combinatorial objects such as permutations, combinations, sets, subsets, graphs, and ladder lotteries [16]. Knuth describes a number of algorithms for listing such combinatorial objects in The Art of Computer Programming, volume 4 [12].

Gray codes are a special type of combinatorial generation in which a constant, minimal amount of change is required to get from one object to the next. For example, a single swap operation, a single rotation, or a single bit flip is used to get from one object to the next. The term "Gray code" comes from the Binary Reflected code given by Frank Gray [8]. In Figure, all 24 permutations of order 4 are listed in Gray code order using the Steinhaus-Johnson-Trotter Gray code.

1234	1243	1423	4123
4132	1432	1342	1324
3124	3142	3412	4312
4321	3421	3241	3214
2314	2341	2431	4231
4213	2413	2143	2134

Figure 1.5: 24 permutations of order 4 listed in Steinhaus-Johnson-Trotter Gray code order. Permutations are read left to right, top to bottom.

There are many known algorithms for listing permutations of order n. Throughout the course of this thesis, a number of such algorithms were researched [13, 3, 4, 9, 10, 6, 29, 22, 17, 1, 12]. Each of these algorithms will be reviewed in Chapter 2. To see all 24 ladders listed in lexicographic order please refer to Figure 1.6; ladders are read left to right, top to bottom. We note that the lexicographic ordering of ladders is not a Gray code order. For example, transitioning from ladder 6 to ladder 7 in Figure 1.6 requires four bars to be changed.

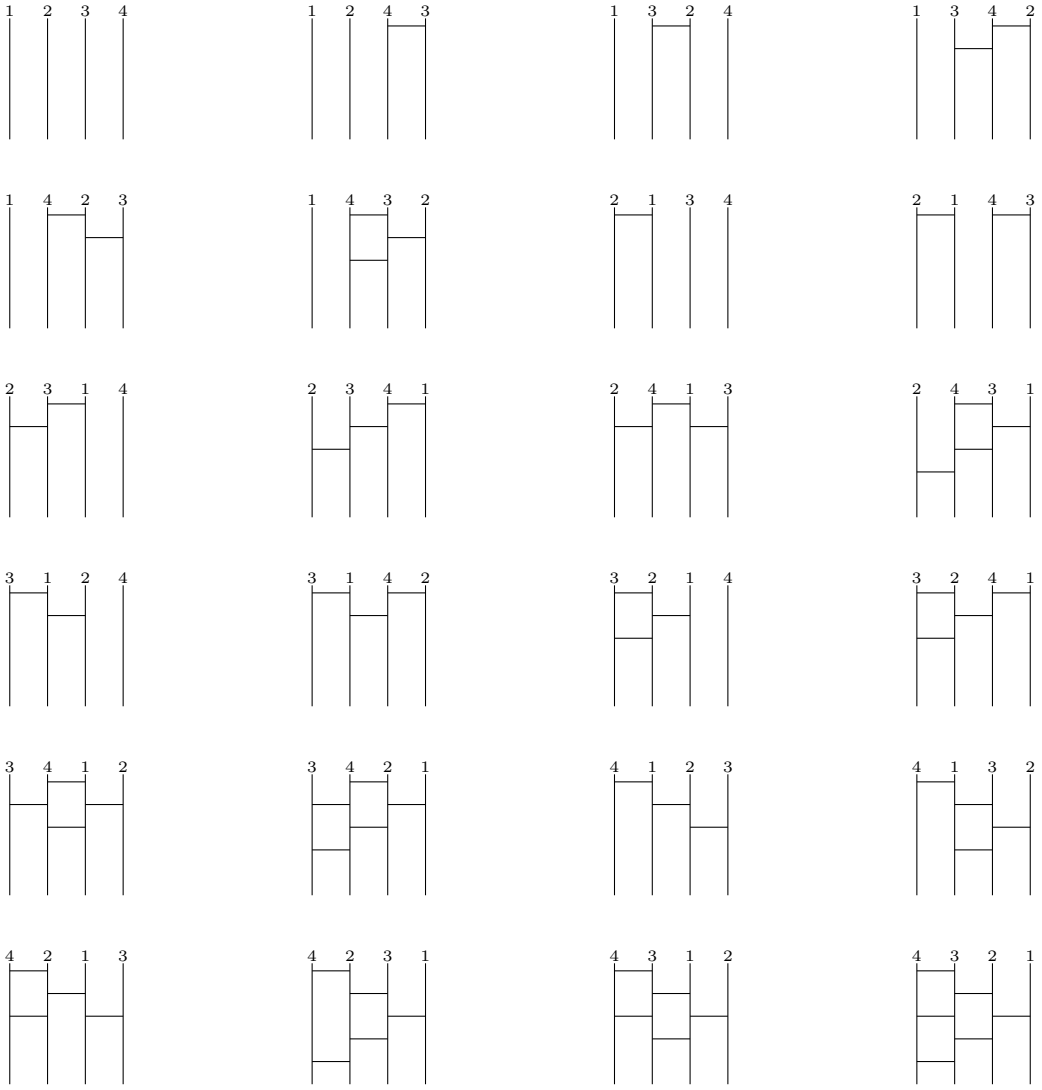


Figure 1.6: All 24 ladders in lexicographic order. Ladders are read left to right, top to bottom. Note how lex order is not a Gray code

1.3 Thesis Statement

We define *The Canonical Ladder Listing Problem* as follows: Given integers $[1 \ldots n]$, provide a listing of all $n!$ optimal ladders, one corresponding to each permutation of order n, in Gray code order whereby successive ladders differ by a *minimal amount of change*. First, we define *minimal change* as either the addition or removal of

a bar. Second, we define *minimal change* as moving a bar; moving a bar is the same as removing one bar and adding a new bar. We provide a canonical ladder representation of an optimal ladder from $OptL\{\pi\}$ in order to translate the ladder to a data structure. By doing so, we allow for a number of efficient solutions to solve The Canonical Ladder Listing Problem in the form of novel equations, algorithms, and Gray codes.

1.4 Contributions

In this thesis we provide the following contributions:

1. Definition of the canonical ladder.
2. Definition of the data structure for the canonical ladder.
3. Function to calculate the location of a bar in $O(1)$ time.
4. $O(n^2)$ algorithm to create the canonical ladder corresponding to a given permutation.
5. $O(1)$ amortized algorithm which lists $n!$ ladders where successive ladders differ by either a single addition of a bar or a single removal of a bar.
6. Gray code for listing all ladders with k bars for some arbitrary k value between $[0 \leq k \leq \binom{n}{2}]$.
7. Gray code for listing all $n!$ ladders by $[0 \ldots k \ldots \binom{n}{2}]$ bars. Successive ladders differ by the relocation of a bar.

1.5 Summary of Past Known Results

To the best of my knowledge, the first paper written on ladder lotteries is titled Efficient Enumeration of Ladder Lotteries and its Application written by Yamanaka,

Nakano, Matsui Uehara and Nakada. The paper was written in 2010 [28]. Since this paper, a number of problems related to ladder lotteries have been solved. In Table 1.1 the reader will find a table of solved problems related to ladder lotteries. In Chapter 2 a more comprehensive analysis of these solved problems will be provided.

Table of Known Results Related to Ladder Lotteries		
Name of Problem	Description	Source
Enumeration Problem	Generates $OptL\{\pi\}$	[28] 2010
Ladder Lottery Realization Problem	Determines time complexity for creating a ladder lottery given a multi-set of bars	[25] 2018
The Reconfiguration Problem	Determine the length of the path between two ladders in $OptL\{\pi\}$	[24] 2017
Enumeration Problem given n, k	Generates all ladders with n lines and k bars; includes non-optimal ladders	[26] 2014
The Coding Problem	Provides a binary string encoding for a ladder lottery	[23] 2012
Counting and Random Generation Problem	Provides a solution for counting ladder lotteries of a given type as well as randomly generating a ladder lottery	[27] 2017

Table 1.1: Table of known solutions for problems related to ladder lotteries

1.6 Overview of Thesis

This thesis is broken down into several sections. In Chapter 2, a literature review of ladder lotteries will be provided along with background information pertinent to this thesis. In Chapter 3, a definition of the canonical ladder is provided along with the algorithm CreateCanonical. In Chapter 4, an algorithm is provided which

lists $n!$ in Gray code order, by adding or removing a bar between successive ladders. In Chapter 5, an algorithm for listing all ladders with k bars in Gray code is order provided. Also in Chapter 5, we provide the algorithm for listing all $n!$ ladders by the number of bars, from $[0 \dots k \dots \binom{n}{2}]$ bars.

Chapter 2

Background and Literature Review

In this chapter we will provide a more comprehensive analysis of the existing research surrounding ladder lotteries. Let x be a ladder lottery or permutation. Throughout this thesis a number of algorithms are presented. Many of these algorithms use the following auxiliary functions:

1. Print(x): Prints x.
2. Swap(x,y): Swaps x, y.
3. Sort(x): Sorts x in ascending order.
4. Sorted(x): Returns true if x is sorted in ascending order, else returns false.
5. Max(x): Returns the maximum element in x.
6. Min(x): Returns the minimum element in x.

The study of ladder lotteries as mathematical objects began in 2010, in the paper Efficient Enumeration of Ladder Lotteries and its Application, written by Matsui, Nakada, Nakano Uehara and Yamanaka [28]. In this paper the authors present the first algorithm for generating $OptL\{\pi\}$ for some arbitrary permutation π. Since this paper emerged, there have been a number of other papers written about ladder lotteries. These papers include The Ladder Lottery Realization Problem, Optimal Reconfiguration of Optimal Ladder Lotteries, Efficient Enumeration of all Ladder Lotteries with K Bars, Coding Ladder Lotteries and Enumeration, Counting, and Random Generation of Ladder Lotteries. This thesis is also heavily influenced by

Efficient Enumeration of Ladder Lotteries and its Application. Throughout Chapter 2, we elaborate on the aforementioned papers pertaining to ladder lotteries.

2.1 Efficient Enumeration of Ladder Lotteries and its Application

In the Efficient Enumeration of Ladder Lotteries and its Application [28], written by Yamanaka, Nakano, Matsui, Uehara, and Nakada, the authors provide an algorithm for generating $OptL\{\pi\}$ for any π, in $\mathcal{O}(1)$ time per ladder. The authors refer to this algorithm as FindAllChildren.

FindAllChildren was used in this thesis to generate the sample data that aided in finding solutions for the Canonical Ladder Listing Problem. To see the tree structure generated by FindAllChildren for $OptL\{(4,3,2,1)\}$ please refer to Figure 2.1.

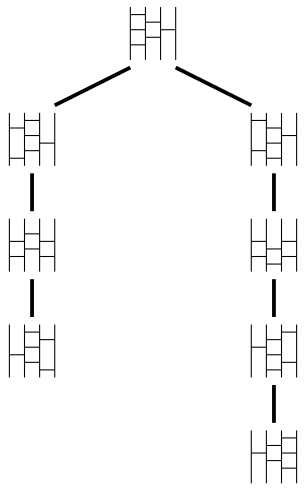


Figure 2.1: The tree structure of $OptL\{(4,3,2,1)\}$ generated by FindAllChildren

Let $x > y > z$ be any three elements in π. Let $pos(x) < pos(z)$ and let $pos(y) < pos(z)$. Let l be a ladder lottery in $OptL\{\pi\}$. Let (x, z) be a bar in l. Let (y, z) be a bar in l. We say the *clean level* of l is 1 plus the maximum value of x for which the following property holds: $\exists(x, z)$ below $(y, z) \in l$. Then, we say the clean level is $x + 1$. If no such x exists, we say the clean level of a ladder is 1. Yamanaka, Nakano, Matsui, Uehara, and Nakada show that the ladder with a clean level of 1 is unique in $OptL\{\pi\}$. This ladder is known as the *root ladder*. The authors do not provide an algorithm for creating the root ladder. Coincidentally, the algorithm CreateCanonical found in Chapter 3 also creates the root ladder for $OptL\{\pi\}$. Hence, we provide the algorithm for creating the root ladder. To see the root ladder of $OptL\{(4, 5, 6, 3, 1, 2)\}$ please refer to figure Figure 2.2, which we note has a clean level of 1.

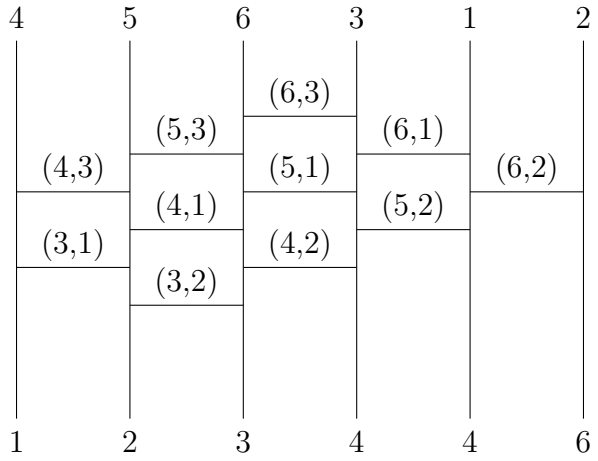


Figure 2.2: The root ladder for $OptL\{(4, 5, 6, 3, 1, 2)\}$. Notice how the clean level of this ladder is 1, thus making it the root ladder.

2.2 Ladder Lottery Realization

In the paper Ladder Lottery Realization [25], written by Horiyama, Uno, Wasa and Yamanaka, the authors provide a rather interesting puzzle in regards to ladder lotteries. The puzzle is known as the ladder lottery realization problem. In order to understand the problem, one must know what a *multi-set* is. A *multi-set* is a set in which an element may appear more than once. The exponent above the element indicates the number of times it appears in the set. For example, given the following multi-set, $\{3^2, 2^4, 5^1\}$ the element 3 appears twice in the set, the element 2 appears four times in the set and the element 5 appears once in the set. The ladder lottery realization puzzle asks, given an arbitrary starting permutation, π, and a multi-set of bars, is there a ladder lottery for π that uses every bar in the multi-set the number of times it appears in the multi-set. For an example of an affirmative solution to the ladder lottery realization problem, see Figure 2.3.

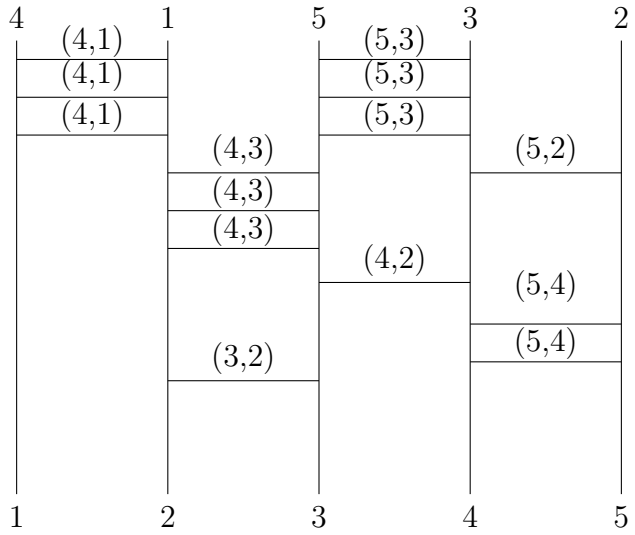


Figure 2.3: An affirmative solution to the Ladder Lottery Realization Problem given a starting permutation $(4, 1, 5, 3, 2)$ and the multi set of bars $\{(4, 1)^3, (4, 3)^3, (4, 2)^1, (5, 4)^2, (5, 3)^3, (5, 2)^1, (3, 2)^1\}$

The authors prove that the ladder lottery realization problem in NP-Hard by reducing the ladder lottery realization to the One-In-Three 3SAT problem, which has already been proven to be NP-Hard. The authors note that there are two cases in which the ladder lottery realization problem can be solved in polynomial time. These cases include the following. First, if every bar in the multi-set appears exactly once and every bar corresponds to an inversion, then an affirmative solution to the ladder lottery realization instance can be achieved in polynomial time. Second, if there is an inversion in the permutation and its bar appears in the multi-set an even number of times, then a negative solution to the ladder lottery realization instance can be achieved in polynomial time. This is because the elements that cross the bar will be uninverted when then be inverted again. Therefore π will not be sorted by the ladder.

2.3 Optimal Reconfiguration of Optimal Ladder Lotteries

In Optimal Reconfiguration of Optimal Ladder Lotteries [24], written by Horiyama, Wasa and Yamanaka, the authors provide a polynomial solution to the *minimal reconfiguration problem* which asks, given two ladders, L_i and L_m, what is the minimal number of swap operations to perform that will transition from L_i to L_m? A *reverse triple* is a relation between three bars, x, y, z in two arbitrary ladders, L_i, L_m, such that if x, y, x are right swapped in L_i, then they are left swapped in L_m or if they are left swapped in L_i then they are right swapped in L_m. Let an *improving triple* be defined as performing a right/left swapping three bars, x, y, z, in L_i such that the result of the swap removes a reverse triple between ladders L_i and L_m. The improving triple is a symmetric relation, therefore performing a right/left swapping of the x, y, z in L_m also results in the removal of a reverse triple between L_i and L_m.

The *minimal length reconfiguration sequence* is the minimal number of improving triples required to transition from L_i to L_m or L_m to L_i. Transitioning from L_i to L_m with the minimal length reconfiguration sequence is achieved by applying an

improving triple to each of the reverse triples between L_i and L_m. That is to say, the length of the reconfiguration sequence is equal to the number of improving triples required to remove all reverse triples between L_i and L_m.

The second contribution of this paper is that it provides a closed form formula for the upper bound for the minimal length reconfiguration sequence for any permutation of size n. That is to say, given some arbitrary π of order n, what is the maximum length of the minimal length reconfiguration sequence between two ladders in $OptL\{\pi\}$? The authors prove that there are two unique ladders in $OptL\{(n, n-1, \ldots, 1)\}$ that have the upper bound for the minimal length reconfiguration sequence. These ladders are the root ladder and *final ladder* which is defined as the unique ladder in $OptL\{\pi\}$ such that $\forall z < y < x : (y, z)$ is above $(x, z) \in l$. The length of the reconfiguration sequence between the root ladder and final ladder in $OptL\{(n, n-1, \ldots, 1)\}$ is $n\binom{(n-1)}{2}$.

2.4 Coding Ladder Lotteries

In the paper Coding Ladder Lotteries [23], written by Aiuchi, Yamanaka, Hirayama and Nishitani, the authors provide three methods to encode ladder lotteries as binary strings. Coding ladder lotteries as binary strings allows for a compact, computer readable, representation of them.

2.4.1 Route Based Encoding

The first method is termed *route based encoding method* in which each route of an element in the permutation has a binary encoding. Let $RC(l)$ be the route encoding for some arbitrary ladder in $OptL\{\pi\}$. For an example of the route encoding for the root ladder of $(3, 2, 5, 4, 1)$ refer to Figure 2.4. The number of bits needed for $RC(l)$ belongs to $\mathcal{O}(n^2)$.

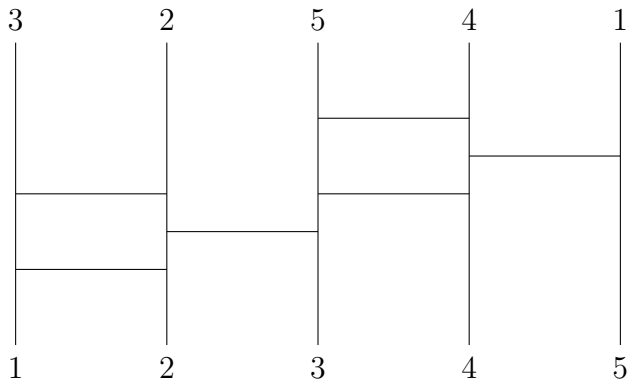


Figure 2.4: The route encoding for the given ladder lottery is 11<u>00</u>10<u>01</u>10<u>00</u>10<u>00</u>0000

2.4.2 Line Based Encoding

The second method is termed *line based encoding* which focuses on encoding the lines of the ladder lottery. Each line is represented as a sequence of endpoints of bars. Let l be an optimal ladder lottery with n lines and b bars, then for some arbitrary line, i, there are zero or more right/left endpoints of bars that come into contact with i. Let $LC(i)$ denote the line based encoding for line i. Let 1 denote a left end point that comes into contact with line i and let 0 denote a right end point that comes into contact with line i. Finally, append a 0 to line i to denote the end of the line. Then line i can be encoded, from top to bottom, as a sequence of $1s$ and $0s$ that terminates in a 0. Since each bar is encoded as two bits, and there are $n-1$ bits as terminating bits; one for each line in l, then the number of bits required is $n+2b-1$, where n is the number of lines and b is the number of bars. Encoding and decoding can be done in $\mathcal{O}(n+b)$ time. Clearly the line-based encoding trumps the route-based encoding in both time and space complexity.

2.4.3 Improved Line Based Encoding

Although the line-based encoding is better than the route based encoding, it can still be further optimized. The authors provide three improvements to the line-based

encoding.

1. The nth line has only right endpoints attached to it, therefore it actually does not need to be encoded. Right endpoints are denoted as 0 and left endpoints are encoded as 1, therefore the number of right endpoints for line n is equal to the number of $1s$ in $LC(i = n - 1)$.

2. Given two bars, x, y, let l_x denote the left endpoint of bar x, let l_y denote the left endpoint of bar y, let r_x denote the right end point of bar x and let r_y denote the right end point of bar y. Let line i be the line of l_x and l_y and let line $i + 1$ be the line of r_x and r_y. If there is not a 0 between the two $1s$ for l_x, l_y in LC_i, it is implied that there is at least one 1 between the two $0s$ for r_x, r_y on LC_{i+1}. Hence, one of the $1s$ in $LC(i + 1)$ can be omitted.

3. Improvement three is based off of saving some bits for right end points/$0s$ in $LC(i = n - 1)$. Since line n has no left end points, then there must be some right endpoints between any two consecutive bars connecting lines $n - 1$ and line n. Knowing this, then given two bars, x and y with l_x/l_y on line $n - 1$ and r_x/r_y on line n, there must be at least one bar, z, with its r_z between l_x and l_y on line $n - 1$. Thus, for every 1 in $LC(i = n - 1)$ except the last 1 a 0 must immediately proceed any 1. Since this 0 is implied, it can be removed from $LC(i = n - 1)$.

The line based encoding for the ladder in Figure 2.5 is 11$\underline{0}$10101110$\underline{0}$0010101$\underline{0}$000. The combination of all three improvements can be done independently.
By applying improvement one, we get 11$\underline{0}$10101110$\underline{0}$0010101$\underline{0}$. Notice how the last three $0s$ were removed because they represented $LC(i = n)$. By applying improvement two to improvement one we get 11$\underline{0}$1001110$\underline{0}$010010. Notice how the second, and eighth 1 were removed because they are implied by the successive $0s$. By applying improvement three to the result of improvement two we get 11$\underline{0}$1001110$\underline{0}$01010. Notice how the last 0 was removed from improvement two. This is because the 0 is implied in

$LC(i = n-1)$ due to the configuration between of bars connecting lines $n-1$ and line n. The improved line based encoding for the ladder in Figure 2.5 is 11$\underline{0}$10011$\underline{0}$001010.

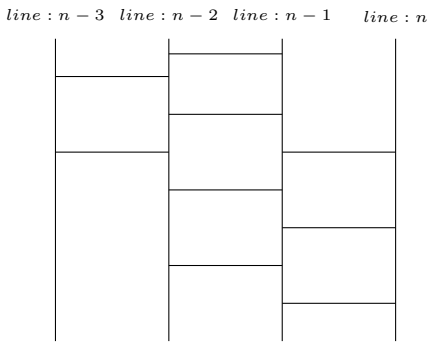


Figure 2.5: A ladder used to illustrate all three improvements. The improved line based encoding is 11$\underline{0}$10011$\underline{0}$001010

2.5 Enumeration, Counting, and Random Generation of Ladder Lotteries

In the paper, Enumeration, Counting, and Random Generation of Ladder Lotteries [27], written by Nakano and Yamanaka the authors consider the problem of enumeration, counting and random generation of ladder lotteries with n lines and b bars. It is important to note that the authors considered both optimal and non-optimal ladders for this paper. Nonetheless, the paper is still fruitful for its modelling of the problems and insights into ladder lotteries. The authors use the line-based encoding, $LC(l)$ for the representation of ladders that was discussed in the review of Coding Ladder Lotteries [23].

2.5.1 Enumeration

The authors denote a set of ladder lotteries with n lines and b bars as $S_{n,b}$. The problem is how to enumerate all the ladders in $S_{n,b}$. The authors use a *forest structure*

to model the problem. A *forest structure* is a set of trees such that each tree in the forest is disjoint union with every other tree in the forest. Consider $S_{n,b}$ to be a tree in a forest. That is to say, a union disjoint subset of all ladders with n lines and b bars. Then $F_{n,b}$, or the forest of all $S_{n,b}$, is the union of all disjoint trees of ladders with n lines and b bars.

2.5.2 Counting

The authors provide a method and algorithm to count all ladders with n lines and b bars. The counting algorithm works by dividing ladders into four types of sub-ladders. For sub-ladder, r, its type is a tuple $t(n, h, p, q)$ where n is the number of lines, h is the number of half bars, p is the number of unmatched end-points on line $n-1$ and q is the number of unmatched end-points on line n. From this type, the authors are able to count all ladders with n lines and b bars.

2.6 Permutations

Ladder lotteries and permutations are intricately related to each other. The research for The Canonical Ladder Listing Problem is highly influenced by permutation listing algorithms. Knuth describes a number of permutation listing algorithms in The Art of Computer Programming [13]. Since this book, many algorithms for enumerating $n!$ permutations have been created. During the research for The Canonical Ladder Listing Problem, twelve of these enumeration algorithms were investigated [13, 3, 4, 9, 10, 6, 29, 22, 17, 1, 12].

We define S_n as the set of all $n!$ permutations of order n. The first algorithm we looked at for listing S_n is the lexicographic algorithm, which orders all $n!$ permutations from smallest to largest. The lexicographic algorithm generates each permutation with an amortized time of $O(n^2 log(n))$ [4]. The second of which is the colexicographic algorithm which orders all $n!$ permutations from largest to smallest.

The colexicographic algorithm generates each permutation with an amortized time of $O(n^2log(n))$ [3]. The third of which is Zak's algorithm which lists S_n by reversing a suffix in one permutation to get to the next permutation. The time complexity of Zak's algorithm is amortized time of $O(n^2)$ [29]. The fourth of which is Heap's algorithm which is a decrease and conquer algorithm. The time complexity of Heap's algorithm is CAT, constant amortized time of $O(1)$ per permutation [9]. The fifth algorithm is the Steinhaus-Johnson-Trotter (SJT) algorithm which generates S_n by performing adjacent swap operations on the permutation resulting in the next permutation. Thus, each permutation in S_n differs from it predecessor by a single swap operation, making SJT a Gray code [10]. To go from any two successive permutations, the time complexity is $O(1)$ [10]. The sixth algorithm is the algorithm using star transpositions that always swaps the first element of the permutation with some other element. It was discovered by Gideon Ehrlich and is described as Algorithm E in Knuth's book [13]. The seventh algorithm is the derangement ordering in which no two consecutive permutations have any elements in the same position. It was first discussed by Savage in [17]. The eighth algorithm is the Single Track listing algorithm. Each column in the list of permutations is a cyclic shift of the first column [1]. The computation for each successive permutation is CAT [1]. The ninth algorithm is the Single Track Gray Code listing algorithm. The properties of the Single Track listing algorithm hold. Furthermore, any two successive permutations differing by at most two transpositions [1]. The tenth listing algorithm is found in Knuth's book. At each step, it either rotates the permutation to the left by one or swaps the first two elements [12]. The problem as to whether such a listing algorithm exists for all n is posed as an open problem in Knuth's book [12]. It was solved by Sawada and Williams in their paper A Hamiltonian Path for the Sigma-Tau Problem [18]. The eleventh algorithm is Corbett's algorithm which rotates a prefix of the first possible length in n,2,$n-1$,3,$n-2$,4, etc. [22]. To see the listings for all the aforementioned algorithms, refer to Figure 2.6 [16].

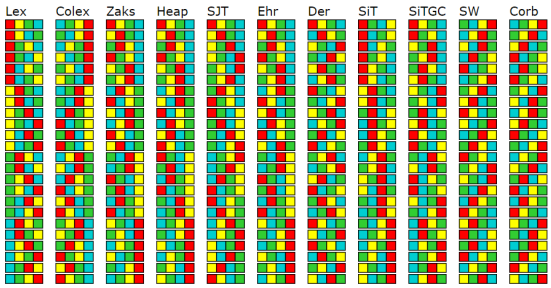


Figure 2.6: Eleven permutation listing algorithms

In Chapter 1, it was stated that a modification of the SJT algorithm was used to solve The Canonical Ladder Listing Problem. SJT generates each permutation in $O(1)$ per permutation and in Gray code order. In Chapter 4, the SJT algorithm is modified to create ladders instead of permutations while still maintaining the same efficiency and order. Below, we further examine the SJT algorithm.

2.6.1 Steinhaus-Johnson-Trotter Algorithm

The Steinhaus-Johnson-Trotter algorithm generates S_n by performing adjacent swap operations on the permutation resulting in the next permutation. Thus, each permutation in S_n differs from it predecessor by a single swap operation. Let an *even permutation* be defined as a permutation with an even number of inversions. Let an *odd permutation* be defined as a permutation with an odd number of inversions. If π of order $n-1$ is an even permutation, then the *nth* element is inserted into all positions of π of order $n-1$ in descending order. If π of order $n-1$ is an odd permutation, the *nth* element is inserted into all positions of π of order $n-1$ in ascending order [10]. For π of order 1 we have $\pi = (1)$. Since there are no inversions in $\pi = (1)$ it is even. Now insert 2 in all positions in $\pi = (1)$ in descending order. Thus we get $(1, 2)$ followed by $(2, 1)$. Since $(1, 2)$ is an even permutation, insert 3 into all positions in descending order resulting in $(1, 2, 3)$, $(1, 3, 2)$ and $(3, 1, 2)$. Since $(2, 1)$ is an odd permutation, insert 3 into all permutations in ascending order resulting in $(3, 2, 1)$, $(2, 3, 1)$ and $(2, 1, 3)$.

The Steinhaus-Johnson-Trotter algorithm is a Gray code, meaning that in order to transition between any two subsequent permutations in S_n, there is a minimal amount of constant change required. The algorithm simply swaps two elements in order to transition between permutations. Each recursive call creates a new permutation with the exception of the initial call to the function in which n recursive calls need to be made before the first permutation is printed. The amortized time to transition between permutations is $O(1)$.

2.7 Permutations By k Inversions

In the previous section we looked at permutation listings for all $n!$ permutations. In this section we look at listings for permutations by a given number of inversions. Let k be the number of inversions for a given permutation of order n. Let $\Pi_{n,k}$ indicate the number of permutations of order n with k inversions. We note that the $\Pi_{n,k}$ is counted by the triangle of Mahonian numbers [19]. Let $S_{n,k}$ denote all permutations of order n with k inversions.

2.7.1 Effler-Ruskey Algorithm

In the paper A CAT Algorithm for Generating Permutations with a Fixed Number of Inversions [6], written by Effler and Ruskey, the authors provide a constant amortized time algorithm for generating all permutations of order n with k inversions. The algorithm is also a *BEST* algorithm (backtracking ensures success at terminals), meaning that when the algorithm backtracks, the back-tracking leads to a successful result. We define the *empty permutation of order* n as an empty vector of length n. Let the initial conditions of the algorithm be the following. Let π be the empty permutation of order n. Let k be initialized to an integer between $[0 \ldots \binom{n}{2}]$. Let *list* be initialized to the list of n elements in strictly ascending order. Let i be the index of element x in *list*, using one indexing. $n - i$ indicates the number of inversions formed with x

Algorithm 1 Generate all permutations with k inversions

```
1: function KInversions(π, n, k, list)
2:     if n = 0 and k = 0 then
3:         Print(π)
4:     else
5:         for i from length of list to 1 do
6:             x ← list_i
7:             if n − i ≤ k ≤ C(n−1, k) + (n − i) then
8:                 p_n ← x
9:                 remove x from list
10:                k ← k − (n − i)
11:                KInversions(π, n − 1, k, list)
12:                insert x in list at correct position.
```

when inserting x into position n in π. The algorithm works as follows. Going right to left in the $list$, each element x is inserted into p_n only if inserting x into position n in π does not exceed the current number of inversions, k. I.e. only if $k - (n - i) \geq 0$. If x is inserted into π at position n then x is removed from $list$, a recursive call is made such that n is reduced by 1 and k is reduced by $(n - i)$. When the function returns from its recursive call, element x is placed back into the list at its original position. To see the algorithm please refer to Algorithm 1.

Below is an example of how the algorithm creates one permutation of order 4 with 2 inversions.

1. Initial Call: KInversions(π=(_,_,_,_),$n = 4$,$k = 2$,$list = [1, 2, 3, 4]$):
   given π=(_,_,_,_), $n = 4$, $k = 2$, $list = [1, 2, 3, 4]$, $i = 2$ and $x = 3$ then inserting x into position $n = 4$ would form $4 - 3 = 1$ inversion(s). Specifically, the inversion $(4, 3)$. Thus, k would be reduced by 1 in the recursive call and n would be reduced by 1 in the recursive call.

2. First recursive call: KInversions(π=(_,_,_,$\underline{3}$),$n = 3$,$k = 1$,$list = [1, 2, 4]$):
   given π=(_,_,_,$\underline{3}$),$n = 3$,$k = 1$,$list = [1, 2, 4]$,$i = 3$ and $x = 4$ then inserting x into position 3 would form $3 - 3 = 0$ inversions. Thus, k would be reduced by 0 in the recursive call and n would be reduced by 1 in the recursive call.

3. Second recursive call: KInversions(π=(_,_,$\underline{4}$,$\underline{3}$),$n = 2$,$k = 1$,$list = [1, 2]$):
   given π=(_,_,$\underline{4}$,$\underline{3}$),$n = 2$,$k = 1$,$list = [1, 2]$,$i = 1$ and $x = 1$ then inserting x into position 1 would form $2 - 1 = 1$ inversion. Thus, k would be reduced by 1 to 0 in the recursive call and n would be reduced by 1 in the recursive call.

4. Third recursive call: KInversions(π=(_,$\underline{1}$,$\underline{4}$,$\underline{3}$),$n = 1$,$k = 0$,$list = [2]$):
   given π=(_,$\underline{1}$,$\underline{4}$,$\underline{3}$), $n = 1$, $k = 0$, $list = [2]$ $i = 1$ and $x = 2$ then inserting x into position 1 would form $1 - 1 = 0$ inversions. Thus, k would be reduced by 0 and n would be reduced by 1.

5. Fourth recursive call: KInversions(π=($\underline{2}$,$\underline{1}$,$\underline{4}$,$\underline{3}$),$n = 0$,$k = 0$,$list = []$):
   Given $n = 0$ and $k = 0$ the algorithm Prints(π) and returns.

2.7.2 Walsh

Despite the efficiency of Effler-Ruskey's algorithm, we note that the permutations are not listed in Gray code order. Therefore, we also look at the paper, Loop Free Sequencing of Bounded Integer Compositions, written by Walsh [21]. Walsh provides an algorithm for listing $S_{n,k}$ in Gray code order in 0(1) time per permutation. Let an *n part composition of a non negative integer k* is an ordered n-tuple, $(g_1, g_2 \ldots, g_n)$, whose sum adds to k. Such a composition is said to be $(m_1, m_2, \ldots m_n)$ bounded when $0 \leq g_i \leq m_i$. Let π be of order $n+1$, then we say g is an n-tuple from $g_1 \ldots g_n$. We say m is a bound on g from $m_1 \ldots m_n$ such that each $m_i = min(n + 1 - i, k)$ where k is the target number of inversions. Each g_i is bounded by $0 \leq g_i \leq m_i$. Let g_i represent the number of inversions formed by placing an element from $[1 \ldots n+1] \in \pi$ at index i in π. We note that placing 1 and index 1 forms 0 inversions and we note that placing $n + 1$ at index 1 forms n inversions. g is known as the inversion vector of π, bounded by m.

Walsh lists $S_{n,k}$ by incrementing some g_i by 1 and decrementing some g_j by 1. Let x and y be positive integers indicating offsets from an index. Incrementing

composition	permutation
4 1 0 0	5 2 1 3 4
3 2 0 0	4 3 1 2 5
2 3 0 0	3 5 1 2 4
1 3 1 0	2 5 3 1 4
2 2 1 0	3 4 2 1 5
3 1 1 0	4 2 3 1 5
4 0 1 0	5 1 3 2 4
3 0 2 0	4 1 5 2 3
2 1 2 0	3 2 5 1 4
1 2 2 0	2 4 5 1 3
0 3 2 0	1 5 4 2 3
0 2 2 1	1 4 5 3 2
1 1 2 1	2 3 5 4 1
2 0 2 1	3 1 5 4 2
3 0 1 1	4 1 3 5 2
2 1 1 1	3 2 4 5 1
1 2 1 1	2 4 3 5 1
0 3 1 1	1 5 3 4 2
1 3 0 1	2 5 1 4 3
2 2 0 1	3 4 1 5 2
3 1 0 1	4 2 1 5 3
4 0 0 1	5 1 2 4 3

Figure 2.7: Walsh's Gray code ordering of inversion vectors and corresponding permutations

g_i by 1 induces a transposition in π with p_i and some element, p_{i+x} in the suffix $p_{i+1} \dots p_{n+1}$, such that $p_i < p_{i+x} \wedge p_{i+x} < \forall p_{i+y} > p_i$. Decrementing g_j by 1 induces a transposition in π with p_j and some element, p_{j+x} in the suffix $p_{j+1} \dots p_{n+1}$, such that $p_j > p_{j+x} \wedge p_{j+x} > \forall p_{j+y} < p_j$. We call i the *first pivot* j the *second pivot*, indicating that the increment occurs before the decrement; each pivot is found in $O(1)$ time. In Figure 2.7 [21], all compositions, and permutations of order 5 with 5 inversions, are listed in the Gray code order found in Walsh's paper. In Chapter 5, by borrowing from Walsh's paper, we derive a Gray code for listing all ladders with n lines and k bars.

2.8 Sorting Networks

Let a *wire* be a horizontal line. Let a *comparator* be a vertical line connecting two wires. A *sorting network* is a device consisting of $[1 \dots n]$ wires and $[0 \dots m]$ comparators such that the sorting network sorts a permutation of n elements into ascending order. The n elements are first listed to the left of each wire in the network. The

elements travel across their respective wires at the same time. When a pair of elements, traveling through a pair of wires, encounter a comparator, the comparator swaps the elements if and only if the top wire's element is greater than the bottom wire's element. A sorting network with n wires and m comparators that can sort any permutation of order n is a *complete sorting network*. To see a complete sorting network for $n = 4$ please refer to Figure 2.8.

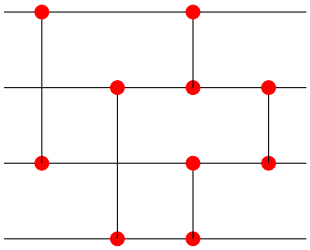


Figure 2.8: Complete Sorting Network for $n = 4$.

Sorting networks were first studied in 1954 by Armstrong, Nelson and O'Connor [13]. Donald Knuth describes how the comparators for binary integers can be implemented as simple, three-state electronic devices [13]. Batcher, in 1968, suggested using them to construct switching networks for computer hardware, replacing both buses and the faster, but more expensive, crossbar switches [2]. Currently, sorting networks are implemented in graphical processing units, GPUs, for faster sorting methods than traditional CPU sorting methods [15].

Let a *minimum sorting network* be defined as a sorting network such that for any arbitrary comparator, c, on wire i, c connects to line $i+1$ or $i-1$. Furthermore, the number of comparators in a minimum sorting network is equal to the number of inversions in π. Clearly, there is a bijection from comparators in a minimum sorting network to the bars in an optimal ladder lottery, and there is a bijection from the wires in a minimum sorting network and the lines in a ladder lottery.

Number of minimum sorting networks/$\lvert OptL\{Rev(\pi)\}\rvert$	
n	Count
1	1
2	1
3	2
4	8
5	62
6	908
7	24698
8	1232944
9	112018190
10	18410581880
11	5449192389984
12	2894710651370536
13	2752596959306389652
14	4675651520558571537540
15	1416380899558002221878639O

Table 2.1: Number of minimum sorting networks and $|OptL\{Rev(\pi)\}|$

2.8.1 The Integer Sequence Relating to the Reverse Permutation

Let $\pi = (n, n-1, \ldots 2, 1)$ refer to the reverse permutation order n. There is an integer sequence that counts the number of minimum sorting networks for π. This integer sequence also counts $OptL\{(n, n-1, \ldots 2, 1)\}$. This sequence grows very quickly, therefore $n = 15$ is the largest value this integer sequence has been calculated for. To refer to the table for this sequence please refer to Table 2.1 [20].

Dumitrescu and Mandal, in their paper New Lower Bounds For The Number of Pseudoline Arrangements [5] published in 2018, provide the current best known lower bound for this sequence as $bn \geq cn^2 - O(nlogn)$ for some constant $c > 0.2083$. In particular, $bn \geq 0.2083n^2$ for large values of n. Where bn is the bound for a given value n. In the paper, Coding and Counting Arrangements of Pseudolines [7] by Felsner and Valtr, written in 2011, the authors demonstrate the best known upper

bound for this sequence is $bn \leq 2^{0.657n^2}$.

Seeing as there is yet to be a closed form solution for this sequence, new values of n are counted by a variety of algorithms. In the paper, Efficient Enumeration of all Ladder Lotteries and its Application [28], the authors were the first to calculate the sequence for $n = 11$ with the algorithm FindAllChildren. In the paper, Counting Primitive Sorting Networks by πDDs [11], written by Kawahara, Minato, Saitoh and Yoshinaka, the authors were the first to calculate for $n = 13$ with a data structure they have termed πDD.

Chapter 3

The Canonical Ladder

In this chapter we describe a canonical ladder for each $OptL\{\pi\}$. We then provide a number of equations to calculate the location of the bar in the canonical ladder; depending on the configuration of π, we use different equations for calculating the location of the bar. We then define the data structure used to represent the canonical ladder. Using the data structure, we provide an algorithm for creating the canonical ladder for any π_n. Lastly, we describe the listing problem in relation to the canonical ladder. To see the canonical ladder for $(3, 5, 4, 1, 2)$, refer to Figure 3.1.

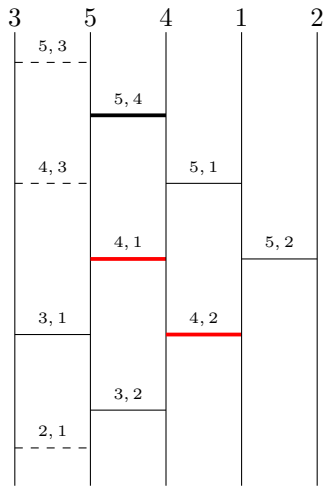


Figure 3.1: The canonical ladder for $(3, 5, 4, 1, 2)$. Bars $(5, 4), (4, 1), (4, 2)$ are part of the route of 4, but only the red bars are associated with the route of 4.

3.1 The Canonical Ladder in Detail

In this section, we fully define the canonical ladder corresponding to π. We first define the associated terminology in order to provide a comprehensive definition of the canonical ladder. Let the *route* of an element be the sequence of bars the element travels along in order to reach its final position in the sorted permutation. The bars are read from top to bottom. In Figure 3.1, the bars $(5,4),(4,1)$ and $(4,2)$ compose the route of 4. For every bar, two elements cross the bar, therefore *bar association* is the association of a bar with the greater of the two elements that cross it. For example, in Figure 3.1, element 4 has the bars $(5,4),(4,1)$ and $(4,2)$ in its route, however only bars $(4,1)$ and $(4,2)$ are associated with element 4. Let the *absence of a bar* corresponding to two uninverted elements in π, $x > y$, be defined as a dashed, horizontal line spanning across a column in the ladder. We say *absence of a bar association* is the association of the absence of a bar with the greater of the two elements that form it. For example,$(5,3)$ are uninverted in $(3,5,4,1,2)$, therefore the absence of the bar is associated with 5. For each element $2 \leq x \leq n \in \pi$, we define the *associated diagonal of* x as a diagonal with $x-1$ row and column coordinates, having one endpoint at $(2n-2x+1,1)$ and the other endpoint at $(2n-x-1,x-1)$. At each row and column in the associated diagonal of x, either a bar associated with x or the absence of a bar associated with x exists. For an example of the associated diagonals for the ladder corresponding to $(3,4,2,1,5)$ refer to Figure 3.2.

With the definition of the associated diagonal, we are now able to fully define the *canonical ladder* as the ladder such that for each element $2 \leq x \leq n$, each bar associated with x exists along the associated diagonal of x. We note that the two aforementioned figures, Figure 3.1 and Figure 3.2, are the canonical ladders corresponding to each of their respective permutations. We use $CL(\pi_n)$ as shorthand notation for 'the canonical ladder corresponding to a permutation of order n'.

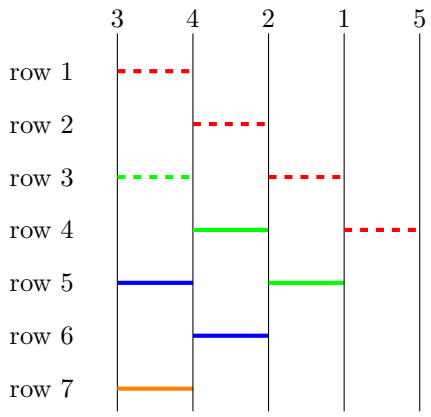


Figure 3.2: The associated diagonal of 5 is in red, the associated diagonal of 4 is in green, the associated diagonal of 3 is in blue, and the associated diagonal of 2 is in orange.

3.2 Locating the bar for a given inversion in an arbitrary $CL(\pi_n)$

In this section, we provide an equation for locating the bar in the canonical ladder associated with a given inversion, in an arbitrary permutation of order n. As previously stated, every bar associated with some element x exists along the associated diagonal of x within the canonical ladder. We use Equation 3.1, to determine the row and column for the bar corresponding to two elements, $x > y$, along the associated diagonal of x. Let π_n be an arbitrary permutation of order n. Let $pos(x)$ be the index of $x \in \pi_n$. To calculate the row and column of the bar, or absence of a bar, for a pair of elements $x > y$, we first map x to a new index termed $pos'(x)$. Let $pos'(x)$ equal $pos(x)$ minus the number of elements greater than x and to the left of x in π. For example, given $\pi_n = (3, 2, 7, 4, 8, 5, 6, 9, 1)$, suppose $x = 6$, then $pos'(6) = pos(6) - 2 = 5$. Once we have calculated $pos'(x)$ we use a similar method to calculate $pos'(y)$. $pos'(y)$ equals $pos(y)$ minus the number of elements greater than y and to the left of y excluding x. For example, suppose x is 7 and y is 5, then $pos'(y)$ is $pos(y) - 1 = 5$. Once we have $pos'(x)$ and $pos'(y)$, we use Equation 3.1 to return the row and column for the bar, or absence of the bar, along the associated diagonal

of x.

$$\text{(row,column)}= \begin{cases} ((2n-x-1)-(x-pos'(y))+1, (x-1)-(x-pos'(y))+1) & \text{if } pos'(x) > pos'(y) \\ ((2n-x-1)-(x-pos'(y)), (x-1)-(x-pos'(y)) & \text{if } pos'(y) > pos'(x) \end{cases} \tag{3.1}$$

Given the inversion $(7,5) \in (3,2,7,4,8,5,6,9,1)$, we calculate the row and column of the bar $(7,5)$ using the second case. We get $row = (18-7-1)-(7-5) = 8$ and $col = 6-(7-5) = 4$. To calculate $pos'(x)$ and $pos'(y)$, is linear amortized time. Once $pos'(x)$ and $pos'(y)$ are calculated, returns the location in $O(1)$ time. We can use Equation 3.1 to look up the location of any bar for a given π_n. In Figure 3.3 we see $CL(3,2,7,4,8,5,6,9,1)$, where each row and column is calculated using Equation 3.1.

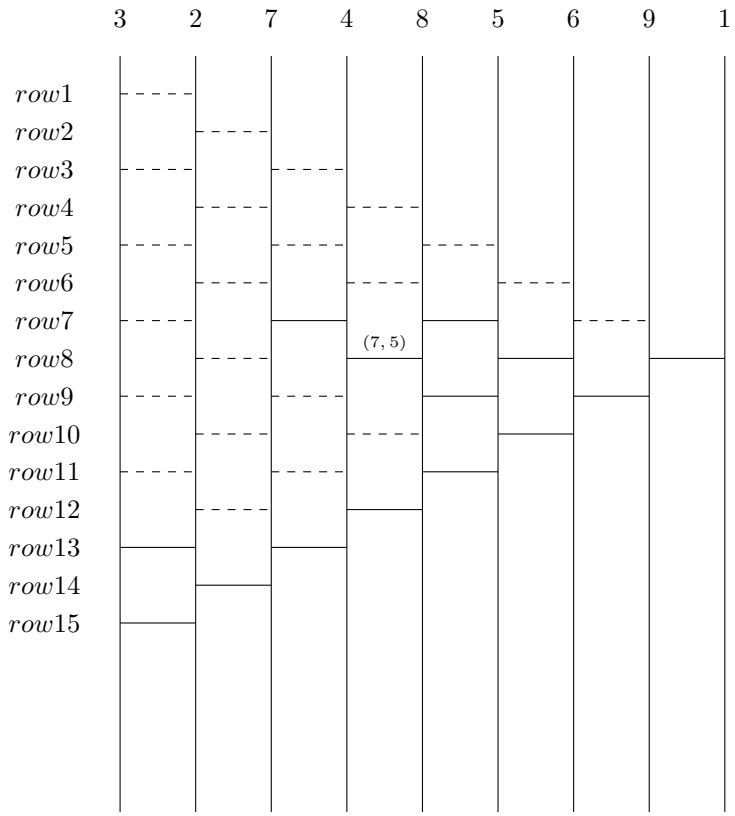


Figure 3.3: Canonical ladder such that each bar or absence of a bar is calculated using the Equation 3.1.

3.3 Locating the bar for a given inversion in $CL(n, n-1, \ldots 1)$

In the previous section, we looked up a bar for a given inversion for some arbitrary permutation in $O(n)$ time. Clearly this is less than ideal. However, if π is an arbitrary permutation of order n, we require linear time to calculate the location of a bar. In this section, we calculate the location of a bar corresponding to an inversion for $CL(n, n-1, \ldots, 1)$ in $O(1)$ time. Let $\pi = (n, n-1, n-2, ..., 1)$, then each associated diagonal in $CL(\pi)$ is fully occupied by bars. Let x, y be a pair of elements in π such that $x > y$ and $2 \leq x \leq n$. We know there exists a bar, (x, y), in $CL(n, n-1, \ldots 1)$ along the associated diagonal of x. We compute the row and column of a bar (x, y)

in $CL(n, n-1, \ldots 1)$ in $O(1)$ time using Equation 3.2.

$$(row, column) = (2n - 2x + (x - y), (x - y)) \tag{3.2}$$

For example, given $\pi = (6, 5, 4, 3, 2, 1)$, bar $(5, 2)$ is located at row 5; note that $5 = (2n-2x)+(x-y)$. Bar $(5, 2)$ is also located at column 3; note that $3 = x-y$. To see the canonical ladder for $(6, 5, 4, 3, 2, 1)$ with the corresponding bars, please refer to Figure 3.4. We use Equation 3.2 to locate a bar when we know $\pi = (n, n-1, \ldots, 2, 1)$.

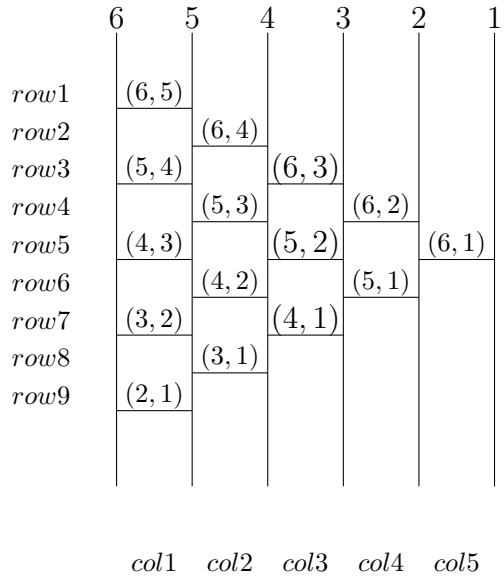


Figure 3.4: Canonical ladder for $(6, 5, 4, 3, 2, 1)$. The location of each bar can be calculated using the formula $(2n - 2x + (x - y), (x - y))$

We use Equation 3.2 when possible because the calculation time is $O(1)$ rather than $O(n)$.

3.4 Algorithm: Create Canonical

In this section we provide the data structure used to represent the canonical ladder in code. We then provide the algorithm for creating the canonical ladder corresponding to any permutation of order n. To create the canonical ladder we must represent

the canonical ladder in code. We use Theorem 3.4.1 and Corollary 3.4.2 to prove the veracity of the representation of the canonical ladder.

Theorem 3.4.1 *The number of rows required for the canonical ladder corresponding to the descending permutation of order n is $2(n-1)-1$.*

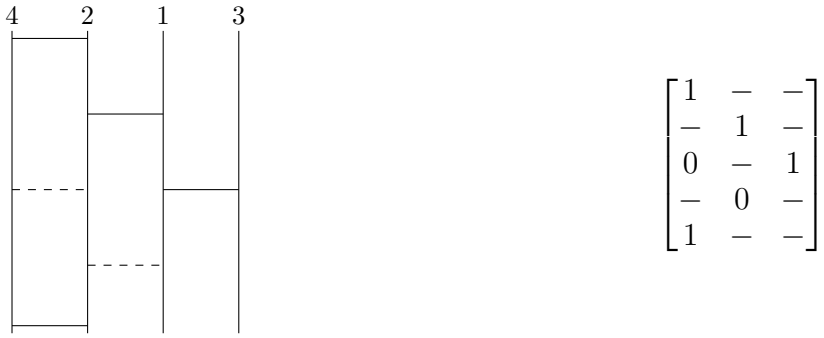
Proof. Let π be the descending permutation of order n. Let $route(x)$ be the route of $n \geq x \geq 2 \in \pi$. We know that when $x = n$, $route(x)$ has $n-1$ bars, each requiring their own row. Thus, $CL(n, n-1, \ldots, 1)$ requires at least $n-1$ rows. For each subsequent x from $[n-1 \ldots 2]$, each $route(x)$ requires one more row to be added to $CL(n, n-1, \ldots 1)$. We get an additional $n-2$ rows added to CL, one for each $route([n-1 \geq x \geq 2])$. Therefore, the total number of rows is $(n-1)+(n-2) = 2(n-1)-1$. □

Corollary 3.4.2 *The upper bound for the number of rows of any canonical ladder of order n is $2(n-1)-1$.*

Proof. By Theorem 3.4.1, we know that $2(n-1)-1$ is the number of rows required for the canonical ladder corresponding to the descending permutation of order n. By removing bars from the canonical ladder for the descending permutation of order n, we can derive any other canonical ladder for any other permutation of order n; removing a bar translates to removing an inversion in π. Removing bars from the canonical ladder corresponding to the descending permutation of order n does not necessarily remove a row from the ladder, however, removing bars from the ladder certainly does not add any more rows to *ladder*. Therefore, $2(n-1)-1$ is the upper bound for the number of rows required for any ladder of order n. □

Using Theorem 3.4.1 and Corollary 3.4.2, we represent a canonical ladder as a binary matrix with $2(n-1)-1$ rows and $n-1$ columns. Let $matrix[row][col] = 1$ indicate a bar at the given row and column in the canonical ladder along an associated diagonal. Let $matrix[row][col] = 0$ indicate the absence of a bar at the given row and column in

the canonical ladder along an associated diagonal. Let $matrix[row][col] = -$ indicate irrelevancy as to whether a 1 or 0 is at the given row and column because the row and column does not lie on an associated diagonal. To see the canonical ladder for $(4, 2, 1, 3)$ and the corresponding binary matrix, refer to Figure 3.5.



$$\begin{bmatrix} 1 & - & - \\ - & 1 & - \\ 0 & - & 1 \\ - & 0 & - \\ 1 & - & - \end{bmatrix}$$

Figure 3.5: Canonical ladder and corresponding binary matrix.

Using Equation 3.1, and the binary matrix representation of a ladder, we define algorithm CreateCanonical which can be found in Algorithm 2 to create the canonical ladder for any given permutation of order n. The initial conditions of CreateCanonical are the following. Let *ladder* be the binary matrix representing $CL(\pi)$ initialized to $-$. Let π be an arbitrary permutation of order n. Let x be the currently maximal element in π, initialized to n. Let GetCoordinates refer to Equation 3.1.

Algorithm 2 The algorithm for creating the canonical ladder of $OptL\{\pi\}$

```
1:  function CreateCanonical(ladder, π, x)
2:      while x > 1 do
3:          pos(x) ← index of x ∈ π
4:          for i from 1 to x do
5:              if i ≠ pos(x) then
6:                  (row, column) ←GetCoordinates(x, pos(x), i)
7:                  if pos(x) < i then
8:                      ladder[row][col] := 1
9:                  else
10:                     ladder[row][col] := 0
11:         π ← π − x
12:         x ← x − 1
```

On each iteration of the while loop, all $x-1$ $1s$ and $0s$ along the associated diagonal of the xth element are added to *ladder* within the for-loop. GetCoordinates returns the row and column coordinate for the respective 1 or 0 for the pair of elements, $x > y$. Note that we call GetCoordinates with the arguments $(x, pos(x), i)$. When referring to Equation 3.1 the variables are $(x, pos'(x), pos'(y))$. Prior to the next iteration of the while loop, x is removed from π and all elements to the right of x are moved to the left by one index. Then, x is decremented by 1. The while loop continues until $x = 1$. Determining $pos(x)$ is done in $O(x)$ time. Removing x from π is done in $0(x)$ time. Calculating the row and column for each (x, y) is done in $O(1)$ time. Overall, the algorithms complexity is $O(n^2)$. We also note that CreateCanonical creates the ladder representation with clean level 1. Given three elements, $x > y > z$ and $pos(x) < pos(y) < pos(z)$, Equation 3.1 returns a lesser row for (x, z) than for (y, z). Therefore, CreateCanonical creates a representation of a ladder with clean level 1. To see the resulting state of *ladder* for each iteration of CreateCanonical with $(5, 7, 3, 4, 1, 2, 6)$ please refer to Figure 3.6.

A main contribution of this thesis is the canonical ladder and the algorithm CreateCanonical. By defining the canonical ladder we are able to list all $n!$ canonical ladders in any order. Let L_n be the set of all $n!$ canonical ladders.

Lemma 3.4.3 *Using* CreateCanonical*, we can list L_n in any order.*

Proof. Let S_n be the set of all permutations of order n. We simply apply an ordering to list S_n. For each permutation in S_n, we apply CreateCanonical, thus listing L_n in the same order as S_n. □

We create each $CL(\pi_n)$ in $O(n^2)$ time using CreateCanonical. However, using CreateCanonical to list L_n is inefficient, seeing as each ladder is created in $O(n^2)$ time. In chapters 4 and 5 we provide two Gray code listing algorithms, ModifiedSJT and ListLnByKBars, to list L_n in constant amortized time. The details of these algorithms are found in chapters 4 and 5 respectively.

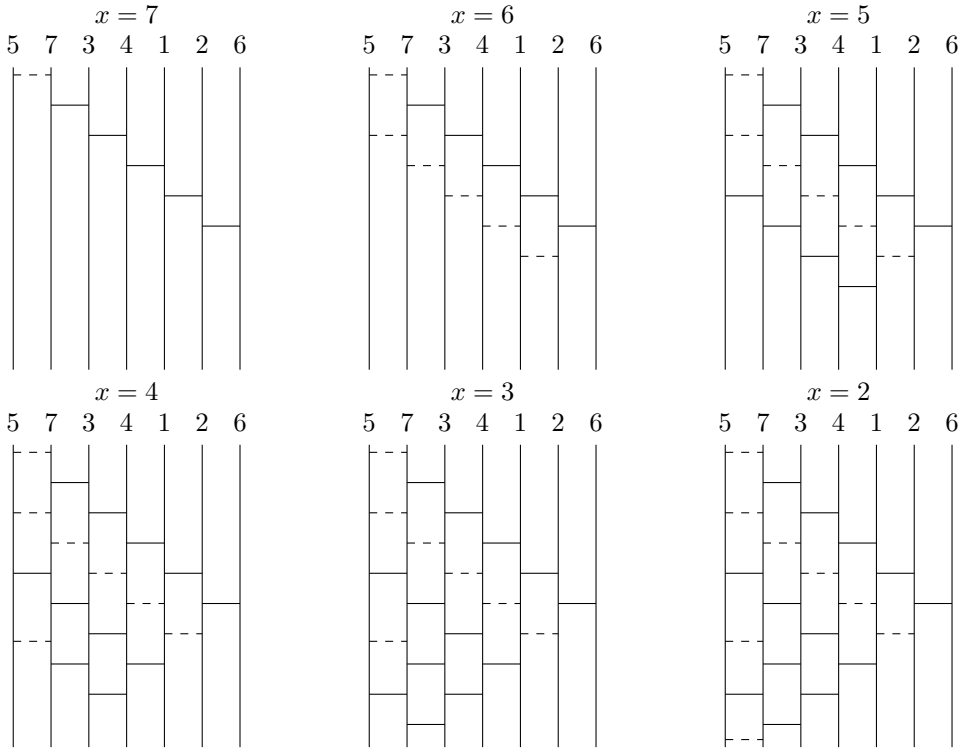


Figure 3.6: The ordering of the state of the ladder when creating the root ladder for $(5, 7, 3, 4, 1, 2, 6)$

Chapter 4

Listing L_n in Gray Code Order by Adding or Removing a Bar

In this chapter, we use the definition of the canonical ladder and the binary matrix representation of the canonical ladder to list L_n in Gray code order. Recall, a Gray code is an ordering such that successive elements in the ordering differ by a minimal amount of change. When listing L_n, a minimal amount of change is defined as either adding or removing a bar, or relocating a bar; relocating a bar is the same as adding one bar and removing another. In the case of this chapter's algorithm, we add or remove a bar when transitioning between ladders in L_n. A key step in this chapter is that we are able to determine the bar corresponding to an inversion in constant time. Thus, the running time for this chapter's algorithm is CAT.

4.1 Equation: GetCoordinates2

In the previous chapter, we described how we could list L_n using the naive approach. However, using the naive approach is slow. In this chapter we seek to improve the running time for listing L_n. In this section, we improve the running time by determining the location of a bar in $O(1)$ time for a pair of adjacent elements. This allows us to take advantage of the Steinhaus-Johnson-Trotter algorithm, which applies adjacent transpositions to elements.

We define r, with respect to a permutation π, as a vector from $[1 \dots x \dots n]$ as follows: r_x equals the number of elements less than x and to the right of x in π.

We note that $[0 \leq r_x \leq x-1]$. For example, given $(3,4,1,5,6,2)$, $r_1 = 0$, $r_2 = 0$, $r_3 = 2$, $r_4 = 2$, $r_5 = 1$ and $r_6 = 1$. We use r to define GetCoordinates2 found in equation 4.1. Let $x > y$ be two elements in π such that $pos(x) = pos(y) + / - 1$. We are going to apply an adjacent transposition to x and y; we need to know where to locate the 1 or 0 in the binary matrix corresponding to the elements x, y in order to flip it from a 1 to a 0 or vice versa. Prior to the transposition of x and y, if $pos(x) = pos(y) - 1$ then GetCoordinates2 returns the row and column of the 1 corresponding to x and y. If $pos(x) = pos(y) + 1$, then GetCoordinates2 returns the row and column of the 0 corresponding to x and y. r_x is updated accordingly by being incremented or decremented by 1.

$$\text{(row,column)} = \begin{cases} ((2n-x-1)-r_x, (x-1)-r_x) & \text{if pos(x)=pos(y)+1} \\ ((2n-x)-r_x, (x)-r_x) & \text{if pos(y) = pos(x)+1} \end{cases} \tag{4.1}$$

Initially, if $pos(x) = pos(y)+1$, then when we apply an adjacent transposition between x and y, an inversion is induced in π. Therefore, r_x is incremented by 1, and the bar (x, y) has been added to the canonical ladder. Initially, if $pos(x) = pos(y)-1$ then when we apply an adjacent transposition between x and y, an inversion is removed from π. Therefore, r_x is decremented by 1, indicating the bar (x, y) has been removed from the canonical ladder. One of the main results of this thesis is ModifiedSJT produces L_n in Gray code order in CAT. ModifiedSJT uses GetCoordinates2 and r to calculate the row and column for every 1 or 0 in the binary matrix in $O(1)$ time.

4.2 Algorithm: ModifiedSJT

In this section we define the algorithm ModifedSJT, using r and Equation 4.1, which can be found in Algorithm 3. Transposing two adjacent elements in π_i results in a

subsequent permutation π_j. The transposition of the two elements results in adding or removing a bar in $CL(\pi_i)$, resulting in $CL(\pi_j)$. The transposition of the two elements is merely simulated in Algorithm 3, seeing as π is not an argument to the function; we are able to calculate the location of the 1 or 0 in the binary matrix without having to induce the transposition in π. This makes our algorithm more interesting, seeing as we do not actually perform any swap operations on inversions. The initial conditions of ModifiedSJT are the following: Let *ladder* be the binary matrix representation of the canonical ladder such that each row and column coordinate along each associated diagonal is initialized to 0. Let n be the maximum element. Let x be initialized to 2; x represents the current element. On each recursive call x is incremented by 1 from $2, 3 \ldots n, n+1$. Let *direction* be a one indexed array set to **false** for all indexes. $direction[x]$ =false implies the xth is being adjacently transposed from right to left. $direction[x]$ =**true** implies the xth is being transposed from left to right. Let r be the inversion vector initialized at all indices to 0; when r is initialized to all $0s$, this means that r is the inversion vector for $\pi = (1, 2, \ldots, n-1, n)$.

Algorithm 3 Modification of the SJT algorithm for listing L_n

```
1: function ModifiedSJT(n, ladder, x, direction, r)
2:     if x > n then
3:         if number of bars in ladder equals 0 then Print(ladder)
4:         return
5:     for i from 1 to x do
6:         if i = 1 then
7:             ModifiedSJT(n, ladder, x + 1, direction, r)
8:         else
9:             if direction[x] =false then
10:                (row, column) ← ((2n − x − 1) − r_x, (x − 1) − r_x)
11:                r_x ← r_x + 1
12:                ladder[row][column] ← 1
13:            else
14:                (row, column) ← ((2n − x) − r_x, (x) − r_x)
15:                r_x ← r_x − 1
16:                ladder[row][column] ← 0
17:            Print(ladder)
18:            ModifiedSJT(n, ladder, x + 1, direction)
19:    if direction[x] = false then direction[x] ← true
20:    else direction[x] ← false
```

We use GetCoordinates2 on lines 10 and 14 of ModifiedSJT to calculate the row and column. When $direction[x]$ equals **false** we use the first case from GetCoordinates2. When $direction[x]$ equals **true** we use the second case from GetCoordinates2 to calculate the row and column.

Given the current value of x, add or remove a bar for the route of x, then add or remove all bars for route $x + 1$. Once all bars for route $x + 1$ have been added or removed, proceed to add or remove the next bar from the route of x. Repeat until all $x - 1$ bars have been added or removed from route x. If $direction[x]$ is $false$, then bars of x will be added to *ladder* from right to left, bottom to top, until no more bars to the route of x can be added. If $direction[x]$ is $true$, then bars will be removed from *ladder*, left to right, top to bottom, until no more bars from the route of x can be removed. Once all the bars for the route of x have been added or removed, then $direction[x]$ is negated, indicating that the opposite operation will be applied

to the bars of the route of x when x is next processed. On each recursive call, x is incremented by 1. When x is greater than n, return.

4.3 Analysis of ModifiedSJT

In this section we provide a theoretical analysis of ModifiedSJT along with the runtime for the algorithm. We have implied that ModifiedSJT produces L_n. We note that there are two general criteria for L_n. The first, each ladder in L_n corresponds to a unique permutation of order n. The second, each ladder in L_n is the canonical ladder. In Lemma 4.3.1, we prove the first criteria for L_n.

Lemma 4.3.1 *Each ladder produced by* ModifiedSJT *corresponds to a unique permutation of order n*

Proof. Since the algorithm is a modification of the Steinhaus-Johnson-Trotter algorithm, a similar proof for the SJT algorithm can be applied to the ModifiedSJT algorithm. Suppose we want to list all $n!$ ladders of order n. Suppose we have all $n-1!$ ladders of order $n-1$, then for each ladder of order $n-1$ add a new column to the right; this results in $n-1$ columns seeing as the number of columns is one less than the number of elements. For each of the $n-1!$ ladders with $n-1$ columns add $0 \ldots n-1$ bars beginning at column $n-1$ and ending at column 1. Doing so results in $(n-1)!n = n!$ ladders of order n. To see an example of the proof please refer to Figure 4.1. □

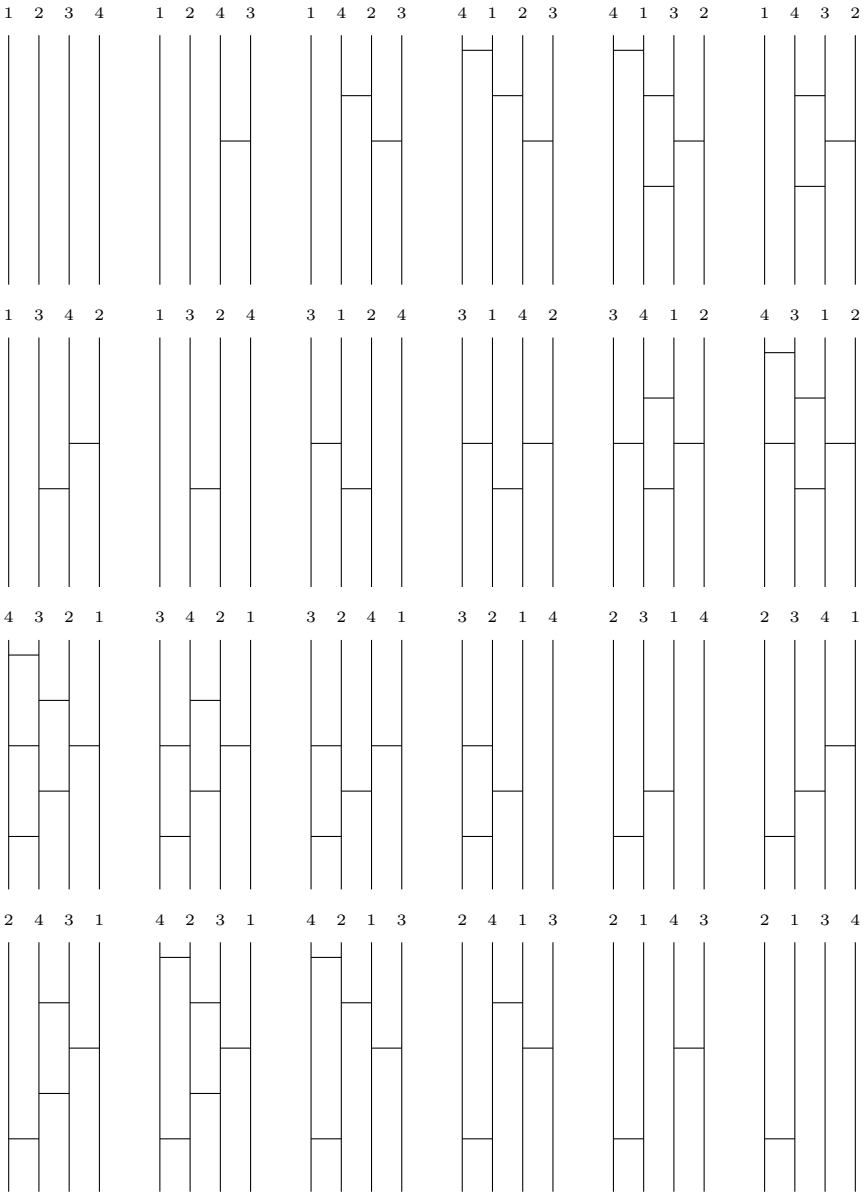


Figure 4.1: L_4 generated by the ModifiedSJT algorithm. The algorithm inserts or removes a bar between any two successive ladders.

To prove that each ladder produced by ModifiedSJT is the canonical ladder, we prove that the calculations for the $(row, column)$ coordinates lie on the associated diagonal of x. Recall that the canonical ladder is the ladder such that for each element, $2 \leq x \leq n$, each bar associated with x lies on the associated diagonal of x. The associated diagonal of x starts at $(2n-2x+1, 1)$ and ends at $(2n-x-1, x-1)$.

The calculations for the row and column for the bar depend on whether the bar is being added or removed. Thus, there are four cases to consider. We will prove that for each case, the calculation for the $(row, column)$ coordinates is located along the associated diagonal of x. The cases are the following:

Case 1: Bar is being added
Row is being calculated.

Case 2: Bar is being added
Column is being calculated.

Case 3: Bar is being removed
Row is being calculated.

Case 4: Bar is being removed.
Column is being calculated.

Lemma 4.3.2 *Assume a bar is being added and a row is being calculated. We use* $(2n-x-1)-r_x$ *to calculate the row. The row calculation is bounded by the uppermost row and bottommost row composing the associated diagonal of* x*.*

Proof. Suppose $r_x = 0$, then we know that x currently has no associated bars in the ladder. We know that the first bar associated with x will be located at row $(2n-x-1)-0$ which is also the bottommost row composing the associated diagonal of x. Suppose $r_x = x-2$, then we know that x has one more associated bar be added to the ladder. The bar will be located at $(2n-x-1)-(x-2)$ which is simplified to

$2n-2x+1$; $2n-2x+1$ equals the uppermost row composing the associated diagonal of x. □

Lemma 4.3.3 *Assume a bar is being added and a column is being calculated. We use $(x-1)-r_x$ to calculate the column. The column calculation is bounded by the leftmost column and rightmost column composing the associated diagonal of x.*

Proof. Suppose $r_x = 0$, then we know that x currently has no associated bars in the ladder. We know that the first bar associated with x will be located at $(x-1)-0$ which is also the rightmost column composing the associated diagonal of x. Suppose $r_x = x-2$, then we know that x has one more associated bar to be added to the ladder. The bar will be located at $(x-1)-(x-2)$ which is simplified to 1; 1 equals the leftmost column composing the associated diagonal of x.

□

Lemma 4.3.4 *Assume a bar is being removed and a row is being calculated. We use $(2n-x)-r_x$ to calculate the row. The row is bounded by the uppermost row and bottommost row composing the associated diagonal of x.*

Proof. Suppose $r_x = x-1$, then we know that x has all of its associated bars in the canonical ladder. We calculate the row of the first bar to be removed at $(2n-x)-(x-1) = (2n-2x)+1$; $(2n-2x)+1$ is the uppermost row of the associated diagonal of x. Suppose $r_x = 1$, then we know that x only has one more bar to be removed from the ladder. We calculate the row of the last bar to be removed at $(2n-x)-1$; $2n-x-1$ is the bottommost row of the associated diagonal of x.

□

Lemma 4.3.5 *Assume a bar is being removed and a column is being calculated. We use $(x)-r_x$ to calculate the column. The column is bounded by the leftmost column and rightmost column composing the associated diagonal of x.*

Proof. Suppose $r_x = x - 1$, then we know that x has all of its associated bars in the canonical ladder. We calculate the column of the first bar to be removed at $(x) - (x - 1) = 1$; 1 is the leftmost column of the associated diagonal of x. Suppose $r_x = 1$, then we know x only has one more bar to be removed from the ladder. We calculate the column of the last bar to be removed at $(x) - 1$; $x - 1$ is the rightmost column of the associated diagonal of x. □

Using Lemmas 4.3.1, 4.3.2, 4.3.3, 4.3.4, 4.3.5 we have shown that ModifiedSJT produces L_n. However, we have yet to prove that ModifedSJT produces L_n in Gray code order. We use Theorem 4.3.6 to prove that ModifiedSJT produces L_n in Gray code order.

Theorem 4.3.6 ModifiedSJT *produces L_n in Gray code order*

Proof. Let l_i be an arbitrary ladder produced by ModifedSJT. Let j be the next ladder to be produced by ModifiedSJT. If $l_i = l_j$ plus or minus a bar, then ModifiedSJT produces L_n in Gray code order. On each iteration of the loop, $direction[x]$ is either true or false, but not both. When $direction[x] = false$, a single 1 is added to the associated diagonal of x, indicating a bar has been added. When $direction[x] = true$, a single 0 is added to the associated diagonal of x, indicating a bar has been removed. With an addition or removal of a 1 or 0, l_i differs from l_j by the addition or removal of a bar. Therefore, L_n is produced by ModifedSJT in Gray code order.

□

Theorem 4.3.7 ModifiedSJT *produces L_n in CAT.*

Proof. The first ladder is printed in $O(n)$ time because $[2 \dots n + 1]$ recursive calls are made before first ladder is printed. Each subsequent ladder deviates from its predecessor by a single addition or removal of a bar. Calculating the location of a 1 or 0, corresponding to a bar or absence of a bar, is done in $O(1)$ time. Once

calculated, the 1 or 0 is added to l_i producing l_j. Once l_i is transformed into l_j, l_j is printed. □

We calculated the runtimes for $n = 1 \ldots 13$. The runtime is calculated without the ladders being printed. When the ladders are printed, the runtime increases by a substantial amount. The runtimes from $n = 9 \ldots n = 13$ are provided in Table 4.1.

Runtimes for generating L_n in seconds	
n	ModifiedSJT Runtime
9	0.00
10	0.03
11	0.25
12	2.78
13	66.97

Table 4.1: The table with the runtimes for listing L_n using ModifiedSJT.

4.4 Chapter Conclusion

We have provided an algorithm for listing L_n with CAT time per ladder by adding or removing a bar. By doing so, we have provided a solution to The Canonical Ladder Listing Problem. We use Equation 4.1 and modified the Steinhaus-Johnson-Trotter algorithm in order to create Algorithm 3. We computed the location of a bar corresponding to any pair of adjacent elements in $O(1)$ time.

Chapter 5

Listing L_n in Gray Code Order with k Bars

In this chapter we provide a Gray code for listing all ladders with k bars. We define $L_{n,k}$ as the set of all canonical ladder lotteries with n lines and exactly k bars, where $0 \leq k \leq \binom{n}{2}$. To list $L_{n,k}$, we define a mapping from associated diagonals to an inversion vector corresponding to π. We apply this mapping to a generate a Gray code for listing $L_{n,k}$. We then provide an algorithm for listing L_n ordered by the number of bars from $0 \leq k \leq \binom{n}{2}$ in Gray code order. We list L_n by k bars as follows - Let k be the current number of bars in the ladder; k is initialized to 0 and terminates at $\binom{n}{2}$. We first list $L_{n,k}$ before we list the first ladder in $L_{n,k+1}$. When listing ladders by k bars, we define minimal change as the relocation of a bar; relocating a bar is equivalent to removing one bar and adding another bar.

5.1 Listing all ladders with n lines and k bars in Gray code order

In Chapter 2, we discussed Effler-Ruskey's algorithm and Walsh's Gray code for listing $S_{n,k}$. In this section, we focus on Walsh's Gray code for listing $L_{n,k}$ by applying a minimal amount of change between subsequent ladders. We relocate a bar to transition from one ladder in $L_{n,k}$ to the next ladder in $L_{n,k}$. Recall, relocating a bar is the same as adding one bar and removing another bar.

An n part composition of a non negative integer k, is an n tuple $g = (g_1, g_2, \ldots g_n)$ whose sum adds to k. We say g is the *bar vector of* $CL(\pi_{n+1})$ when we map g to a canonical ladder as follows - For each index x in g, x is mapped to the associated

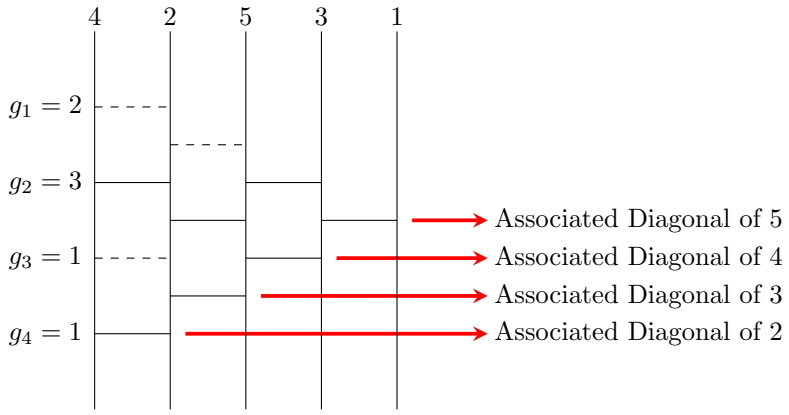


Figure 5.1: Mapping of the composition $(2, 3, 1, 1)$ to the associated diagonals of $CL(4, 2, 5, 3, 1)$

diagonal of $((n + 1) - x) + 1$. Furthermore, g_x equals the number of bars along the associated diagonal of element $((n + 1) - x) + 1$. To see the mapping of g to the associated diagonals, refer to Figure 5.1.

g is $(m_1, \ldots, m_n)$ bounded, if for each g_x, $0 \leq g_x \leq m_x$ where $m_x = (n+1)-x$; we let m_x be the fixed upper bound on g such that $g_x \leq m_x$. We let 0 be the fixed lower bound on g such that $0 \leq g_x$. We let $k-(g_{x+1}+\cdots+g_n)$ be the fluid upper bound on g such that $g_x \leq k-(g_{x+1}+\cdots+g_n)$. We let $k-(g_{x+1}+\cdots+g_n)-(m_1+\cdots+m_{x-1})$ be the fluid lower bound on g such that $k-(g_{x+1}+\cdots+g_n)-(m_1+\cdots+m_{x-1}) \leq g_x$. We let the maximum value of $g_x = min(m_x, k - (g_{x+1} + \cdots + g_n))$. We let the minimum value of $g_x = max(0, k - (g_{x+1} + \cdots + g_n) - (m_1 + \cdots + m_{x-1}))$. For example, suppose $k = 9$, $n = 5$, $x = 3$, $g =$(_,_,_,1) and $m = (4, 3, 2, 1)$. The minimum value of $g_3 = max(0, 9 - 1 - (4 + 3))$ and the maximum value of $g_3 = min(2, 9 - 1)$. Walsh's Gray Code for listing compositions is as follows follows - We start with the lexicographically largest composition $(k, 0, \ldots, 0)$. We define the *last value of* g_x as either the max or min value of g_x depending on whether the suffix $(g_{x+1} + \cdots + g_n)$ is odd or even. If the suffix is of odd parity, then the last value of g_x is equal to the $max(0, k-(g_{x+1}+\cdots+g_n)-(m_1+\cdots+m_{x-1}))$. If the suffix is of even parity, then the last value of g_x is equal to the $min(m_x, k - (g_{x+1} + \cdots + g_n))$. We scan left to right, starting at $x = 2$, to find the first g_x that is not at its last value; if no such

g_x exists, then the current composition is the last one. Once found, g_x is increased by 1 if $(g_{x+1} + \cdots + g_n)$ is even, else g_x is decreased by 1. Because we are obligated to ensure g sums to k, when g_x is incremented or decremented by 1, then we must decrement or increment some $(g_1, \ldots g_w \ldots, g_{x-1})$ by 1. We determine which g_w to change by scanning right to left, starting at g_{x-1} until we find the smallest $w < x$ such that g_w can be incremented by 1 if g_x was decremented by 1, or decremented by 1 if g_x was incremented by 1.

Although we use the same sequencing rule as Walsh to list the compositions, our mapping of the composition to the associated diagonals lends to a different ordering $S_{n+1,k}$ than that of Walsh's ordering of $S_{n+1,k}$. Note, that Walsh's g composition is the same as our bar vector, however we map our bar vector to the associate diagonals of the canonical ladder. In Table 5.1 we see the composition listing for $n = 4$ and $k = 2$, Walsh's listing of $S_{5,2}$, and our listing of permutations of $S_{5,2}$, derived from our mapping of the bar vector to the canonical ladder. We have

g/Bar vector	Walsh's Permutation	Our Permutation
$(2,0,0,0)$	$(3,1,2,4,5)$	$(1,2,5,3,4)$
$(1,1,0,0)$	$(2,3,1,4,5)$	$(1,2,4,5,3)$
$(0,2,0,0)$	$(1,4,2,3,5)$	$(1,4,2,3,5)$
$(0,1,1,0)$	$(1,3,4,2,5)$	$(1,3,4,2,5)$
$(1,0,1,0)$	$(2,1,4,3,5)$	$(1,3,2,5,4)$
$(0,0,2,0)$	$(1,2,5,3,4)$	$(3,1,2,4,5)$
$(0,0,1,1)$	$(1,2,4,5,3)$	$(2,3,1,4,5)$
$(0,1,0,1)$	$(1,4,3,5,2)$	$(2,1,4,3,5)$
$(1,0,0,1)$	$(2,1,3,5,4)$	$(2,1,3,5,4)$

Table 5.1: Table comparing the listing of permutations with the same composition

claimed we can list compositions g in such a way that we can list $L_{n+1,k}$ by mapping each g_x to the number of bars along the associated diagonal of $((n+1) - x) + 1$. We have yet to demonstrate the details of this mapping. If $k > 0$, we define *the lexicographically largest ladder* as the ladder such that the associated diagonal of

any element $y > w$, the number of bars in $y's$ diagonal is the $min(m_{(n+1-y)+1}, k -$ the number of bars along the associated diagonals of $n+1, n, \ldots, y+1)$. We start with the lexicographically largest ladder and the corresponding lexicographically largest g. To list g, we use the same sequencing rule as Walsh. To list $L_{n+1,k}$, we let $x' = (n+1) - x + 1$ and we let $w' = (n+1) - w + 1$, where $x \neq w$. We add or remove one bar from the associated diagonal of x' and remove or add another bar from the associated diagonal of w'; we always do the inverse operation on x' and w' in order to ensure k does not change. When we remove a bar, we do so from the top left of an associated diagonal, and when we add a bar, we do so to the bottom right of an associated diagonal. To find our x' value, we scan g from $(g_2, \ldots, g_n)$ until we find a g_x that is not at its last value. We then increment g_x by 1 and add a bar to the associated diagonal of x' if the suffix $(g_{x+1} + \cdots + g_n)$ is even. Otherwise we decrement g_x by 1 and remove a bar from the associated diagonal of x'. Once we have added or removed a bar from x' we scan the prefix $(g_{x-1}, \ldots, g_1)$ from right to left, to find the smallest w value less than x such that g_w is not at its last value. If g_x was incremented by 1, then g_w is decremented by 1 and a bar along the associated diagonal of w' is removed. If g_x was decremented by 1, then g_w is incremented by 1 and a bar along the associated diagonal of w' is added. We use Equation 5.1 to calculate the row and column of the bar to be added or removed.

$$\text{(row,column)}= \begin{cases} ((n+x) - g(x), (n+1-x) - g(x) + 1) & \text{if a bar is being removed} \\ ((n+x-1) - g(x), (n+1-x) - g(x)) & \text{if a bar is being added} \end{cases} \tag{5.1}$$

In Figure 5.2 we provide the listing of $L_{5,2}$ using Walsh's Gray code for listing g and our mapping of g to the associated diagonals of the canonical ladder.

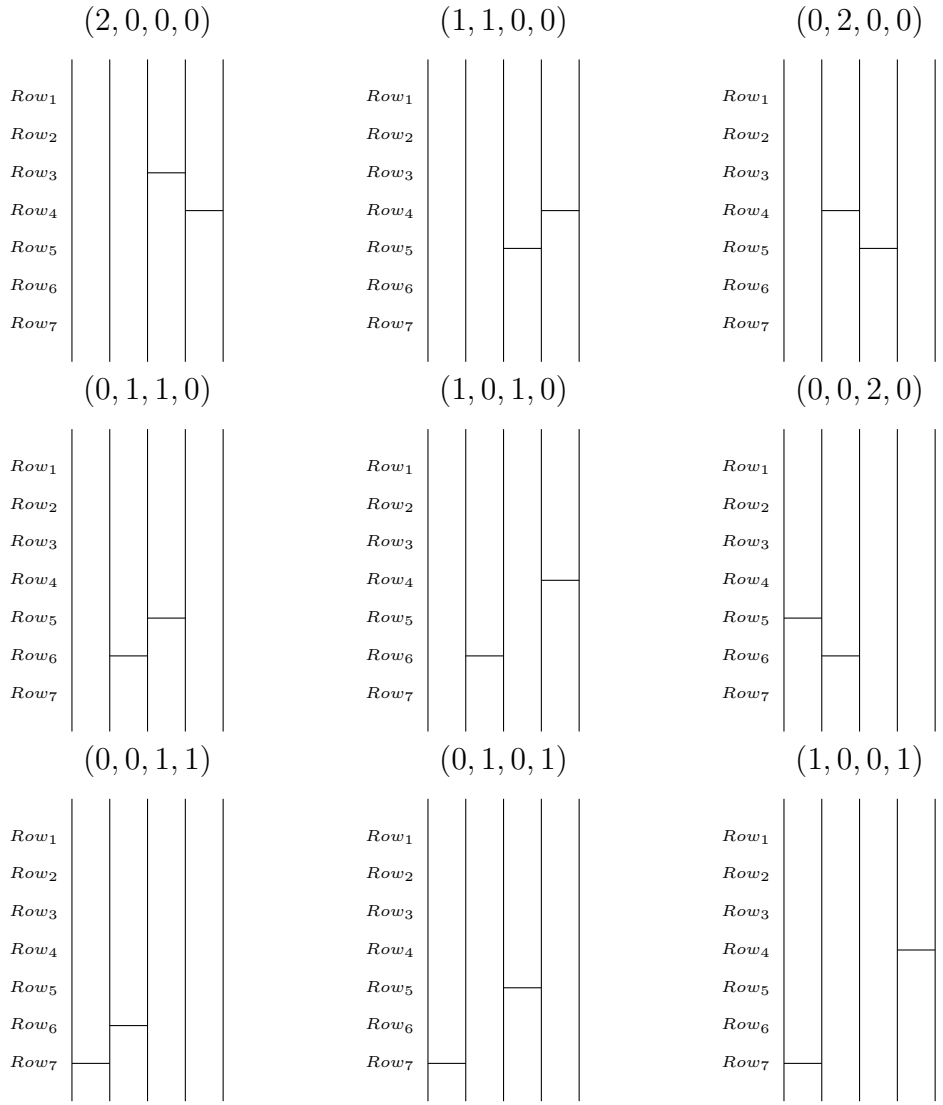


Figure 5.2: Listing $L_{5,2}$ with Walsh's Gray code on bar vector g. Ladders are read left to right, top to bottom. g is listed above each ladder.

Walsh provides a loop free algorithm to list g in Gray code order [21]. His algorithm provides the next instance of g in $O(1)$ time by providing the x and w indices in $O(1)$ time. We use Walsh's algorithm to provide the x and w value in order to list the next canonical ladder in $L_{n+1,k}$. Walsh refers to x as pivot one and w as pivot two. We note that $x > w$. We also note that g_x is incremented or decremented before g_w is decremented or incremented. In Algorithm 4, termed NextLadderNK, we list the next canonical ladder using the pivots x and w provided by Walsh's algorithm. The initial conditions of NextLadderNK are the following. Let x be pivot one returned from Walsh's algorithm. Let w be pivot two returned from Walsh's algorithm. Let CL be a canonical ladder with $n+1$ lines and k bars. Let *increment* be a Boolean variable set to **true** if the pivot one is incremented and pivot two is decremented, else *increment* is set to **false**.

Algorithm 4 Algorithm to list the next ladder in $L_{n+1,k}$

1: **function** NextLadderNK(x, w, CL, *increment*)
2: **if** *increment* =**true then** ▷ Add bar to ass. diag. $(n+x)-1$
3: $(row_x, column_x) \leftarrow ((n+x-1)-g_x, (n+1-x)-g_x)$
4: $(row_w, column_w) \leftarrow ((n+w)-g_w, (n+1-w)-g_w+1)$
5: $CL[row_x][column_x] \leftarrow 1$
6: $CL[row_w][column_w] \leftarrow 0$
7: **else** ▷ Remove bar from ass. diag. $(n+x)-1$
8: $(row_w, column_w) \leftarrow ((n+w-1)-g_w, (n+1-w)-g_w)$
9: $(row_x, column_x) \leftarrow ((n+x)-g_x, (n+1-x)-g_x+1)$
10: $CL[row_x][column_x] \leftarrow 0$
11: $CL[row_w][column_w] \leftarrow 1$

Corollary 5.1.1 NextLadderNK *produces the next ladder in* $L_{n+1,k}$ *in* $O(1)$ *time.*

Proof. From Walsh's paper, we know that we can attain the first and second pivot in $O(1)$ time. We use these pivots, x and w, to calculate the row and column for the coordinates of the bar to add, and the coordinates of the bar to remove, in $O(1)$ time. □

We use NextLadderNK as a subroutine in Algorithm 5, termed ListLNK,

which lists $L_{n,k}$ in Gray code order. Let k be the number of bars. Let n be the number of lines in CL. Let g be initialized to the lexicographically largest composition; $(k, 0, \ldots, 0)$. Let m be the bound on g. We note that g and m are of order $n-1$. Let CL be the canonical ladder data structure initialized as the lexicographically largest ladder. Let $p1$ and $p2$ be pivot one and pivot two respectively. Let *increment* be a Boolean set to **true** if $p1$ is to be incremented and $p2$ is to be decremented, else *increment* is set to **false**.

Algorithm 5 Algorithm for listing Ln, k

```
1: function ListLNK(g, m, CL, k)
2:     Print(CL)
3:     increment ← true
4:     while g_1 < m_1 and g_1 < k − (g_2 + ··· + g_{n−1}) do        ▷ while not at last composition
5:         (p1, p2) ←Walsh(g, m, ...)
6:         if (g_{p1+1}, + ..., +g_{n−1}) is even then p1 ← p1 + 1, p2 ← p2 − 1
7:         else p1 ← p1 − 1, p2 ← p2 + 1
8:         if g_1 > m_1 or g_1 > k − (g_2 + ··· + g_{n−1}) then▷ surpassed last composition
9:             g_1 ← g_1 − 1
10:            g_{p1} ← g_{p1} + 1
11:            return
12:        if (g_{p1+1} + ··· + g_{n−1}) is even then increment ← true
13:        else: increment ← false
14:        NextLadderNK(p1, p2, CL, increment)
15:        Print(CL)
```

By applying Walsh's algorithm, which returns $p1$ and $p2$ in $O(1)$ time, we produce $L_{n,k}$ in constant amortized time per ladder. We note that line 6 of the algorithm determines the parity of the suffix; Walsh demonstrates that the parity of the suffix can be determined in $O(1)$ time by maintaining a number of auxiliary data structures [21].

5.2 Listing L_n by k bars

In the previous section, we applied Walsh's Gray code for listing $L_{n,k}$. In this section, we further modify Walsh's Gray code to list L_n by way of listing $L_{n,0} \dots L_{n,k} \dots L_{n,\binom{n}{2}}$ in Gray code order. To reiterate, we list L_n by way of $L_{n,k}$ by first listing all ladders with exactly k bars before we list the first ladder with $k+1$ bars. In Figure 5.3 we show the listing of L_4 by listing $L_{4,0} \dots L_{4,k} \dots L_{4,6}$.

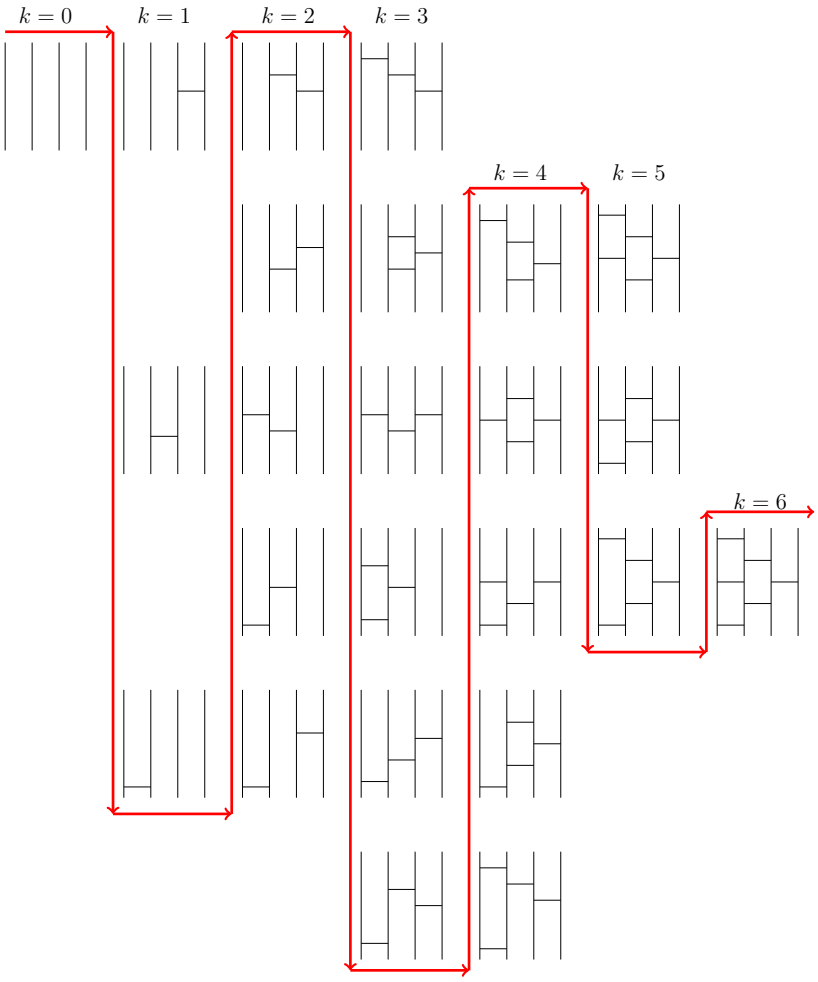


Figure 5.3: Listing of L_4 by k bars. For each k, successive ladders differ by the relocation of a bar. Transitioning from the last ladder in $L_{n,k}$ to the first ladder in $L_{n,k+1}$ requires the addition of a bar.

We previously defined the lexicographically largest ladder as the ladder such

the associated diagonal of any element $y > w$, the number of bars in $y's$ diagonal is the $min(m_{(n+1-y)+1}, k -$ the number of bars along the associated diagonals of $n + 1, n, \ldots, y + 1)$. We now define the *terminating ladder* as follows - let the first $k - 1$ bars compose the sub-ladder along the associated diagonals from $3 \ldots y \ldots n$; this sub-ladder is the lexicographically largest sub-ladder. Bar k occupies the associated diagonal of 2. For an example of the terminating ladder for $L_{5,6}$, please refer to Figure 5.4. We describe the corresponding g, termed the *terminating composition* g, as $(k - 1, 0, \ldots, 1)$.

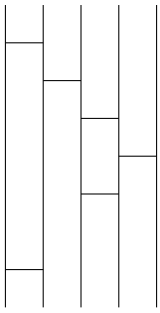


Figure 5.4: The terminating ladder for $L_{5,6}$

In order to list L_n by way of $L_{n,k}$, we use Walsh's Gray code on composition g when k is odd. Recall, that when k is odd we start at the lexicographically largest g and end at the terminating g. Therefore, our description of listing $L_{n,k}$ by way of g in the previous section still holds for odd values of k. However, when k is even we start with the terminating g and we have to work backwards so to speak, ending at the lexicographically largest g. We assume by providing the inverse of Walsh's Gray code, we can list g for even values of k. We provide two conjectures, Conjecture 5.2.1 and Conjecture 5.2.2, that could allow for the algorithmic implementation of the reverse ordering of Walsh's Gray code. If Conjecture 5.2.1 is true, we have a sufficient condition for Conjecture 5.2.2 being true.

Conjecture 5.2.1 *We refer to x as $p1$ and w as $p2$. When listing g in reverse order, we list g as follows-starting from g_2, scan left to right until you find a g_x not at its last value such that if $(g_{x+1}, + \cdots +, g_n)$ is odd and there exists some minimum*

$1 \leq w \leq x-1$ *such that* g_w *that is not at its minimum value then increment* g_{p1} *and decrement* g_{p2}. *If* $(g_{x+1}+\cdots+g_n)$ *is even and there exists some minimum* $1 \leq w \leq x-1$ *such that* g_w *is not at its maximum value, decrement* g_{p1} *and increment* g_{p2}.

Under the assumption Conjecture 5.2.1 is true, we state Conjecture 5.2.2.

Conjecture 5.2.2 *Assuming 5.2.1 is true, by starting with the terminating composition of* g *and ending at the lexicographically largest composition of* g, *we can implement the inverse of Walsh's algorithm for listing* g *in reverse ordering with* $O(1)$ *time per instance of* g.

In Figure 5.5, we apply Conjecture 5.2.1 to provide the Gray code for the composition g on $S_{5,4}$ in reverse order. We also give the Gray code for the permutations when we map g to the associated diagonals of the canonical ladder. We start with the terminating composition of g and the permutation corresponding to the terminating ladder. We end at the lexicographically largest composition and the permutation corresponding to the lexicographically largest canonical ladder. The pivots, $p1$ and $p2$ are bolded; the pivots bolded in composition i indicate which two values in composition i will be incremented and decremented in order to transition to composition $i+1$. Recall, that the first pivot $p1$ is greater than the second pivot $p2$.

Composition	Permutation
$(\mathbf{3}, 0, 0, \mathbf{1})$	$(2, 5, 1, 3, 4)$
$(\mathbf{2}, \mathbf{1}, 0, 1)$	$(2, 1, 5, 4, 3)$
$(\mathbf{1}, \mathbf{2}, 0, 1)$	$(2, 4, 1, 5, 3)$
$(\mathbf{0}, \mathbf{3}, 0, 1)$	$(4, 2, 1, 3, 5)$
$(\mathbf{0}, 2, \mathbf{1}, 1)$	$(2, 4, 3, 1, 5)$
$(\mathbf{1}, \mathbf{1}, 1, 1)$	$(2, 3, 4, 5, 1)$
$(\mathbf{2}, \mathbf{0}, 1, 1)$	$(2, 3, 5, 1, 4)$
$(\mathbf{1}, 0, \mathbf{2}, 1)$	$(3, 2, 1, 5, 4)$
$(0, \mathbf{1}, 2, \mathbf{1})$	$(3, 2, 4, 1, 5)$
$(\mathbf{0}, \mathbf{2}, 2, 0)$	$(3, 4, 1, 2, 5)$
$(\mathbf{1}, \mathbf{1}, 2, 0)$	$(3, 1, 4, 5, 2)$
$(\mathbf{2}, 0, \mathbf{2}, 0)$	$(3, 1, 5, 2, 4)$
$(\mathbf{3}, \mathbf{0}, 1, 0)$	$(1, 5, 3, 2, 4)$
$(\mathbf{2}, \mathbf{1}, 1, 0)$	$(1, 3, 5, 4, 2)$
$(\mathbf{1}, \mathbf{2}, 1, 0)$	$(1, 4, 3, 5, 2)$
$(\mathbf{0}, 3, \mathbf{1}, 0)$	$(4, 1, 3, 2, 5)$
$(\mathbf{1}, \mathbf{3}, 0, 0)$	$(4, 1, 2, 5, 3)$
$(\mathbf{2}, \mathbf{2}, 0, 0)$	$(1, 4, 5, 2, 3)$
$(\mathbf{3}, \mathbf{1}, 0, 0)$	$(1, 5, 2, 4, 3)$
$(4, 0, 0, 0)$	$(5, 1, 2, 3, 4)$

Figure 5.5: Ordering of g based on Conjecture 5.2.1 and the ordering of permutations corresponding to $L_{5,4}$ when g is mapped to the associated diagonals.

We use Conjecture 5.2.1 and Conjecture 5.2.2 to create Algorithm 6, termed ListLNKReverse to provide a reverse listing of all ladders of order n with k bars in Gray code order. Let k be the number of bars. Let n be the number of lines in CL. Let g be initialized to the terminating composition; $(k-1, 0, \ldots, 1)$. Let m be the bound on g. We note that g and m are of order $n-1$. Let CL be the canonical ladder data structure initialized as the lexicographically largest ladder. Let $p1$ and $p2$ be pivot one and pivot two respectively. Let *increment* be a Boolean set to **true** if $p1$ is to be incremented and $p2$ is to be decremented, else *increment* is set to **false**. Let ReverseWalsh be the hypothetical implementation of the inverse of Walsh's algorithm, which is used as a subroutine to return $p1$ and $p2$ in such a way that g can be listed in reverse order.

Algorithm 6 Algorithm for listing Ln, k starting from the terminating ladder and ending at the lexicographically largest ladder.

```
1: function ListLNKReverse(g, m, CL, k)
2:     Print(CL)
3:     increment ← true
4:     while g_1 < m_1 and g_1 < k − (g_2 + ··· + g_{n−1}) do ▷ while not at lex. largest composition
5:         (p1, p2) ←ReverseWalsh(g, m, ...)        ▷ Assuming Conjecture 5.2.2 is correct
6:         if (g_{p1+1}, +..., +g_{n−1}) is odd then p1 ← p1 + 1, p2 ← p2 − 1
7:         else p1 ← p1 − 1, p2 ← p2 + 1
8:         if g_1 > m_1 or g_1 > k − (g_2 + ··· + g_{n−1}) then      ▷ surpassed lex. largest composition
9:             g_1 ← g_1 − 1
10:            g_2 ← g2_{+1}
11:            return
12:        if (g_{p1+1}, +..., +g_{n−1}) is odd then increment ← true      ▷ Assuming Conjecture 5.2.1 is correct
13:        else: increment ← false
14:        NextLadderNK(p1, p2, CL, increment)
15:        Print(CL)
```

Assuming we implement the inverse of Walsh's algorithm, we would be able to list $L_{n,k}$ in constant amortized time in the reverse ordering of Walsh's Gray code order.

When we list L_n, if k is odd, the first ladder listed in $L_{n,k}$ is the lexicographically largest ladder and the last ladder listed is the terminating ladder. If k is even, the first ladder listed in $L_{n,k}$ is the terminating ladder and the last ladder listed is the lexicographically largest ladder. When listing L_n, we use the terminating ladder and the lexicographically largest ladder as the transitioning ladders between $L_{n,k}$ and $L_{n,k+1}$. When we reach the last ladder in $L_{n,k}$, we add a bar to said ladder and transition to the first ladder of $L_{n,k+1}$. We refer to this bar as the *incrementing bar*. To calculate the coordinates of the incrementing bar when transitioning from $L_{n,k}$ to $L_{n,k+1}$, we scan g from left to right to find the smallest x value such that g_x is not at its maximum value. We then use the same formula for adding a bar found in Equation 5.1, except we replace n with $n-1$ seeing as g is of order $n-1$ when mapping to canonical ladders with n lines. If the last ladder of $L_{n,k}$ is a terminating ladder, when we add the incrementing bar, the first ladder of $L_{n,k+1}$ is also a terminating ladder. If the las ladder of $L_{n,k}$ is a lexicographically largest ladder, when we add the incrementing bar, the first ladder of $L_{n,k+1}$ is also a lexicographically largest ladder.

To transition from $L_{n,k}$ to $L_{n,k+1}$ we want to add the incrementing bar to the largest associated diagonal that has room for one more bar. We know that by mapping g to the associated diagonals, we can use m to determine the largest associated diagonal that has room for one more bar. Based on our mapping of g to the associated diagonals, the smallest x value in g not at its m_x value corresponds to the largest associated diagonal where we can add bar $k+1$. Let $MinX$ be a global variable that keeps track of the smallest x value in g that is not at its m_x value. When we list g for $L_{n,k}$, we update $MinX$ with every transition from g^i to g^{i+1} using Algorithm 7. Let $p1$ be pivot one returned by Walsh's Gray code, let $p2$ be pivot two returned by Walsh's Gray code, let g be the current composition, let m be the upper bound on g, and let $MinX$ be the current smallest x value such that g_x is not at its m_x bound. With

Algorithm 7 Algorithm for updating the minimum x value such that $g_x < m_x$

```
1: function UpdateMinX(p1, p2, g, m)
2:     if p1 < MinX and g_p1 <Min(m_p1, k − (g_{p1+1} + ··· + g_{n−1})) then
3:         MinX ← p1
4:     if p2 < MinX and g_p2 <Min(m_p2, k − (g_{p1+1} + ··· + g_{n−1})) then
5:         MinX ← p2
6:     if g_MinX = Min(m_MinX, k − (g_{p1+1} + ··· + g_{n−1})) then
7:         if MinX! = p1 then
8:             MinX ← p1 return
9:         if MinX! = p2 then
10:            MinX ← p2 return
```

UpdateMinX, we are able to determine the row and column of the incrementing bar in $O(1)$ time. *Due to the ordering of the algorithms and concepts in this chapter, our pseudocode for* ListLNK *and* ListLNKReverse *does not contain a function call to* UpdateMinX. *However, when listing L_n we would call* UpdateMinX *at the end of the while loops found in* ListLNK *and* ListLNKReverse *to ensure that we have the correct $MinX$ value.* When ListLNK or ListLNKReverse terminate, the last value of $MinX$ is the value corresponding to the largest associated diagonal with room for one more bar. In Lemma 5.2.3 we prove that UpdateMinX updates $MinX$ with the minimum x value.

Lemma 5.2.3 UpdateMinX *updates $MinX$ with the minimum x value such that g_x is not at its max value m_x*

Proof. Each call to UpdateMinX either updates $MinX$ with $p1$ or $p2$, or leaves $MinX$ at its current value. When $k = 0$ $MinX = 1$. For each subsequent k value, if $MinX$ is at its m_x value, then $MinX = p1$ or $MinX = p2$ seeing as in the previous instance of g, g_{MinX} was not at its m_{MinX} value, for if it were, then the $MinX$ value would have been updated in the previous call to UpdateMinX. Therefore, if g_{MinX} is at its m_{MinX} value, g_{MinX} has been incremented in the current instance of g. Seeing as $p1$ and $p2$ are the two pivots, then g_{p1} and g_{p2} have been updated in the current instance of g. If g_{p1} has been incremented, then $MinX$ equals $p1$. If g_{p2} has been

incremented, then $MinX$ equals $p2$. We replace g_{MinX} with $p1$ if $g_{MinX} = m_{MinX}$ and $MinX$ does not equal $p1$, or we replace $MinX$ with $p2$ if $g_{MinX} = m_{MinX}$ and $MinX$ does not equal $p2$. If $MinX$ is not at its m_{MinX} value, we replace $MinX$ with $p1$ only if g_{p1} is not at its m_{p1} value and $p1 < MinX$. The same goes for $p2$. Thus, $MinX$ is always equal to the smallest x value such that g_{MinX} is not at its m_x value. □

In Algorithm 8, termed ListLnByKBars, we use the mapping of g to the canonical ladder, ListLNK, ListLNKReverse, and UpdateMinX as subroutines for listing L_n in Gray code order. We do so by listing $L_{n,0} \dots L_{n,k} \dots L_{n,\binom{n}{2}}$. Let k be initialized to 0; when k is even we call ListLNKReverse and when k is odd we call ListLNK. Let g be of order $n-1$ and initialized to $(0, 0, \dots, 0)$. Let m be the bound on g initialized to $(n-1, n-2, \dots 1)$. Let CL be initialized to the canonical ladder corresponding the ascending permutation; i.e. the ladder with zero bars. Let $MinX$ be initialized to 1. Figure 5.3 shows the listing of L_n produced by ListLnByKBars.

Algorithm 8 Algorithm to list L_n by $[0 \dots k \dots \binom{n}{2}]$ bars

```
1: function ListLnByKBars(g, m, k, CL, MinX)
2:     if k = 0 then
3:         Print(CL)
4:         ListLnByKBars(g, m, k + 1, CL, MinX)
5:     if k > (n choose 2) then return
6:     g_MinX ← g_MinX + 1
7:     (r1, c1) ← (((n−1)+MinX−1)−g(MinX), ((n−1)+1−MinX)−g(MinX))
8:     CL[r1][c1] ← 1
9:     if k is even and k > 0 then MinX ←ListLNKReverse(g, m, CL, k)
10:    else MinX ←ListLNK(g, m, CL, k)
11:    ListLnByKBars(g, m, k + 1, CL, MinX)
```

If k is odd, we list Ln, k in accordance with Walsh's ordering of g and if k is even we list $L_{n,k}$ in the reverse ordering of Walsh's Gray code. When we call ListLNK and ListLNKReverse as subroutines, we ensure that UpdateMinX is

called as a subroutine in these functions. By doing so, $Minx$ is the smallest x value not at its m_x value. With the correct value for $MinX$, we add a bar to the largest associated diagonal with room for the incrementing bar. We know that the maximum number of bars is $\binom{n}{2}$, therefore we terminate recursion when k exceeds $\binom{n}{2}$. Assuming ReverseWalsh is correct, ListLnByKBars lists L_n in constant amortized time and in Gray code order.

Lemma 5.2.4 *Each ladder produced by* ListLnByKBars *is unique.*

Proof. Walsh's Gray code lists all unique instances of g when k is odd. Each instance of g corresponds to a unique ladder due to our mapping to the associated diagonals. We assume that the ReverseWalsh produces the reverse ordering of Walsh's Gray code when applied to even values of k, making all ladders for even values of k unique. When we transition from k to $k+1$ we add a bar, making the last ladder of k and the first ladder of $k+1$ unique. □

5.3 Chapter Conclusion

We have provided a theoretical algorithm for listing L_n in Gray code order by k bars. Furthermore, we would be able to do so in constant amortized time. We have provided a second solution to The Canonical Ladder Listing Problem. What makes this solution interesting is that we list L_n ordered by the number of bars. Our future work is as follows:

1. Implement Walsh's loop free algorithm
2. Design and implement ReverseWalsh
3. Prove Conjectures 5.2.1 and 5.2.2
4. Develop applications for listing L_n by k bars

Chapter 6

Summary and Future Work

In summary, we have defined the canonical ladder, designed a representation of the canonical ladder along with equations to calculate the coordinates of the bars in the representation, implemented an algorithm to create the canonical ladder, implemented a constant amortized algorithm to list L_n in Gray code order by adding or removing a bar, designed a constant amortized algorithm to list $L_{n,k}$ in Gray code order, and designed a constant amortized algorithm to list L_n by listing $L_{n,0} \dots L_{n,k} \dots L_{n,\binom{n}{2}}$. The algorithms can be used to list sorting network arrangements that correspond to the canonical ladders.

We are left with the following open problems:

1. Proving Conjecture 5.2.1
2. Proving Conjecture 5.2.2
3. Implementing Algorithm 6, assuming Conjecture 5.2.2 is true.
4. Implement Algorithm 8, assuming we can implement ListLNKReverse.
5. Continuing to work on the counting problem for the number of ladders in $OptL\{\pi\}$, specifically when $\pi = (n, n-1, \dots, 2, 1)$.

Bibliography

[1] J. Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. www.jjj.de, 2010.

[2] K. Batcher. Sorting Networks and their Applications. *Spring Joint Computer Conference*, 1968.

[3] B. Bauslaugh and F. Ruskey. Generating Alternating Permutations Lexicographically. *BIT*, 30:17–26, 1990.

[4] C. T. Djamengi and M. Tchuente. A Cost-Optimal Algorithm for Permutation Generation in Lexicographic Order. *Journal of Parallel and Distributed Computing*, 1(44), 1997.

[5] A. Dumitrescu and R. Mandal. New Lower Bounds for the Number of Pseudoline Arrangements. *arXiv:1809.03619 [math.CO]*, 2018.

[6] S. Effler and F. Ruskey. A CAT Algorithm for Generating Permutations with a Fixed Number of Inversions. *Information Processing Letters*, 86(2):107–112, 2002.

[7] S. Felser and P. Valtr. Coding and Counting Arrangements of Pseudolines. *Discrete Comput Geom*, 46(405), 2011.

[8] F. Gray. *Pulse Code Communication*. Bell Telephone Laboratories Incorporated, New York USA, Serial NO 785697 edition, 1953.

[9] B. R. Heap. Permutations by Interchange. *The Computer Journal*, 3(6), 1963.

[10] S. M. Johnson. Generation of Permutations by Adjacent Transposition. *Mathematics of Computation*, 17:282–285, 1963.

[11] J. Kawahara, T. Saitoh, R. Yoshinaka, and M. Shin-Ichi. Couting Primitive Sorting Networks by πDDs. *TCS-TR-A*, 11(54), 2011.

[12] D. Knuth. *The Art of Computer Programming*, volume 4. Addison-Wesley, 2011.

[13] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA, 1997.

[14] Louis-Frederic. *Japan encyclopedia*. Belknap, 2002.

[15] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu Computing. *Proceedings of the IEEE*, 5(96), 2008.

[16] Ruskey, Mutze, Sawada, and Williams. The Combinatorial Object Server, 2018.

[17] C. Savage. A Survey of Combinatorial Gray Codes. *SIAM Rev*, 39(4), 1997.

[18] J. Sawada and A. Williams. A Hamiltonian Path for the Sigma Tau Problem. pages 568–575, 2018.

[19] N. Sloane. The On-line Encyclopedia of Integer Sequences, 1964.

[20] N. Sloane. The On-line Encyclopedia of Integer Sequences, 1964.

[21] T. Walsh. Loop-free Sequencing of Bounded Integer Compositions. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 33, 2000.

[22] A. Williams. The greedy gray code algorithm. volume 8037, pages 525–536, 08 2013.

[23] K. Yamanaka, T. Aruchi, T. Hiriyama, and Y. Nishitani. Coding Ladder Lotteries. *AL-142*, 2012(10), 2012.

[24] K. Yamanaka, T. Hiriyama, and K. Wasa. Optimal Reconfiguration of Optimal Ladder Lotteries. *Proceedings of the 10th Japanese-Hungarian Symposium on Discrete Mathematics and Its Applications*, 2017.

[25] K. Yamanaka, T. Horiyama, T. Uno, and K. Wasa. Ladder-Lottery Realization. In *CCCG*, 2018.

[26] K. Yamanaka and N. Shin-Ichi. Efficient Enumeration of all Ladder Lotteries with K Bars. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E97.A:1163–1170, 06 2014.

[27] K. Yamanaka and N. Shin-Ichi. Enumeration, Counting, and Random Generation of Ladder Lotteries. *IEICE Transactions on Information and Systems*, E100.D(3):444–451, 2017.

[28] K. Yamanaka, N. Shin-Ichi, Y. Matsui, R. Uehara, and K. Nakada. Efficient Enumeration of all Ladder Lotteries and its Application. *Theoretical Computer Science*, 411(16):1714 – 1722, 2010.

[29] S. Zaks. A New Algorithm for Generation of Permutations. *BIT Numerical Mathematics*, 1(24), 1984.

Appendix A

Appendix

A.1 Table of $OptL\{\pi\}$ for all π of order 5

Permutation	number	k	$OptL\{\pi\}$
1 2 3 4 5	1	0	1
1 2 3 5 4	2	1	1
1 2 4 3 5	3	1	1
1 3 2 4 5	4	1	1
2 1 3 4 5	5	1	1
1 2 4 5 3	6	2	1
1 2 5 3 4	7	2	1
1 3 2 5 4	8	2	1
2 1 3 5 4	9	2	1
1 3 4 2 5	10	2	1
1 4 2 3 5	11	2	1
2 1 4 3 5	12	2	1
2 3 1 4 5	13	2	1
3 1 2 4 5	14	2	1
1 3 4 5 2	15	3	1
1 4 2 5 3	17	3	1
2 1 4 5 3	18	3	1
1 3 5 2 4	19	3	1
1 5 2 3 4	20	3	1
2 1 5 3 4	21	3	1
2 3 1 5 4	22	3	1
3 1 2 5 4	23	3	1
2 3 4 1 5	24	3	1
3 1 4 2 5	26	3	1
2 4 1 3 5	27	3	1
4 1 2 3 5	28	3	1
1 4 3 2 5	25	3	2
1 2 5 4 3	16	3	2
3 2 1 4 5	29	3	2
2 3 4 5 1	30	4	1
3 1 4 5 2	33	4	1
1 4 5 2 3	34	4	1
2 4 1 5 3	37	4	1
4 1 2 5 3	38	4	1
2 3 5 1 4	39	4	1
3 1 5 2 4	41	4	1
2 5 1 3 4	42	4	1
5 1 2 3 4	43	4	1
1 3 5 4 2	31	4	2
1 4 3 5 2	32	4	2
1 5 2 4 3	35	4	2

2 1 5 4 3	36	4	2
1 5 3 2 4	40	4	2
3 2 1 5 4	44	4	2
2 4 3 1 5	45	4	2
3 2 4 1 5	46	4	2
4 1 3 2 5	48	4	2
4 2 1 3 5	49	4	2
2 4 5 1 3	58	5	1
4 1 5 2 3	60	5	1
3 5 1 2 4	66	5	1
2 3 5 4 1	50	5	2
2 4 3 5 1	51	5	2
3 2 4 5 1	52	5	2
3 1 5 4 2	55	5	2
4 1 3 5 2	57	5	2
2 5 1 4 3	61	5	2
5 1 2 4 3	62	5	2
4 2 1 5 3	63	5	2
2 5 3 1 4	64	5	2
3 2 5 1 4	65	5	2
5 1 3 2 4	67	5	2
5 2 1 3 4	68	5	2
1 5 4 2 3	59	5	3
3 4 2 1 5	69	5	3
4 2 3 1 5	70	5	3
4 3 1 2 5	71	5	3
1 4 5 3 2	53	5	3
1 5 3 4 2	54	5	3
3 4 5 1 2	77	6	1
4 5 1 2 3	85	6	1
4 2 5 1 3	84	6	2
3 5 1 4 2	80	6	2
2 4 5 3 1	72	6	3
2 5 3 4 1	73	6	3
3 4 2 5 1	75	6	3
4 2 3 5 1	76	6	3
3 5 2 1 4	88	6	3
5 2 3 1 4	89	6	3
5 3 1 2 4	90	6	3
4 1 5 3 2	79	6	3
5 1 3 4 2	81	6	3
4 3 1 5 2	82	6	3
2 5 4 1 3	83	6	3
5 1 4 2 3	86	6	3
5 2 1 4 3	87	6	4
1 5 4 3 2	78	6	8
4 3 2 1 5	91	6	8
3 5 4 1 2	98	7	3
4 3 5 1 2	99	7	3
4 5 1 3 2	100	7	3
4 5 2 1 3	103	7	3
3 4 5 2 1	92	7	4
5 2 3 4 1	96	7	4
5 4 1 2 3	105	7	4
5 3 1 4 2	102	7	5
5 2 4 1 3	104	7	5
4 2 5 3 1	94	7	5
3 5 2 4 1	95	7	5
2 5 4 3 1	93	7	8
4 3 2 5 1	97	7	8
5 1 4 3 2	101	7	8

5 3 2 1 4	106	7	8
4 5 2 3 1	109	8	6
4 5 3 1 2	112	8	6
5 3 4 1 2	113	8	6
3 5 4 2 1	107	8	11
4 3 5 2 1	108	8	11
5 2 4 3 1	110	8	11
5 3 2 4 1	111	8	11
5 4 1 3 2	114	8	11
5 4 2 1 3	115	8	11
4 5 3 2 1	116	9	20
5 3 4 2 1	117	9	20
5 4 2 3 1	118	9	20
5 4 3 1 2	119	9	20
5 4 3 2 1	120	10	62

A.2 Code for CreateCanonical

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


//ladder data structure
typedef struct ladder
{
    int** ladder;
    int numRows;
    int numCols;
    int numBars;
} Ladder;

//initialize the ladder
Ladder* initLadder(int n);

//returns the coordinates of the bar or absence of a bar formed between x and y
int* GetCoordinates(int n, int x, int posX, int posY);
//creates a copy of a permutation
int* copyPerm(int* og, int start, int end);
//creates a copy of og permutation minus the element at index
int* removeElement(int* og, int index, int n);
//creates the canonical ladder
void CreateCanonical(Ladder* l, int* pi, int n);
//prints the ladder
void printLadder(Ladder *l);


//initializes the ladder data structure
//n is the length of a permutation
Ladder* initLadder(int n)
{
        Ladder* l = calloc(1, sizeof(Ladder));
        int nCols = n-1;
        int nRows = 2*(n-1) - 1;
        int nBars = 0;
        l->ladder = calloc(nRows, sizeof(int*));
        for(int i = 0; i < nRows; i++)l->ladder[i] = calloc(nCols, sizeof(int));
        l->numRows = nRows;
        l->numCols = nCols;
        l->numBars = nBars;
        return l;
}//end function

/**
        Returns the coordinates corresponding to a bar or absence of a bar
        formed between elements x and y which lies on the associated diagonal of 'x'.
        Parameter conditions: n is the size of a permutation, x is the element whose diagonal we are popula
                                                posY is the position of 'y' in pi. posX != posY and
```

```
    Post conditions: The (row, column) coordinates of the bar or absence of a bar corresponding to the two
**/

int* GetCoordinates(int n, int x, int posX, int posY){

        int* coordinates = calloc(2, sizeof(int));
        //if x is to the right of y, then they form the absence of an inversion. Use
        //equation 3.1.1 from Equation 3.1
        if(posX > posY){
                int row = (2*n - x - 1) - (x - posY) + 1;
                int col = (x - 1) - (x - posY) + 1;
                coordinates[0] = row ;
                coordinates[1] = col;
        }//end If
        //else if posX is < posY, then they form an inversion. Use equation 3.1.2 from Equation 3.1
        else{
                int row = (2*n - x - 1) - (x - posY);
                int col = (x - 1) - (x - posY);
                coordinates[0] = row;
                coordinates[1] = col;
        }//end else

        return coordinates;

}//end function

//creates a copy of a permutation from a start index to an end index
int* copyPerm(int* og, int start, int end){

        int size = (end - start);
        int* copy = calloc(size, sizeof(int));
        for(int i = start; i < end; i++){
                copy[i - start] = og[i];
        }
        return copy;
}

int* removeElement(int* og, int index, int n){

        int* leftPart = copyPerm(og, 0, index);
        int* rightPart = copyPerm(og, index+1, n);
        int size = n-1;
        int* newPerm = calloc(n-1, sizeof(int));
        for(int i = 0; i < index; i++)newPerm[i] = leftPart[i];
        for(int i = 0; i < n-index-1; i++)newPerm[index+i] = rightPart[i];
        free(leftPart);
        free(rightPart);
        return newPerm;
}

/**
Sets ladder l to the Canical ladder.
Parameter conditions: l is created using initLadder, pi is a permutation of order n
Post conditions: l is set to the canonical ladder corresponding to pi
**/
```

```
void CreateCanonical(Ladder* l, int* pi, int n)
{
        int x = n;
        int* pi2 = copyPerm(pi, 0, n);
        //from element x=n down to element x=2
        while(x > 1)
        {
                int index = -1;
                //get the index of element 'x' in pi
                for(int i = 0; i < x; i++){
                        if(pi2[i] == x){
                                index = i;
                                break;
                        }//end if
                }//end for
                //populate the associated diagonal of x with 1s and 0s.
                for(int i = 0; i < x; i++){
                        if(i != index){
                                int* coordinates = GetCoordinates(n, x, index, i);
                                int row = coordinates[0];
                                int col = coordinates[1];
                                if(i < index){
                    l->ladder[row][col] = 0;
                                }else{
                    l->ladder[row][col] = 1;
                                        l->numBars++;
                                }
                        }
                }//end for
                pi2 = removeElement(pi2, index, x);
        x--;
        }//end while
}

void printLadder(Ladder *l)
{

    int n = l->numCols + 1;
    int offset = 2*(n-1) - 1;
    printf("\n\n");
    for (int i = 0; i < offset; i++)
    {
        for (int j = 0; j < l->numCols; j++)
        {
            printf("|");
            if (l->ladder[i][j] == 1)
                printf("-----");
            else
                printf("     ");
        }
        printf("|");
        printf("\n");
    }
}
```

```
//main for testing
int main(int argc, char* argv[])
{
    //testing permutation
    int pi[] = {4,1,7,2,3,6,5};
    int n = 7;
    Ladder* l = initLadder(n);
    CreateCanonical(l, pi, n);
    printLadder(l);
}
```

A.3 Code for ModifiedSJT

```
//Gets the coordinates of the row and coulumn for ModifiedSJT
int* GetCoordinatesTwo(int* rX, int n, int x, int posX, int posY);


//modified SJT algorithm for listing Ln
void ModifiedSJT(Ladder* l, int x, int n, bool* direction, int* rX);

//used in ModifiedSJT to return the location of a 1 or 0 indicating the addition or removal of
//a bar between two adjacent elements, x>y
int* GetCoordinatesTwo(int* rX, int n, int x, int posX, int posY)
{
        int* coordinates = calloc(2, sizeof(int));
        //if posX is to the right of posY then x and y are currently
        //uninverted and will be inverted. Therefore, we add a bar along the associated diagonal of 'x'
        if(posX == posY + 1)
        {
                coordinates[0] = (2*n - x - 1) - (*rX)-1;
                coordinates[1] = (x-1) - (*rX)-1;
                (*rX)++;

        }
        //else if x is to the left of y by 1 position, then x and y are currently inverted and will become
        //uninverted. Therefore, we remove a bar along the associated diagnoal of 'x'
        else if(posX == posY - 1)
        {

                coordinates[0] = (2*n - x) - (*rX)-1;
                coordinates[1] = x - (*rX)-1;
                (*rX)--;

        }
        //else error occured
        else
        {
                printf("Error occurred\n");
                exit(0);
        }
```

```
        return coordinates;
}

/**
Modify the SJT algorithm to list Ln by adding or removing a bar
**/
void ModifiedSJT(Ladder* l, int x, int n, bool* direction, int* rX)
{
        if(x > n)
        {
                if(l->numBars == 0)printLadder(l);
                return;
        }
        for(int i = 0; i < x; i++)
        {
                if(i == 0)ModifiedSJT(l, x+1, n, direction, rX);
                else
                {
                        //adding to the associated diagonal of x
                        if(direction[x-1] == false)
                        {
                                int* coordinates = GetCoordinatesTwo(&(rX[x-1]), n, x, 1, 0);
                                //subtract 1 for 0 index
                                int row = coordinates[0];
                                int column = coordinates[1];

                                l->ladder[row][column] = 1;
                                l->numBars++;

                        }
                        //removing a bar from the associated diagonal of x
                        else{
                                int* coordinates = GetCoordinatesTwo(&(rX[x-1]), n, x, 0, 1);
                                int row = coordinates[0];
                                int column = coordinates[1];


                                l->ladder[row][column] = 0;
                                l->numBars--;
                        }
                        printLadder(l);
                        ModifiedSJT(l, x+1, n, direction, rX);


                }
        }
        direction[x-1] = !(direction[x-1]);
}


//testing main
int main(int argc, char* argv[])
{
```

```
    int n = 5;
    Ladder* l = initLadder(n);


        int* rX = calloc(n, sizeof(int));
        bool* direction = calloc(n, sizeof(int));
        for(int i = 0; i < n; i++){
                direction[i] = false;
                rX[i] = 0;
        }
        ModifiedSJT(l, 2, n, direction, rX);
}
```