Amidakuji: Counting, Listing and Optimization Algorithms

by

Patrick Di Salvo

A Thesis

presented to

The University of Guelph

In partial fulfilment of requirements
for the degree of

Masc

in

Computer Science

Guelph, Ontario, Canada

ABSTRACT

Amidakuji: Counting, Listing and Optimization Algorithms

Patrick Di Salvo
University of Guelph, 2020

Advisor:
Dr. Joe Sawada
Dr. Charlie Obimbo

Amtrak, or ladder-lotteries in English, are an abstract mathematical object which correspond to permutations. First written about in 2010, ladder-lotteries are a relatively new mathematical object. They are of interest to the field of theoretical computer science because of their similarities to other mathematical objects such as primitive sorting networks. This thesis provides an overview of ladder-lotteries along with solutions to three problems pertaining to ladder-lotteries; these problems are known as the counting problem, the listing problem and the minimal height problem. The solutions to these problems come in the form of novel algorithms, recurrence relations and formulas. The study of ladder-lotteries falls under the sub-discipline of theoretical computer science. The potential applications for ladder-lotteries is also discussed in this thesis, along with the similarities between ladder-lotteries and other mathematical objects.

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 [**?**]. 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. The more bars there are the more complicated (and fun) the Amidakuji is. No two endpoints of two bars can be touching. 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. The prize at the bottom of the ladder lottery is their prize [**?**]. See Figure **??** for an example of a ladder lottery.

The game of Amidakuji has an interesting history. In Japanese, 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 the his Western Paradie. 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 [**?**]. It was in Japan where the game Amidakuji began. Amidakuji began in Japan in the Muromachi period, which spanned from 1336 to 1573 [**?**].

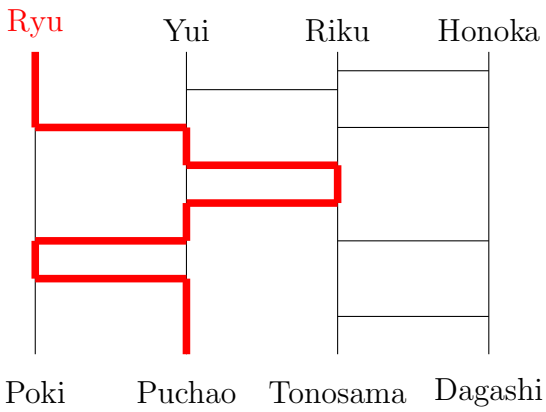


Figure 1.1: A ladder lottery where Ryu gets Puchao, Yui gets Dagashi, Riku gets Tonosama and Honoka gets Poki. You can see that Ryu's path is marked by red bars.

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, which is why the game is called Amidakuji. Kuji is the Japanese word for lottery. Hence the name of the game being Amidakuji.In English, Amidakuji traslates to ladder lottery.

An interesting property about a ladder lottery is it is associated with a *permutation*. A *permutation* is a unique ordering of objects [**?**]. For the purposes of this paper, the objects of a permutation will be integers ranging from $1, 2, \ldots, N$. Consider a permutation $\pi = (p_1, p_2, \ldots, p_n)$. An *inversion* is a relation between two elements in π, p_i and p_j, such that if $p_i > p_j$ and $i < j$ then p_i and p_j form an inversion. For example, given $\pi = (4, 3, 5, 1, 2)$, its set of inversions $= \{(4, 3), (4, 1), (4, 2), (3, 1), (3, 2), (5, 1), (5, 2)\}$. An *adjacent transposition* is defined as a swap of two adjacent elements in a permutation. A ladder lottery associated with a permutation is characterized as follows:

1. The n elments 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.

3. The ladder lottery is represented as a two dimensional array.

4. A bar in the ladder lottery performs an adjacent transposition on two inversions in π, thus transforming π into some new permutation π_n. The *state of* π is defined as π having undergone k adjacent transpositions.

5. A *column* is defined as a gap between two lines in the ladder lottery; the number of columns is equal to $n - 1$.

6. Each *row* in the ladder lottery contains l bars, where $l \geq 1$. Hence, each row represents a change of state in π size l.

7. Rows are counted top to bottom. The final row in the ladder lottery is the row such that the bars in said row perform adjacent transpositions on adjacent inversions in the current state of π resulting in the sorted permutation.

Let k denote the number of inversions for some arbitrary π. An *optimal ladder lottery* is a special case of ladder lottery in which there are k bars in the ladder; one for each *inversion* in π [**?**]. An optimal ladder lottery sorts π using exactly k bars. 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. The number of bars in an optimal ladder lottery is the lower bound for the number of bars in a ladder lottery required to sort π. To see an example of two ladder lotteries associated with $\pi = (4, 3, 2, 1)$, one optimal and one non-optimal, please refer to Figure **??**.

Figure 1.2: Two ladders for the permutation (4, 3, 2, 1). The left ladder is an optimal ladder and the right ladder is not. Therefore the left ladder belongs to $optL\{(4,3,2,1)\}$. The bold bars in the right ladder are redundant, thus the right ladder is not optimal

Lemma 1.0.1 *Every permutation has a finite set of optimal ladder lotteries associated with it.*

Proof. Let k be the number of bars in a ladder such that k equals the number of invserions in π. Assume there is one row in the ladder for each of the k bars; thus there are k rows in the ladder. Let π be of order n. Let $n-1$ be the number of columns in the ladder. Hence, the dimensions of the ladder associated with π are $(n-1)$ by (k). The total number of ladders with $(n-1)$ columns and k rows is $\binom{(n-1)(k)}{k}$ which is finite. For each optimal ladder lottery with $(n-1)$ columns and k bars, their aggregation forms a subset of all $\binom{(n-1)(k)}{k}$ ladder lotteries. Therefore, the set of all optimal ladder lotteries is finite. Let this set be known as $OptL\{\pi\}$. The ladders that are in $OptL\{\pi\}$ are optimal ladders which again are defined as follows:

- Each ladder has the minimal number of bars required to sort π which equals k.
- Each ladder sorts π using exactly k bars.

Thus, the set of finite ladder lotteries associated with π is finite. QED. To see the $OptL\{(3,4,2,1)\}$ please refer to Figure **??**. □

For the remainder of this paper, only optimal ladder lotteries will be discussed, with one exception. Therefore, when the term ladder lottery is used, assume optimal

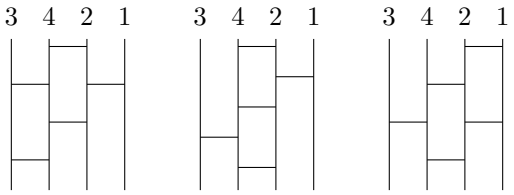


Figure 1.3: All the minimal ladders in $OptL\{(3,4,2,1)\}$

ladder lottery unless otherwise stated.

1.1 Thesis Statement

This thesis considers two problems related to ladder lotteries. The first problem is the so called Canonical Ladder Listing Problem. This problem asks, given all $n!$ permutations of order n, is there an efficient algorithm for listing all $n!$ ladders? In other words, is there an easy way to transition from one ladder to the next until all $n!$ ladders have been listed? Efficiency is defined as relocating the minimal number of bars in $ladder_i$ to get to $ladder_{i+1}$ or add/removing the minimal number of bars in $ladder_i$ to get to $ladder_{i+1}$. This thesis provides two such algorithms for solving the Canonical Ladder Listing Problem. The first is a modification of the Steinhaus-Johnson Trotter Gray Code. The second is a modification of the Effler-Rusky algorithm.

The second problem is the so called Minimum Height Problem which asks, given all the ladders in $OptL\{\pi\}$, which ladder(s) are the shortest? That is to say, which ladders have the least number of rows? Furthermore, given some arbitrary π, is there an efficient algorithm to create a ladder for π with minimal height? Efficiency is defined as $O(n^2)$. This thesis provides an upper and lower bound for the minimal height of ladders along with an efficient heuristic algorithm for creating a ladder with minimal height. The heuristic algorithm uses the properties of bar compression and zig-zagginess in order to create a ladder with minimal height.

1.2 Overview of Thesis

This thesis is broken down into several sections. Firstly, a literature review of ladder lotteries will be provided. The literature review focuses on solved problems pertaining to ladder lotteris along with the commonalities between ladder lotteries and other mathematical objects. Following the literature review, two chapters pertaining to the two problems will be provided. In each of these chapters there is an introduction to the problem, a methodology section, a results section and a conclusion section. The introduction section introduces the problem to the reader, providing the necessary definitions and concepts. The methodology sections contain the algorithms and formulas used to solve the respective problems. Following the methodology sections, the results generated by the algorithms will be presented. In the results sections there will be proofs and formulas for certain propositions made in regards to the respective problems. Following the results section, an analysis of the results will be presented along with a summary of future work. In this section, the failures and successes of this research will be analyzed. There will also be commentary on open (unsolved) problems related to ladder lotteries and a discussion of how research on ladder lotteries could be used in other fields. Finally, a conclusion that summarizes the thesis will be provided.

1.3 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 [**?**]. In this paper, the authors provide an algorithm known as $FindAllChildren$ which generates $OptL\{\pi\}$; the details of the algorithm are discussed in chapter two. Since this paper, a number of problems related to ladder lotteries have been solved. In Table **??** you

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 $OptL\{\pi\}$	Efficient Enumeration of Ladder Lotteries and its Application
Ladder Lottery Realization Problem	Determines time complexity for creating a ladder lottery given a multi-set of bars	Ladder Lottery Realization
The Reconfiguration Problem	Determine the length of the path between two ladders in $OptL\{\pi\}$	Optimal Reconfiguration of Optimal Ladder Lotteries
Enumeration Problem given n, k	Generates all ladders with n lines and k bars; includes non-optimal ladders	Efficient Enumeration of all Ladder Lotteries with K Bars
The Coding Problem	Provides a binary string encoding for a ladder lottery	Coding Ladder Lotteries
Counting and Random Generation Problem	Provides a solution for counting ladder lotteries of a given type as well as randomly generating a ladder lottery	Enumeration, Counting, and Random Generation of Ladder Lotteries

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

Chapter 2

Background

In the background section we will provide a more comprehensive analysis of the existing research surrounding ladder lotteries. We will also go through the foundational concepts, algorithms and proofs that act as the backbone for the results in chapters three and four. The remainder of this paragraph will detail the fundamental data structures and auxiliary functions used for this thesis. We have already defined π in the introduction and we have already discussed a ladder lottery as a concept. Let n be the number of elements in π. Let the *ladder data structure*, or *ladder* for short, be a two dimensional array with $0 \leq k \leq \binom{n}{2}$ rows and $n-1$ columns. The number of rows in *ladder* is zero if π is in ascending order. The number of rows in ladder is $\binom{n}{2}$ if π is in descending order. The number of columns is always $n-1$ seeing as the columns represent gaps between lines. Thus, given n lines, there are $n-1$ columns in ladder. Let $\tau(p_i, p_j)$ be an adjacent transposition of an adjacent inversion $(p_i, p_j) \in \pi$. Throughout the thesis a number of algorithms are presented. Many of these algorithms use the following auxilary functions:

1. PRINT(X): Prints x.
2. SWAP(X,Y): Swaps x and y.
3. SORT(X): Sorts x in ascending order.
4. MAX(X): Returns the maximum element in x.
5. 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 [**?**]. 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. It is in these papers that the aforementioned problems related to ladder lotteries are solved. Thus, a comprehensive analysis of these papers is required for getting the full breadth of the literature surrounding ladder lotteries. This thesis is also heavily influenced by Efficient Enumeration of Ladder Lotteries and its Application. Some of the foundational concepts, algorithms and proofs will be discussed in the subsection pertaining to Efficient Enumeration of Ladder Lotteries and its Application seeing as the background information for this thesis is intricately related to the findings in the paper Efficient Enumeration of Ladder Lotteries and its Application.

2.1 Efficient Enumeration of Ladder Lotteries and its Application

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

Algorithm 1 The algorithm for listing $OptL\{\pi\}$.

1: **function** FindAllChildren($ladder$, $cleanLevel$, n)
2: $currentRoute \leftarrow n$
3: **while** $currentRoute \geq cleanLevel$ **do**
4: going top left to bottom right
5: **for** $bar \in currentRoute$ **do**
6: $row \leftarrow$ row of bar in $ladder$
7: $col \leftarrow$ col of bar in $ladder$
8: $lowerNeighbor \leftarrow ladder[row - 1][col]$
9: **if** $lowerNeighbor$ is right swappable **then**
10: RightSwap($ladder$, bar, $lowerNeighbor$)
11: FindAllChildren($ladder$, $y + 1$, n)
12: LeftSwap(ladder, bar, lowerNeighbor)
13: **end if**
14: **end for**
15: $currentRoute \leftarrow currentRoute - 1$
16: **end while**
17: $currentRoute \leftarrow cleanLevel - 1$
18: **for** $bar \in currentRoute$ **do**
19: $row \leftarrow$ row of bar in $ladder$
20: $col \leftarrow$ col of bar in $ladder$
21: $lowerNeighbor \leftarrow ladder[row - 1][col]$
22: **if** lowerNeighbor is right swappable **and** is the rightmost bar of $currentRoute - 1$ **then**
23: RightSwap($ladder$, bar)
24: FindAllChildren($ladder$, $cleanLevel$, n)
25: LeftSwap($ladder$, bar)
26: **end if**
27: **end for**
28: **end function**

FindAllChildren is the first algorithm for generating $OptL\{\pi\}$. FindAllChildren enumerates $OptL\{\pi\}$ as a tree of ladders. The parent-child relation in the tree is based on a *local swap operation* which corresponds to a braid relation and is a local modification of *ladder* as shown in Figure **??**. To get from the parent to the child a right swap operation is performed. To get from the child to the parent a left swap operation is performed. Given an arbitrary bar, x, it can be right swapped if and only if there are two bars, y, z where $y \neq z$ such that all the following conidtions are met [**?**].

- The left end point of z is directly above the left end point of x.
- The left end point of y is directly above the right end point if x.
- The right end point of z is directly above the left end point of y.

Given an arbitrary bar, x, it can be left swapped if and only if there are two bars, y, z where $y \neq z$ such that the following conditions are met [**?**].

- The right end point of z is directly below the right end point of x.
- The right end point of y is directly below the left end point if x.
- The left end point of z is directly below the right end point of y.

In the left ladder in Figure **??** bar $x = (3, 1)$, bar $y = (5, 1)$ and bar $z = (5, 3)$. Bar x can be right swapped seeing as the three conditions for performing a right swap operation are met. In the right ladder in Figure **??** bar $x = (3, 1)$, bar $y = (5, 1)$ and bar $z = (5, 3)$. Bar x can be left swapped seeing as the three conditions for performing a left swap operation are met.

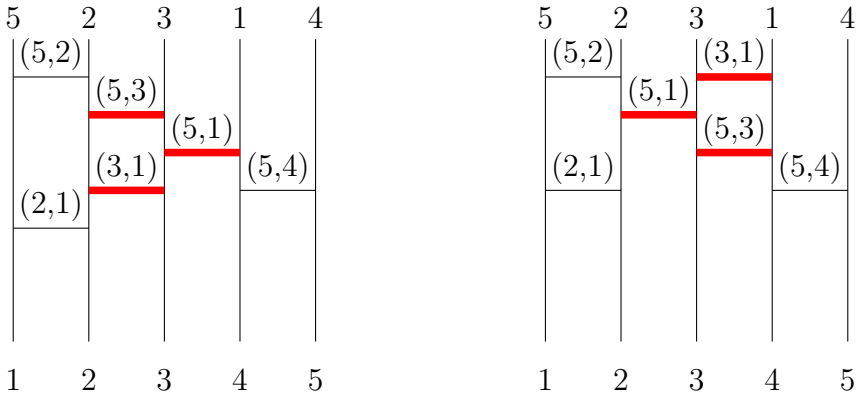


Figure 2.1: Example of a local swap operation. When a right swap operation is permformed on the left ladder, the result is the right ladder. When a left swap operation is permformed on the right ladder, the result is the left ladder.

FindAllChildren was used in this thesis to generate the sample data that aided in finding solutions for the Gray Code Problem in chapter three and the Minimum Height Problem in chapter four. Therefore, this algorithm is paramount for the findings of this thesis. To see the tree structure generated by FindAllChildren for $OptL\{(4,3,2,1)\}$ please refer to Figure **??**. The paper also presents the number of

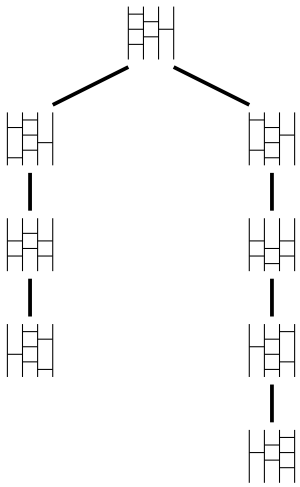


Figure 2.2: The tree structure of $OptL\{(4,3,2,1)\}$ generated by FindAllChildren

ladder lotteries in $OptL\{(11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)\}$ which is $5, 449, 192, 389, 984$ [**?**]. FindAllChildren is based on several key concepts. One of which is the local swap operation which has already been discussed. The next fundamental concept is the *route* of an element, which is 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. For every bar, two elements cross the bar, therefore the bar is associated with the route of the greater of the two elements [**?**]. In Figure **??** the route of element 4 is the sequence of bars $(4, 1), (4, 2), (5, 4)$. To see the route of an element please refer to Figure **??**. When a right swap operation occurs, the bar of the route associated with a lesser element is swapped above two bars of a route associated with a greater element. For example, in Figure **??** in the right ladder, the bar $(3, 1)$ which is associated with the route of element 3 is right swapped above the two bars $(5, 1)$ and $(5, 3)$ both of which are associated with the route of 5.

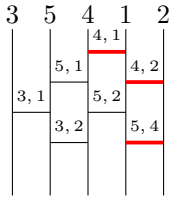


Figure 2.3: The route of element 4=(4,1),(4,2),(5,4).

The *clean level* is defined as one more than the largest element associated with any bar that has undergone a right swap operation. For example, if the largest element associated with a bar that has undergone a right swap operation is element 4, then the clean level is $5 = 4 + 1$. Please refer to Figure **??** for an example of the clean level. If no bars have been right swapped, then the clean level is 1. If the largest element to have undergone a right swap operation is the maximal element in π, then the clean level is $max + 1$.

Each $OptL\{\pi\}$ has a unique ladder with a clean level of one. This ladder is known as the *root ladder*. To see the root ladder of $OptL\{(4, 5, 6, 3, 1, 2)\}$ please refer

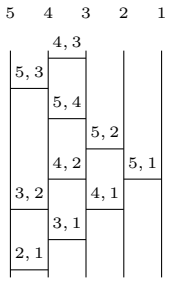


Figure 2.4: A ladder with a clean level of 6. The largest element whose associated bars have undergone a right swap operation is 5

to figure Figure **??**. To see a proof for the uniqueness of the root ladder please refer to Theorem **??**.

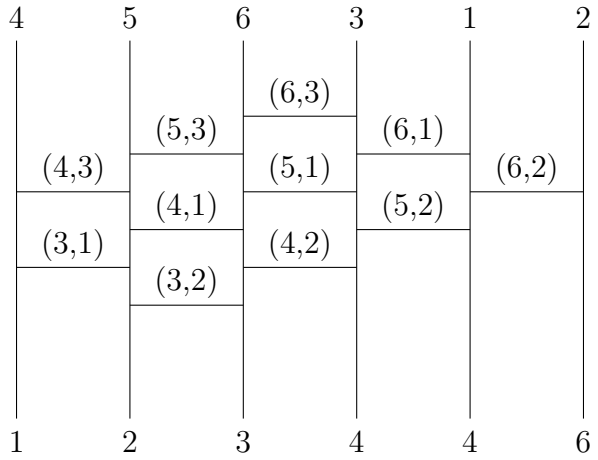


Figure 2.5: The root ladder for $OptL\{(4, 5, 6, 3, 1, 2)\}$. Notice how no bar associated with a lesser element is above bars associated with a greater element. Therefore, the clean level of this ladder is 1, thus making it the root ladder.

Theorem 2.1.1 *The root ladder is unique to* $OptL\{\pi\}$.

Proof. Since the parent to child relation of the tree structure generated by FindAllChildren is based on a right swap operation and the child to parent relation of the tree structure generated by FindAllChildren is based on a left swap operation, then there exists a ladder in the tree structure such that no bars have been right

swapped. If this were not the case, then we affirm an infinite regress. Thus, there exists a ladder in the tree structure that is not derived from a right swap operation. This ladder is the root ladder seeing as once a right swap operation has occurred, the clean level of the ladder ceases to equal one. But we have already stated that the clean level of the root ladder is one. Therefore, this ladder must be the root ladder. To prove that the root ladder is unique, consider that we are dealing with a tree structure. Trees have a single root. The root in the tree structure for $OptL\{\pi\}$ is the ladder that has not undergone any right swap operations. Thus, the root ladder is the unique root of the tree structure generated by FindAllChildren. QED.

□

Corollary 2.1.2 *If* $|OptL\{\pi\}| = 1$ *then the ladder in* $OptL\{pi\}$ *must be the root ladder.*

Proof. If there is only one ladder in the set, then that means no bars have been swapped in said ladder. Thus, it must be the root ladder. QED. □

The authors provide a good description of the foundational concepts required to generate $OptL\{\pi\}$ using FindAllChildren. However, the authors do not provide an algorithm for performing a local swap operation nor do they provide an algorithm for creating the root ladder. In order to effectively implement FindAllChildren, algorithms for creating the root ladder and performing local swap operations are necessary. Seeing as the root ladder cannot be created by performing a right swap operation, a separate algorithm is required for creating the root ladder. This thesis provides novel algorithms for creating the root ladder and for performing a local swap operation. Please refer to Algorithm **??** to see the algorithm for creating the root ladder. The initial conditions of Algorithm **??** are the following. Let *ladder* be initialized to an empty ladder. Let n be initialized to $\textsc{Max}(\pi)$. Let row be initialized to one. π is an arbitrary permutation of order n.

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

```
1: function CreateRoot(ladder, π, n, row)
2:     if n = 1 then
3:         return
4:     end if
5:     x ← index of Max(π)
6:     for i ← x + 1 to n do
7:         if p_x > p_i and x < i then
8:             column ← i
9:             if this is the first bar to be added then
10:                while bar cannot be added do
11:                    row ← row + 1
12:                end while
13:                ladder[row][column] ← 1
14:            else
15:                row ← row + 1
16:                ladder[row][column] ← 1
17:            end if
18:        end if
19:    end for
20:    π ← π − largestElement
21:    CreateRoot(ladder, π, n − 1, row ← 1)
22: end function
```

Let *ladder* be a two dimensional array, let π be the current state of the permutation, let *row* be the current row in the ladder. First, the index of the largest element in π is assigned to x. Once found, the algorithm loops from $x+1$ to n. If $p_x > p_i$ then a bar is to be added to *ladder* at $row, column = i$. There are two cases for calculating the *row*.

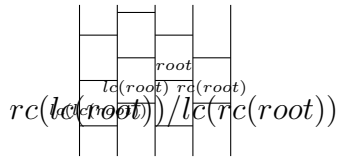
Case 0: *First bar is being added*
This is the first bar to be added to the route of the largest element. *row* is incremented until a bar can be added to *ladder* at *row* and *column*. A bar can be added if neither of its endpoints are touching the endpoints of any other bar.

Case 1: *Second or greater bar is being added*
If this is second or greater bar to be added to the route of the largest element in π then $row \leftarrow row + 1$. Once all the bars for the route of the largest element have been added, the largest element from π is removed; $\pi \leftarrow \pi -$ MAX(π). $n \leftarrow n-1$ and $row \leftarrow 1$. The algorithm then makes a recursive call.

Lemma 2.1.3 *The time complexity for CreateRoot is* $O(n^3)$

Proof. The outer for-loop of the function runs from some arbitrary index to n on each function call. The inner for loop runs at most $2(n-1)-1$ times which is reduced to n. Thus, we get $O(n^2)$. The following recursion holds, $CreateRoot(n-k) = CreateRoot(n-k+1) + O((n-k)^2) = (CreateRoot(n-k+2) + O((n-k+1)^2)) + O((n-k)^2) \ldots$. Which is reduced to $O(n(n+1)(2n+1)/6) = O(n^3)$. QED. □

The following algorithms are used to perform a local swap operation: Algorithm **??**, Algorithm **??** and Algorithm **??**. Let *ladder* be initialized to an empty ladder. Let *bar* be the bar to be right or left swapped in the *ladder*. In Algorithm **??** *offset* is initialized to 2 when right swapping and initialized to -2 when left swapping. In Algorithm **??** *index* is initialiezed to 1 when right swapping and -1 when left swapping. Let the *right child* of some arbitrary bar w, $rc(w)$ for short, be the bar one row below, and one column to the right of w. Let the *left child* of some arbitrary bar w, $lc(w)$ for short, be the bar one row below, and one column to the left of w. Let the *right sibling* of w, $rs(w)$ for short, be defined as the bar on the same row as some arbitrary bar w and two columns to the right of bar w. Let the *upper neighbor* of w, $un(w)$ for short, be the bar that is two rows above the row of w and is in the same column as w. Let the *right neighbor* of w, $rn(w)$ for short, be the bar that is one row above w and one column to the right of w. Let the lower neighbor of w, $ln(w)$ for short be the bar that is two rows below w and in the same column as w. Let the left neighbor of w, $lfn(n)$ for short, be the bar one row below and one column to the left of w. Let the *sub-ladder* be a subset of bars such that each bar in the sub-ladder is a left or right child of some other bar other than the root of the sub-ladder. For an example of a sub-ladder please refer to Figure **??**.



The circled bars are the bars representing a subladder.

Algorithm 3 Perform a right swap operation on a bar

```
1: function RightSwap(ladder, bar)
2:     row ← bar's row in ladder
3:     col ← bar's column in ladder
4:     upperNeighbor ← un(bar)
5:     rightNeighbor ← rn(bar)
6:     if col <= n − 3 then
7:         subLadderRoot ← rs(bar)
8:         ShiftSubLadder(ladder, subLadderRoot, 2, 1)
9:     end if
10:    Swap(upperNeighbor, ladder[row + 1][col + 1])
11:    Swap(bar, rightNeighbor)
12: end function
```

Algorithm 4 Perform a left swap operation on a bar

```
1: function LeftSwap(ladder, bar)
2:     row ← bar's row in ladder
3:     col ← bar's column in ladder
4:     lowerNeighbor ← ln(bar)
5:     leftNeighbor ← lfn(bar)
6:     if col < n − 1 then
7:         subLadderRoot ← rc(lowerNeighbor)
8:         ShiftSubLadder(ladder, leftSibling, −2, −1)
9:     end if
10:    Swap(lowerNeighbor, ladder[row − 1][col − 1])
11:    Swap(bar, leftNeighbor)
12: end function
```

The way that the aforementioned three algorithms work in order to complete a local swap operation is as follows. When performing a right swap operation, RightSwap takes the current bar, x, and gets its upper neighbor z and its right neighbor y; x, z and y meet the criteria for performing a right swap operation. Then, RightSwap calls ShiftSubLadder with the *offset* value of 2 and the *index* value of one. Algorithm ShiftSubLadder ensures that the bottom right sub-ladder, beginning at the right sibling of $x/rs(x)$, is shifted down the ladder such that when the right swap operation is performed, the root of the sub-ladder is the right child of z. When a right swap operation is about to occur, bar z will be moved from its current row and column to its current row + 3 and its current column +1. Once the right swap operation is performed, $rs(x)$ becomes $rc(z)$. y and x are swapped. Then the function is complete.

The left swap operation reverses the resulting ladder from the right swap operation.

Therefore, one can derive the left swap operation and the shift required for left swapping by deriving them from the right swap operation and the shift required for the right swap operation. To see an example of Algorithm **??** in conjunction with Algorithm **??** please refer to Figure **??**.

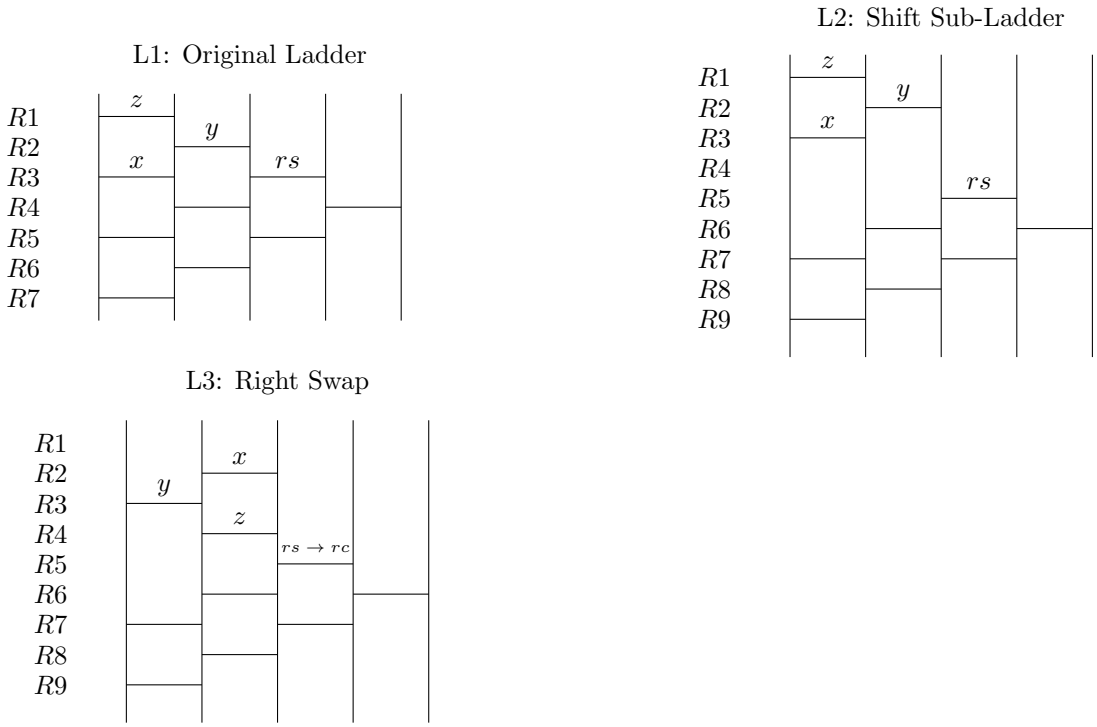


Figure 2.6: x, y, z to be right swapped. rs is the right sibling; the root bar of the right sub-ladder. Going right to left, top to bottom. L1=original ladder, L2=shifting the right sub-ladder down two rows. L3 = right swap on x, y, z

2.2 Ladder Lottery Realization

In the paper Ladder Lottery Realization, 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 appears 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 *non-optimal* 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 **??**.

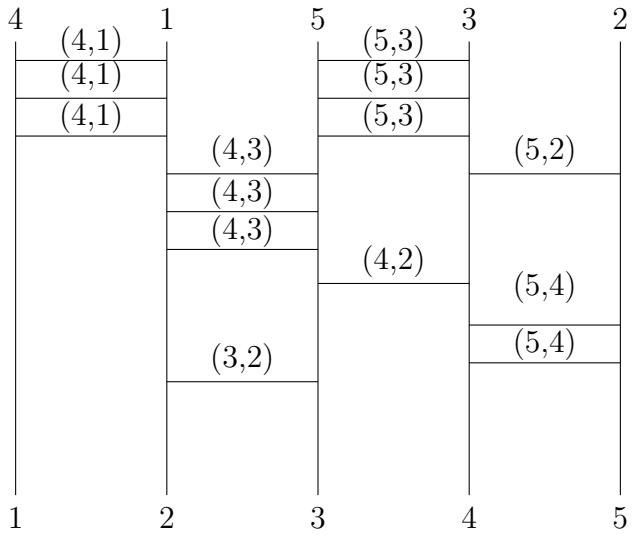


Figure 2.7: An affirmative solution to the Ladder Lottery Realization Problem given a starting permutation $(4, 1, 5, 3, 2)$ and the multi set of bars $\{(4,1)^3, (4,3)^3, (4,2)^1, (5,4)^2, (5,3)^3, (5,2)^1, (3,2)^1\}$

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 One-In-Three 3SAT problem is a problem with a given a set of variables, X, a set of *disjunctive clauses*, C, which are disjunctive expressions over literals of X. Each clause in C must contain three literals, then there is a truth assignment for X such that each clause in C has exactly one true literal [**?**]. For example, let $X = \{p, q, r, s, t\}$ and let $C = \{C_{p,q,s}, C_{r,q,s}C_{p,s,t}, C_{r,t,q}\}$, the question is whether it is possible for each clause to have exactly one true literal. The answer in this case is yes. If $p = T$, $r = T$, $q = F$, $s = F$ and $t = T$ then all the clauses in C have exactly one true literal. The authors reduce the ladder lottery realization problem to the One-In-Three 3SAT problem by devising four gadgets [**?**]. The result of the reduction is that the arbitrary starting permutation is equivalent to X in the One-In-Three 3SAT problem and the multi-set of bars is equivalent C in the One-In-Three 3SAT problem [**?**].

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, written by Horiyama, Wasa and Yamanaka, the authors provide a polynomial solution to the *minimal reconfigu-*

ration problem which states that given two ladder is $OptL\{\pi\}$, L_i and L_m, what is the minimal number of swap operations to perform that will transition from L_i to L_m [**?**]? The authors answer the question based on the local swap operations previously explained along with some other concepts. The first of these concepts is termed the *reverse triple* [**?**]. Basically, 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 [**?**]. The second of the concepts is the *improving triple* [**?**]. The improving triple is 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 $OptL\{\pi = (N, N-1, \ldots, 1)\}$ that have the upper bound for the minimal length reconfiguration sequence [**?**]. These ladders are the root ladder and *terminating ladder* in $OptL\{\pi = (N, N-1, \ldots, 1)\}$ that have a minimal reconfiguration sequence equal to the upper bound. The terminating ladder in $OptL\{\pi = (N, N-1, \ldots, 1)\}$ is defined as the ladder such that every possible right swap operation has been per-

formed. The length of the reconfiguration sequence between the root ladder and terminating ladder in $OptL\{\pi = (N, N-1, \ldots, 1)\}$ is $N\binom{(N-1)}{2}$ [**?**]. This is because the number of reverse triples between the root ladder and the terminating ladder in $OptL\{\pi(N, N-1, \ldots, 1)\}$ is equal to $N\binom{(N-1)}{2}$. Thus, in order to reconfigure the root to the terminating ladder, or vice versa, each reverse triple between them must be improved by applying one improving triple.

2.4 Efficient Enumeration of all Ladder Lotteries with K Bars

In Efficient Enumeration of all Ladder Lotteries with K Bars, written by Nakano and Yamanaka, the authors apply the same algorithm used in Efficient Enumeration of Optimal Ladder Lotteries and its Application for generating all ladder lotteries with k bars where the number of inversions in $\pi \leq k \leq +\infty$. In other words, the authors use the algorithm in Efficient Enumeration of Optimal Ladder Lotteries and its Application for generating non-optimal ladders [**?**].

2.5 Coding Ladder Lotteries

In the paper Coding Ladder Lotteries, written by Aiuchi, Yamanaka, Hirayama and Nishitani, the authors provide three methods to encode ladder lotteries as binary strings [**?**]. Coding discrete objects as binary strings is an appealing theme because it allows for compact representation of them for a computer [**?**].

2.5.1 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 l be a ladder lottery for some arbitrary permutation π of order n. The route of element p_i is encoded by keeping in

mind p_i crosses bars in its route going left zero or more times and crosses bars in its route going right zero or more times [**?**]. The maximum number of bars p_i can have is $n-1$, therefore the upper bound for the number of left/right crossings for p_i is $n-1$ [**?**]. Let a left crossing be denoted with a $'0'$ and let a right crossing be denoted with a $'1'$. Let $C(p_i)$ be the route encoding for the i^{th} element in π. To construct $C(p_i)$, append 0 and 1 to each other representing the left and right crossings of p_i from the top left to bottom right of the ladder [**?**]. If the number of crossings for p_i is less than $n-1$, append $0s$ to the encoding of the route of p_i until the encoding is of length $n-1$ [**?**]. Let $RC(l)$ be the route encoding for some arbitrary ladder in $OptL\{\pi\}$. $RC(l)$ is $C(p_1), C(p_2), \ldots C(p_n)$. For an example of the route encoding for the root ladder of $(3,2,5,4,1)$ refer to Figure **??**. In **??** you will see that $C(p_1)$ is 11$\underline{00}$. Underlined $0s$ are the $0s$ added to ensure the length of $C(p_1)$ is $n-1$. Since the length of $C(\pi)$ is $n-1$ and the number of elements in π is n then the length of $RC(l) = n(n-1)$. Hence the number of bits needed for $RC(l)$ belongs to $\mathcal{O}(n^2)$.

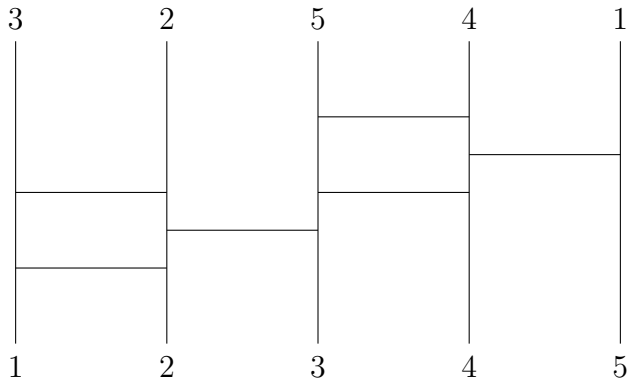


Figure 2.8: The route encoding for the following ladder lottery is

11$\underline{00}$10$\underline{01}$10$\underline{00}$1$\underline{00}$0000

2.5.2 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. Given the ladder in Figure **??**, $LC(3)$ is $001\underline{0}$. The $\underline{0}$ denotes the end of the line. Let $LC(l)$ be the line encoding for some arbitrary ladder, then $LC(l) = LC(1), LC(2), \ldots LC(n)$. Let $l(4, 2, 3, 1)$ refer to the ladder in Figure **??**, then $LC(l(4, 2, 3, 1)) = 11\underline{0}010\underline{0}110\underline{0}010\underline{0}0\underline{0}$.

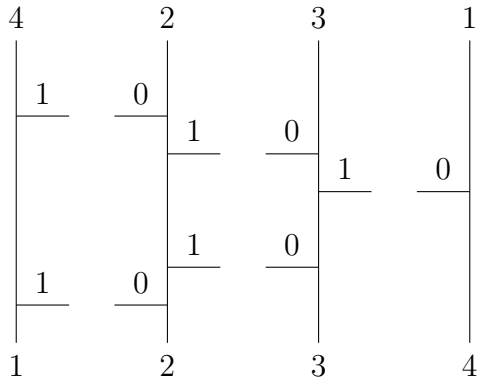


Figure 2.9: $LC(l(4, 2, 3, 1)) = LC(i = 1) = 11\underline{0}LC(i = 2) = 0110\underline{0}LC(i = 3) = 010\underline{0}LC(i = 4) = 0$

In order to reconstruct l from its $LC(l)$, or in other words decode $LC(l)$ it is important to recognize that the first line only has left endpoints attached to it [**?**]. Since left end points are encoded as a 1 then it is guarenteed that the first 0 represents the end of line 1. Secondly, the last/nth line has only right end points attached to it. Therefore $LC(i = n)$ will only have $0s$. Therefore, $LC(i = n)$ does not require a

terminating 0. Thirdly, for any line $i+1$, if line $i+1$ has a 0 then there must be a corresponding 1 in line i. That is to say, if the right end point of a bar is on line $i+1$ then that same bar must have a left endpoint on line i. To decode $LC(l)$ start by decoding line 1. The line will contain 0 or more left end points. To decode $LC(i+1)$ where $i+1 > 1$, go to $LC(i)$ and match each 1 in $LC(i)$ with a 0 in $LC(i+1)$. Let k = the number of $1s$ in $LC(i)$. Let j = the number of $0s$ in $LC(i+1)$ then $k = j - 1$; due to the last 0 in $LC(i+1)$ denoting the end of line $i+1$. Intuitively, this means match every left end point of a bar in line i with a right end point in line $i+1$. The last 0 represents the end of line $i+1$. For an example of a full decoding of $LC(l(4,2,3,1))$ please refer to Figure **??**.

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.

2.5.3 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. These three improvements can be combined to really help improve the line based encoding's space efficiency [**?**].

2.5.3.1 Improvement 1

Since the *nth* line has only right endpoints attached to it, then 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 $LC(i = n-1)$. Thus, there is no need for $LC(i = N)$ [**?**]. The encoding with improvement one for the ladder in Figure **??** is 11$\underline{0}$0110$\underline{0}$010.

2.5.3.2 Improvement 2

Improvement two is based off of the fact that for any 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.

Lemma 2.5.1 *There are three possible cases for the placement of x and y in some arbitrary ladder from $OptL\{\pi\}$. The first case is that there is at least one other bar, z, with a right end point, r_z between l_x and l_y on line i. The second case is that there is at least one other bar z, with a left end point, l_z, between r_x and r_y on line $i+1$. The third case is that there is at least one bar, z, with a right end point, r_z, between l_x and l_y on line i and there is at least one other bar, $z\prime$ with a left end point, $l_{z\prime}$, between r_x and r_y on line $i+1$ [?]. For an example of all three cases refer to Figure* **??**

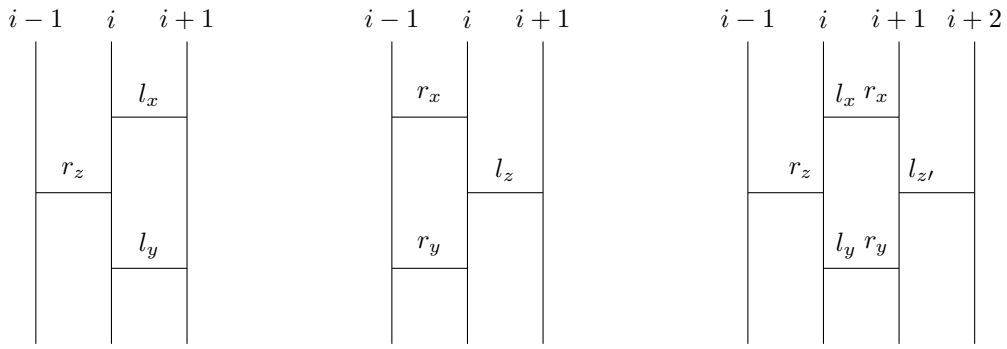


Figure 2.10: Three examples of the three cases for the placement of bars x and y in a ladder lottery

Proof. Suppose that none of the above cases hold. Let l be an optimal ladder lottery with bars x and bar y. If none of the cases hold then x and y are directly above/below each other without the endpoint of some third bar z between l_x and l_y or between r_x and r_y. Let x be the bar for the inversion of two elements p_i and p_j in π. As p_i and p_j travel through the ladder they will cross each other at bar x; thus uninverting them. Since bar y is directly below bar x, then p_i and p_j will cross bar y

thus re-inverting them. Therefore, there will need to be a third bar that uninverts p_i and p_j a second time. Since this third bar is redundant, l is non-optimal which is a contradiction. Let x be a bar for two elements in π, p_i and p_j such that p_i and p_j do not form an inversion. Then x will transpose p_i and p_j and y will transpose them a second time. Thus making both x and y redundant bars which is also a contradiction. Therefore one of the above cases must hold. □

Knowing that one of the three above cases must hold is beneficial for improving the line-based encoding. If l_x and l_y on line i have no r_z between them, then there must be at least one $l_{z'}$ between r_x and r_y on line $i+1$. Since a left endpoint is encoded as a 1 and a right endpoint is encoded as a 0, a 1 can be omitted for the encoding of line $i+1$ if l_x and l_y have no r_z between them on line i [**?**]. That is to say, 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. The line encoding with improvement two for the ladder in Figure **??** is $11\underline{0}010\underline{0}00\underline{0}0$.

2.5.3.3 Improvement 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. If you refer to Figure **??**, then the only configuration for lines $n-2, n-1, n$ is the middle configuration [**?**]. 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 $LC(i = n-1)$ [**?**]. For an example of improvement three with its line encoding for $LC(i = n-1)$ please refer to Figure **??**

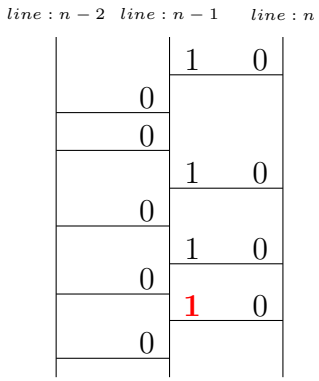


Figure 2.11: The line coding for $LC(i = n-1)$ with imrpovement three is 1011100. The red, bold 1 represents the last left end point in $LC(i = n-1)$, therefore the proceeding 0 must be included in LC_{n-1}. For every other 1 in $LC(i = n-1)$, a 0 is omitted following said 1.

2.5.3.4 Combining All Three

The combination of all three improvements can be done independently. Let $IC(l(4,2,3,1))$ be the *improved line-based encoding* for $l(4,2,3,1)$ by applying improvements 1-3 to $LC(l(4,2,3,1))$. Recall that $LC(l)$ denotes the line-based encoding for some ladder l. The $LC(l(4,2,3,1))$ for the ladder in Figure **??** is 11010101000101010000. By applying improvement one, we get 11010101100010101 0. Notice how the last three $0s$ from $LC(l(4,2,3,1))$ were removed because they represented $LC(i = n)$. By applying improvement two to improvement one we get 11010011001001 0. 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 11010011000101 0. 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 $IC(l(4,2,3,1))$ for the ladder in Figure **??** is $IC(l(4,2,3,1)) = 11010011000101 0$.

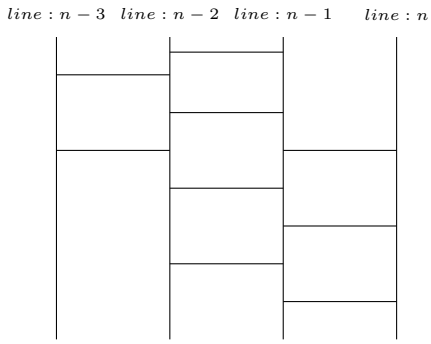


Figure 2.12: A ladder used to illustrate all three improvements $IC(l)$.

$$IC(l) = 11\underline{0}10011\underline{0}001010\underline{}$$

2.6 Enumeration, Counting, and Random Generation of Ladder Lotteries

In the paper, Enumeration, Counting, and Random Generation of Ladder Lotteries, 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.

2.6.1 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 [**?**]. For an example of a forest for $F_{3,2}$ refer to Figure **??**.

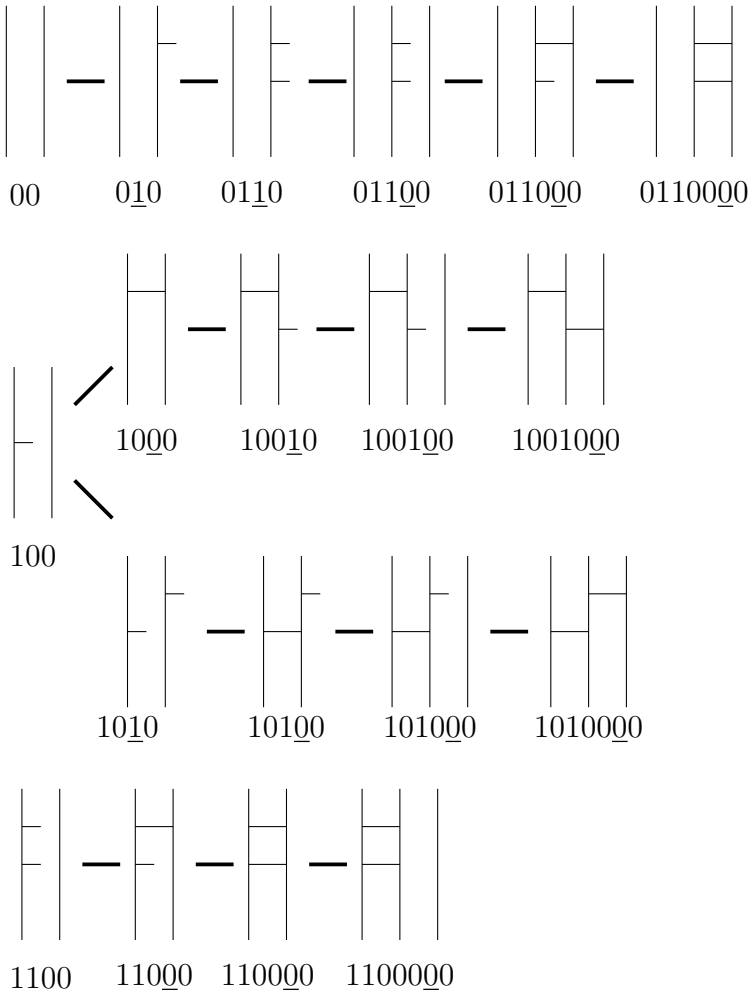


Figure 2.13: The forest, $F_{3,2}$ where 3 is the number of lines and 2 is the number of bars. All ladders with 3 lines and 2 bars are leaf nodes of one of three trees $S_{3,2}$. The underlined bits are the inserted second last bit from the parent's line-encoding resulting in the child's line encoding

The authors create $F_{N,b}$ by defining a removal sequence for each $LC(l)$ [**?**]. Each ladder, l, in $F_{N,b}$ is a leaf node. By removing the second last bit of $LC(L)$ the result is $P(LC(l))$ and the resulting substructure is some *sub-ladder*, $P(l)$, which is an

incomplete ladder containing unmatched endpoints of bars and/or a missing line [**?**]. For example, given $LC(l) = 10100$, $P(LC(l)) = 1010$. Notice how the second last bit was removed. By removing the second last bit from $P(LC(l))$ we get $P(P(LC(l)))$ and $P(P(L))$ respectively. The removal sequence is repeated until the sub-ladder consists of two lines with 0 endpoints attached to line 2 and 0 to r left endpoints are attached to line 1. There are $r + 1$ terminating sub-ladders, i.e., roots of trees in $F_{N,b}$.The removal sequence is unique for each ladder in $F_{N,b}$ is unique.

2.6.2 Counting

The authors provide a method and algorithm to count all ladders with N lines and b bars. According to the authors, the enumeration algorithm is much slower than the counting algorithm [**?**]. 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 there are four sub-divisions of sub-ladders.[**?**]

2.6.2.1 $h < p + q$ **or** $n < 2$

There are zero ladders because it is impossible for the root sub-ladder to have less than two lines. It is also impossible for the number of half bars, h, to be less than the number of detached left end points on line $n-1$ plus the number of detached end points on line n.

2.6.2.2 $n = 2$ **and** $h = p$ **and** $q = 0$

There is only one ladder because the number of half bars on the last line is 0 since $q = 0$. Therefore all half bars are on the $n - 1th$ line of the sub-ladder. This is known because $h = p$ which means the number of half bars is the same as the number of unmatched bars on line $n - 1$. Hence, the unmatched half bars on the $n - 1th$

line must be connected to the n line. Once these are all matched the ladder will be complete. Thus, there is only one ladder for this case.

2.6.2.3 $(n \geq 3$ **or** $h > p)$ **and** $q = 0$

If this is the case, then there are no endpoints attached to line n, but the number of half bars is greater than the number of endpoints attached to line $n-1$, which means there is some line(s) $n-t$, $t > 2$ that have end points attached to them. Let r be a sub-ladder of type $r = t(n, h, p, q)$ with the the above values for n, h, p, q. In order to count the number of ladders of type $t(n \geq 3, h > p, q = 0)$ the authors demonstrate an injection $|t(n \geq 3, h > p, q = 0)| = |t(n-1, h, 0, p)| + |t(n, h-1, p+1, q)|$.[**?**] Let $P(r)$ be r with the removal of $r's$ second last bit in $LC(r)$; i.e. the parent of r. The $LC(R)$ must have a 0 for the second last bit. This 0 designates either the end of line $n-1$ or a right endpoint of a bar attached to line $n-1$. If the second last bit in $LC(r)$ is the right end point of some bar, then $P(r) = t(n, h-1, p+1, q)$. This is because the $n-1th$ bar has a right end point that must be connected to some left endpoint at line $n-2$. Since the removal sequence of the second last bit ensures that there cannot be a right end-point detached from a left end-point. Only left end-points can be detached from right end-points [**?**]. However, if the second last bit of $LC(r)$ designates the end of line $n-1$, then $P(r) = t(n-1, h, 0, p)$. This is because the removal of the second last bit is the removal of the end of line $n-1$ in r. Thus, line n must be empty in r since the last bit in $LC(r)$ designated the end of line n. Thus, if line n is empty and the end point of line $n-1$ has been removed from $LC(r)$, resulting in $P(LC(r))$, the last bit in $P(LC(r))$ must be the end of line $n-1$ in r resulting in a pre-ladder with one less line than r.

2.6.2.4 $h \geq p+q$ **and** $q > 0$

Let r be a pre-ladder of type $t(n, h, p, q)$. The authors demonstrate $|t(n, h \geq p+q, q > 0)| = |t(n, h-1, p+1, q)| + |t(n, h-1, p, q-1)|$.[**?**] The second last bit of $LC(r)$ is

either a 0 or a 1. If it is a 0 then it represents a right end point attached to line n. Thus, removing it to get $P(LC(R))$ is in effect detaching a right end point from some left end point on line $n-1$. Therefore, the parent, $P(R)$ is of type $t(n, h-1, p+1, q)$. Seeing as in the parent, there is now a left end point detached from its right end point in R. However, if the second last bit of $LC(R)$ is a 1, then this indicates the left half of a bar on line n. But since there is no bar $n+1$, this left end point must be detached. Therefore, by removing this 1 in $LC(R)$ results in a parent with one less detached end point on line n. Thus $P(R)$ is of type $t(n, h-1, p, q-1)$.

2.6.3 Random Generation

The random generation of ladder lotteries with N lines and b bars is done by the recurrence relations in the counting and enumerating sections. The goal is to produce some L of type $t(n, 2b, 0, 0)$ where the number of half bars equals the total $2(b)$ and there are no detached end points on lines $n-1$ and n. This implies that there are no detached endpoints on any line $n-t$ where $t \geq 2$ because the removal sequence from the $LC(pre-ladder)$ ensures that any line before $n-1$ has no detached endpoints. Thus, if L is of type $t(n, 2b, 0, 0)$ it is no longer a pre-ladder but a complete ladder with n lines and b bars [**?**].

The authors use an algorithm to generate a random integer, x, in $[1 \ldots |t(n, h, p, q)|]$. where $t(n, h, p, q)$ corresponds to some parent type of ladder. $t(n1, h1, p1, q1)$ corresponds to one child type of $t(n, h, p, q)$ and $t(n2, h2, p2, q2)$ corresponds to the other child type. If $x \leq |t(n1, h1, p1, q1)|$ then generate a pre-ladder of type $t(n1, h1, p1, q1)$ else generate a pre-ladder of type $t(n2, h2, p2, q2)$ [**?**]. Continue until there is type $t(n, 2b, 0, 0)$ which corresponds to a complete ladder lottery with n lines and b bars.

2.7 Permutations

Ladder lotteries and permutations are intricately related to each other. Each π has an $OptL\{\pi\}$ such that each ladder form $OptL\{\pi\}$ sorts π. The so called Listing Problem is one of the problems addressed in this thesis. In brief, this problem is about how to list all $N!$ ladders efficiently. The research for this problem is highly influenced by permutation enumeration research. Knuth describes a number of permutation enumeration algorithms in The Art of Computer Programming [**?**]. Since this book, many algorithms for enumerating $N!$ permutations have been created. During my research for the Listing Problem, I investigated five of these enumeration algorithms. They are the lexicographic algorithm, Heap's algorithm, Zak's algorithm, Steinhaus-Johnson Trotter's algorithm and Effler-Rusky's algorithm. Each of these algorithms will be examined in detail.

2.7.1 Lexicographic Algorithms

2.7.1.1 Basic Lexicographic Algorithm

The lexicographic algorithm enumerates permutations in order from smallest to largest. The lexicographic order for S_3 is $(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)$. This is the same way the words in a dictionary are ordered in the sense that $a < b < c \ldots y < z$. There are many algorithms for enumerating permutations lexiographically [**?**, **?**, **?**, **?**, **?**]. Several will be discussed in this thesis. The first of which is the basic lexicographic ordering algorithm. Please refer to Algorithm **??** to see the basic lexicographic listing algorithm. This algorithm works by beginning with the identity

permutation of order N. Then going right to left, it finds an increasing substring of size two. Once found, the algorithm finds the smallest value greater than the value at index $i-1$ in π to the right of index $i-1$; let this value be known as V. Then the algorithm swaps V p_{i-1} in π. The algorithm sorts π from index i to index N and a recursive call is made. The algorithm terminates when π is in decending order.

Lemma 2.7.1 *The time complexity for the basic lexicographic algorithm is* $O(N!) * N^2 log(N)$.

Proof. The algorithm generates all $N!$ permutations which accounts for the $N!$ factor. The for loop runs between 0 to N times per function call which accounts for the N factors. Lastly, the right portion of the permutation needs to be sorted on every function call. Sorting is done in $Nlog_N$ time. Thus, $N * Nlog_N$ is simplified to $N^2 log_N$. Thus, we get $O(N! * N^2 log_N)$. End of proof. □

2.7.1.2 Alternating Lexicographic Algorithm

In the paper Generating Alternating Permutations Lexicographically, written by Bauslaugh and Ruskey, the authors provide an algorithm for generating permutations in lexicographic ordering such that the permutations form a zig-zag pattern. The zig-zag pattern is formally deifned as follows, given π of order N: If $N = 2K$ then zig-zag $= p_1 < p_2 > p_3 < p_4 \dots p_{N-1} < p_N$. If $N = 2K + 1$ then zig-zag $= p_1 < p_2 > p_3 < p_4 \dots p_{N-1} > p_N$ [**?**]. Please refer to Algorithm **??** to see the Alternating Lexicographic Algorithm.

The initial values for the function are $m = N$, $val = 1$, $level = 0$ and $s = \{1 \dots N\}$ and $\pi = ()$. On each function call val is inserted into π at $level$. Then val is removed from s. If $level$ is odd then the set t gets every value from s less than the max value in s and less than val. If $level$ is even then t gets every value from s greater than the min value in S and greater than Val. Then, for each x in T, the function makes a recursive call with m equal to $m-1$, k equal to x, $level$ equal to

Alternating Permutations of order 5			
13254	14253	14352	15243
15342	23154	24153	24351
25143	25341	34152	34251
35142	35241	45132	45231

$level + 1$ and s equal to s without element x. To see the permutations of order 5 generated by the alternating lexicographic algorithm, please refer to Table **??**.

The authors state that the algorithm is *constant average time* which means the total amount of computation done in generating all the objects, divided by the number of objects, is bounded by a constant. On the average and up to a constant factor no algorithm can run faster [**?**]. The constant refers to the number of function calls before the algorithm terminates. It is defined as follows. Let AL denote the recurrence relation for the Alternating Lexicographic algorithm, let k denote the first element in π and let N denote the number of elements in the set. If $N = 1$ and $K = 1$ then $AL(N, K) = 0$. If $N = N$ and $K = N - 1$ then $AL(N, K) = 1 + AL(N - 1, 1)$ For all other cases, then $AL(N, K) = AL(N, K + 1) + A(N - 1, N - K)$ [**?**].

2.7.2 Heap's Algorithm

Heap's algorithm was developed by B.R. Heap in 1963 [**?**]. The algorithm is based on rotating elements in an array such that the Nth position is occupied by the Nth element for all permutations of the $(N-1)$ objects. Then the Nth element is swapped with one of the elements of in the first $(N - 1)$ positions. The process repeats itself until all of the N elements have occupied the Nth positions. To see Heap's algorithm please refer to Algorithm **??**.

Lemma 2.7.2 *The time complexity of Heap's algorithm is* $O(N!)$

Proof. For each value of $N = [1 \ldots N]$, the algorithm makes N recursive calls, each of which produces all $(N-1)!$ permutations of order $(N-1)$. Thus, you get $N(N-1)! = N!$ which is $O(N!)$. End of proof. □

2.7.3 Zak's Algorithm

Zak's Algorithm was first written about by Shmuel Zak's in the paper A New Algorithm For Generation Of Permutations [**?**].The algorithm is based on reversing suffixes in a permutation of different sizes until all $N!$ permutations have been listed. The algorithm makes use of a *suffix vector* which is a vector for holding all the suffix sizes to be reversed. Since there are $N!$ permutations, there are $N!$ non-unique suffix sizes held in the suffix vector. The recurrence relation for the suffix sizes is as follows. Let $S(N)$ denote the suffix of size N. Then we get the following recurrence relation for $S(N)$. If $N = 2$ then $S(N) = 2$, else $S(N) = (S(N-1)N)^{N-1}S(N-1)$. Let $V(N)$ denote the suffix vector of size N. Then if N equals two, append 2 to $V(N)$, else append $V(N-1)$ followed by N $N-1$ times to $V(N)$ [**?**]. On the Nth time append $V(N-1)$. Once the suffix vector has been crated, then for each suffix size in the suffix vector, reverse the suffix of that size in π. To see Zak's Algorithm for creating the suffix vector please refer to Algorithm **??**

Lemma 2.7.3 *The time complexity of Suffix Vector is* $O(N!)$

Proof. For each value of $N = [1 \ldots N]$, the algorithm makes N recursive calls, each of which produces all $(N-1)!$ permutations of order $(N-1)$. Thus, you get $N(N-1)! = N!$ which is $O(N!)$. End of proof. □

Lemma 2.7.4 *The time complexity to create all* $N!$ *is* $O(N!(N))$

Proof. Reversing a suffix is done in $O(N)$ time. The reversal happens $N!$ times. Therefore $O(N(N!))$. End of proof. □

2.7.4 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 [**?**]. This makes the SJT algorithm a very efficient algorithm for listing S_N. Let an *even permutation* be defined as a permutation with an even number of inversion. Let an *odd permutation* be defined as a permutation with an odd number of inversions. 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 even permutation. The Nth element is inserted into all positions of π of order $N-1$ in ascending order if π of order $N-1$ is an odd permutation [**?**]. 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 $(2, 1, 3)$. To see the Steinhaus-Johnson Trotter Algorithm please refer to Algorithm **??**

Initialize π to the identity permutation, initialize *currentElement* to 2, initialize N to p_{max} and initialize all indices of *direction* to true. If *currentElement* is greater than N, print π and return. Otherwise, begin a for loop that runs $i = [1 \ldots N-1]$ times. In the for loop, first make a recursive call with *currentElement* increasing by one. If *direction*[*currentElement*] is true, then swap *currentElement* in π with its left neighbor. Else *direction*[*currentElement*] is false then swap the *currentElement* in π with its right neighbor. Once the for loop has exited, make one more recursive call with *currentElement* + 1 outside the for loop; this avoids an extra swap operation from occurring while still maintaining the correct number of recursive calls. Lastly, negate *direction*[*currentElement*], which effectively changes the direction of the *currentElement* for the next time it is to be swapped.

Lemma 2.7.5 *The time complexity of SJT is $O(N!)$.*

Proof. The algorithm lists all $N!$ permutations. Given a permutation of order $N-1$, N permutations of order N are derived from this given permutation. Thus, the recurrence relation for $SJT(N) = N(SJT(N-1))$ with $SJT(1) = 1$. If $currentElement = N$ then each recursive call outputs a new permutation. If $currentElement < N$ then the number of recursive calls that need to be made before a permutation is output is $N - currentElement + 1$. □

2.7.5 Effler-Rusky Algorithm

In the paper A CAT Algorithm for Generating Permutations with a Fixed Number of Inversions, written by Effler and Rusky, the authors provide a *constant amortized time* algorithm for generating all permutations of order N with k inversions [**?**]. *Constant amortized time* refers to the average time taken per operation over many operations, given that worst case scenarios are rare. 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 [**?**]. The algorithm moves from right to left and is based off of placing an element $x \in [1 \dots N]$ at the next index in π from right to left. The placement of x in π reduces k. Let $pos(x)$ be the position of element x in an ordered list of remaining elements of $[N]$. The remaining elements are defined as the elements of $[N]$ that have yet to be added to π. For example, let $N = 4$. Let the current state of $\pi = (,,,2)$. Let the remaining elements be ascending in $L = [1 \dots 4] - \{2\}$. Assume element $x = 3$ is to be placed at position $N' = 3$ in π. Assume the current value of $K = 2$. $pos(3)$ in L is two. Thus, k is reduced by $N' - pos(x) = 3 - 2 = 1$; $K = K - 1 = 1$. Meaning that by placing element 3 in position three in π, the remaining number of inversions to be created in π is $K = 1$. To see the algorithm please refer to Algorithm **??**.

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 $[1 \ldots N]$ wires and $[0 \ldots M]$ 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 **??**.

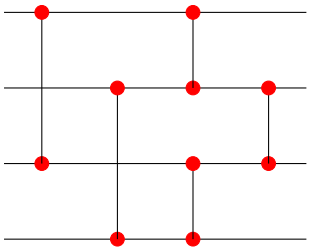


Figure 2.14: Complete Sorting Network for $N = 4$.

Sorting networks were first studied in 1954 by Armstrong, Nelson and O'Connor. Sorting networks can be implemented either in hardware or in software [**?**]. Donald Knuth describes how the comparators for binary integers can be implemented as simple, three-state electronic devices [**?**]. Batcher, in 1968, suggested using them to construct switching networks for computer hardware, replacing both buses and the faster, but more expensive, crossbar switches [**?**]. Since the 2000s, sorting networks are used by the *general purpose graphics processing unit community*, which are a group of people who use the GPU for non-graphical programming, for constructing sorting algorithms [**?**].

Sorting networks are intricately related to ladder-lotteries. Let a *minimum sort-*

ing 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 one to one mapping from the comparators in a minimum sorting network to the bars in an optimal ladder lottery and there is a one to one mapping from the wires in a minimum sorting network and the lines in a ladder lottery [**?**].

2.8.1 The Integer Sequence Relating to the Reverse Permutation

Let $Rev(\pi)$ refer to the reverse permutation of $[1 \dots N]$. There is an integer sequence that counts the number of minimum sorting networks for $Rev(\pi)$. This integer sequence also counts $OptL\{Rev(\pi)\}$. 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 **??** [**?**].

There is currently no known closed form solution for this sequence. Thus, new values of N are counted by a variety of algorithms. In the paper, Efficient Enumeration of all Ladder Lotteries and its Application, 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, 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 [**?**]. The data structure is a digraph that holds a set of elementary permutations along with a number of operations that are applied to the elementary permutations [**?**]. The data structure resembles a digraph with two sink node; one sink node is labelled the zero sink node and the other is labelled the one sink node [**?**]. A πDD is defined as follows [**?**]:

- There are two sink vertices labelled the zero and one sink node.
- Each other vertex has two outgoing edges labelled with a zero or one known as the *zero edge* and *one edge* respectively

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	14163808995580022218786390

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

- Each vertex P is labelled by a tuple $(xP,yP) \in SXS$ where S is an elementary permutation and $xP > yP$.
- (ordered) If the 0-edge of a vertex P points Q, then either $xP = xQ$ and $yP < yQ$ or $xP > xQ$ holds. If the 1-edge of a vertex P points R, then $xP > xR$ holds.
- There is no vertex P whose 1-edge directly points the 0-terminal
- There are no distinct vertices P and Q such that $(xP, yP) = (xQ, yQ)$
  their 0-edges point the same vertex
  their 1-edges point the same vertex

Each vertex P represents a set ΠP of permutations. Let $\tau xP, yP$ be defined as

a transposition of $x, y \in P$. P is recursively defined as follows:

- if P is the zero sink then $P = \emptyset$
- else if P is the one sink then $P = \pi_{ID}$
- else $\Pi P = \Pi Q \cup (\Pi R.\tau xP, yP) = \Pi Q \cup \{\pi\tau xP, yP | \pi \in R\}$ whose 0 edge points to Q and whose 1 edge points to R.

Algorithm 5 Shifts the sub tree of bars up or down the ladder depending on if a right or left swap operation is being performed

```
1: function ShiftSubLadder(ladder, bar, offset, index)
2:     if ladder[row][col] = 0 then
3:         return
4:     end if
5:     row ← bar's row in ladder
6:     col ← bar's column in ladder
7:     if lc(bar) = 0 and rc(bar) = 0 then
8:         Swap(ladder[row + offset][col], ladder[row][col])
9:     else
10:        rightChild ← rc(bar)
11:        leftChild ← lc(bar)
12:        if rightChild ≠ 0 then
13:            ShiftSubLadder(ladder, rightChild, offset, index)
14:        end if
15:        if leftChild ≠ 0 then
16:            ShiftSubLadder(ladder, leftChild, offset, index)
17:        end if
18:        Swap(ladder[row + offset][col], ladder[row][col])
19:    end if
20: end function
```

Algorithm 6 The Basic Lexicographic Algorithm

1: **function** BASIC LEX(π, N)
2: **for** $i \leftarrow N, i >= 1, I \leftarrow i - 1$ **do**
3: **if** $p_i > p_{i-1}$ **then**
4: $v \leftarrow Min(\pi \setminus (p_0, p_1, \ldots p_{i-1}))|v > p_{i-1}$.
5: $Swap(p_{i-1}, v)$ in π.
6: $Sort(p_i, p_{i+1} \ldots p_N)$
7: break
8: **end if**
9: **end for**
10: $BaiscLex(\pi, N)$
11: **end function**

Algorithm 7 Alternating Lexicographic Enumeration Algorithm

```
1: function AlternatingLex(m, val, level : int, s = {1...N}, π)
2:     p_level ← val
3:     s ← s − {val}
4:     if m = 1 then
5:         print(π)
6:         return
7:     end if
8:     t ← {}
9:     if level = 2k + 1 then
10:        t ← {x ∈ s | x < s_max and x < val}
11:    else
12:        t ← {x ∈ s | x > s_min and x > val}
13:    end if
14:    for x ∈ t do
15:        AlternatingLex(m − 1, x, level + 1, s − {x}, π)
16:    end for
17: end function
```

Algorithm 8 Heaps Algorithm for Generating all $N!$ Permutations

```
1: function HEAPS(π, N)
2:     if N = 1 then
3:         Print(π)
4:     else
5:         for i ← 1, i ≤ N, i ← i + 1 do
6:             Heaps(π, N − 1)
7:             if N = 2k + 1 then
8:                 Swap(p_1, p_N)
9:             else
10:                Swap(p_i, p_N)
11:            end if
12:        end for
13:    end if
14: end function
```

Algorithm 9 Creating the suffix vector

```
1: function SuffixVector(vector, N)
2:     if N = 2 then
3:         append 2 to vector
4:     else
5:         for i ← 1, i < N, i ← i + 1 do
6:             append SuffixVector(vector, N − 1) to vector
7:             append N to vector
8:         end for
9:         append SuffixVector(vector, N − 1) to vector
10:    end if
11: end function
```

Algorithm 10 SJT Algorithm for listing S_N

```
1: function SJT(π, currentElement, N, direction = [])
2:     if currentElement > N then
3:         print(π)
4:         return
5:     end if
6:     for i ← 1, i < currentElement, i ← i + 1 do
7:         SJT(π, currentElement + 1, N)
8:         index ← index of currentElement in π
9:         if direction[currentElement] = true then
10:            Swap(p_index, p_index−1)
11:        else
12:            Swap(p_index, p_index+1)
13:        end if
14:    end for
15:    SJT(π, currentElement + 1, N, direction)
16:    direction[currentElement] ←!(direction[currentElement])
17: end function
```

Algorithm 11 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 x ∈ list do
6:             if N − pos(x) ≤ k ≤ (N−1 choose k) + n − pos(x) then p_N ← x
7:                 remove x from list
8:                 KInversions(π, N − 1, k ← k − (N − pos(x)), list)
9:                 insert x in list at correct position.
10:            end if
11:        end for
12:    end if
13: end function
```

Chapter 3

The Listing Problem

3.1 Introduction to the Problem

Listing problems are common problems in combinatorics. In general, listing problems focus on enumerating the objects of a given finite set in some specific order. The listing problem in this thesis will be termed *The Canonical Ladder Listing Problem*. The problem is stated as follows: Let S_N be the set of all $N!$ permutations of order N. To see S_4 please refer to table **??**.

Table 3.1: Table for all $4! = 24$, permutations of order 4

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

Let π be a permutation from S_N. Let *the canonical ladder* be a unique ladder from each permutation's $OptL\{\pi\}$. Let $CanL\{\pi_N\}$ be the set of all canonical ladders for each π in S_N. Let l_i be some arbitrary canonical ladder from $CanL\{\pi_N\}$. A *change* is defined as the insertion or deletion of one or more bar(s) to get from l_i to l_j or the relocation of one or more bars in l_i to get to l_j. The *relocation* of a bar is defined as moving a bar from a given row and column to a new row and/or column in the

ladder under the following conditions. The relocation cannot be a right/left swap operation. If the relocation of the bar moves the bar to a new row, but not a new column, then the endpoint of the bar being moved must cross the endpoint of a bar not being moved. To see examples of the relocation of a bar please refer to Figure **??**.

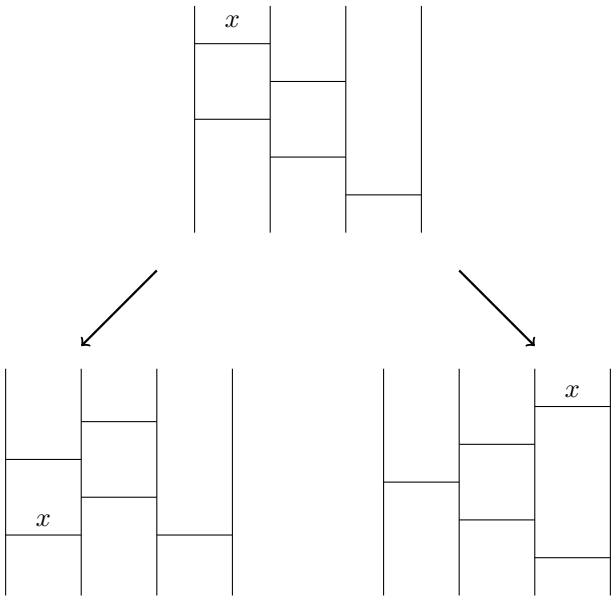


Figure 3.1: Example of relocating bar x

The canonical listing problem asks, given all S_N, is there an efficient way to generate $CanL\{\pi_N\}$? Efficiency is defined as using minimal change to transition from l_i to l_j. For example, let $N = 4$, then $|S_N| = 24$ Since each permutation has at least one ladder in its respective $OptL\{\pi\}$, then $|CanL\{\pi_4\} = 24|$. $CanL\{\pi\}$ consists of a canonical representative from each $OptL\{\pi\}$.

Theorem 3.1.1 *In order to transition from canonical ladder l_i to canonical ladder l_j, at least one bar has to be added or removed from l_i or at least one bar has to be relocated in l_i.*

Proof. We begin this proof by contradiction. Suppose l_i is some canonical ladder in $CanL\{\pi_N\}$. Suppose that l_j is the next canonical ladder in $CanL\{\pi_N\}$. Suppose a

bar does not need to be added or removed from l_i to get to l_j nor does a bar need to be relocated to get to l_i to l_j. We know that each bar in l_i uninverts a single inversion in p_i. We know that each bar in l_j uninverts a single inversion in p_j. We know that $p_i \neq p_j$. Therefore we know that $l_i \neq l_j$. Two ladders corresponding to two different permutations differ from each other in three ways. The first way is by the number of lines, the second way is by the number of bars, and the third way is the location of bars. Note that two ladders can differ in more than one of the three ways. In the case of l_i and l_j, they have the same number of lines seeing as they are ladders of order N. Therefore, they cannot differ in terms of the number of lines. We also assumed that l_i and l_j have the same number of bars and the same location of bars. Which means that the ladders are the same. But we already stated that $l_i \neq L_j$. Therefore we have a contradiction. Which means that either a bar needs to be added/removed from l_i to get to l_j or a bar needs to be relocated in l_i to get to l_j. End of proof. $\square$

In this thesis, two listing algorithms are used to list the canonical ladders for each $CanL\{\pi_N\}$. The first of these listing algorithms is a modification of the Steinhaus-Johnson-Trotter permutation listing algorithm. The second listing algorithm is influenced by Effler and Ruskey's algorithm in their paper A CAT Algorithm for Generating Permutations with a Fixed number of Inversions. This second listing algorithm is termed the cyclic-bar algorithm. Both of these algorithms will be described, explained and analyzed throughout the remainder of the chapter.

Before proceeding, the selection process for choosing the canonical ladder will be explained. In general, the canonical representative from $OptL\pi_N$ is chosen based on a tree structure. The root of the tree is the *identity ladder* which is the only ladder corresponding to $\pi = (1, 2 \ldots, N)$. Proceeding from the root ladder, for every canonical ladder l_i, it is chosen based on the minimal amount of change required to get from l_{i-1} to l_i. Since the minimal amount of change is defined as the insertion or deletion of a single bar or the relocation of a single bar, then the canonical representative, l_i is

derived from l_{i-1} by applying the minimal amount of change. The exception to this rule is in the cyclic-bar algorithm in which transitioning between trees in the forest structure generated by the algorithm.

3.2 Procedure

So far, the problem has been introduced and the required terminology has been defined. Recall that there are two changes; the insertion/deletion of bars or relocation of bars. However, there has yet to be discussion regarding the two listing algorithms. In the procedure section we look at each of the algorithms and explain what each of the algorithms are doing. The goal is to transition from l_i to l_{i+1} in $CanL\pi_N$ with minimal change, which means adding or removing the least number of bars to get from l_i to l_{i+1} or relocating the least number of bars to get from l_i to l_{i+1}.

The reason that the $ModifiedSJT$ and CI algorithms were chosen is because they allow for minimal change from l_i to l_{i+1}. While conducting this research, modifications to the permutation listing algorithms mentioned in chapter one were applied for listing $CanL\{\pi_N\}$. Recall that these listing algorithms were Zaks, Heaps, and Lexicographic. These listing algorithms did not allow for minimal change when transitioning from l_i to l_{i+1}.

3.2.1 The Modified Steinhaus-Johnson-Trotter Algorithm

Algorithm 12 $ModifiedSJT$ algorithm for listing $CanL\{\pi_N\}$

```
1: function ModifiedSJT(N, ladder[2(N − 1) − 1][N − 1], route, direction[N])
2:     if route > N then
3:         print(Ladder)
4:         return
5:     end if
6:     for i ← 0,i < route, i ← i + 1 do
7:         if i := 0 then
8:             ModifiedSJT(N, ladder, route + 1, direction)
9:         else
10:            if direction[N] = false then
11:                row ← (N) + (N − route) − (i);
12:                col ← route − i
13:                ladder[row][col] ← 1
14:            else
15:                col ← i
16:                row ← 2(N − route) + (i)
17:                ladder[row][col] ← 0
18:            end if
19:        end if
20:        ModifiedSJT(N, ladder, route + 1, direction)
21:    end for
22:    direction[route] ←!(direction[route])
23: end function
```

Let the *identity ladder* be the ladder for the sorted permutation from $[1 \dots N]$. Let the initial conditions of the algorithm be the fallowing. Let a 1 at $row = i, col = j$ indicate a bar at $row = i, col = j$. Let a 0 at $row = i, col = j$ indicate the absence of a bar at $row = i, col = j$. Let *ladder* be a 1 indexed two dimensional array initialized with all zeros. Let $N \geq 2$. Let N be the maximum element. Let *direction* be a one indexed array set to false for all indexes $[2 \dots N]$. Let false indicate 'to add a bar'. Let true indicate to 'remove a bar'. Let *route* be the current route to have its bars added or removed; *route* is incremented from $[2 \dots N]$ seeing as 1 has no bars belonging to its route in *ladder*. The principles of the algorithm are the following. If $direction[route]$ is false, then bars will be added to *route*, from right to left, bottom to top, until no more bars to *route* can be added. If $direction[route]$ is true, then bars will be removed from *route*, left to right, top to bottom, until no more bars from *route* can be removed. Once all the bars for *route* have been added or removed, then $direction[route] \leftarrow !direction[route]$ indicating that the opposite operation will be applied to the bars of *route*. The maximum number of bars for a given *route* is $route - 1$ and the minimum number of bars is 1. The initial *route* value is 2. On each recursive call, *route* is incremented by 1. The base case is $route > N$ indicating that there is no such *route*, therefore return. The algorithm is much like the SJT algorithm; given the current number of bars in the ladder for $route = k$, add or remove all bars for $route = k + 1$, then add or remove one bar from $route = k$. Repeat until all $k - 1$ bars have been added or removed from route k.

Lemma 3.2.1 *$ModifiedSJT$ produces $CanL\{\pi_N\}$*

Proof. Since the alorithm 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 $N-1$, move one column to the right in the ladder. For each odd numbered ladder of order $N-1$ add $0 \dots N-1$

bars to the Nth route starting from the rightmost column and ending at the first column. Doing so results in N ladders derived from a given odd numbered ladder of order $N-1$; the Nth ladder has all $N-1$ bars added to the Nth route. For each even numbered ladder of order $N-1$ remove $0 \ldots N-1$ bars from the Nth route beginning at column 1 and ending at the rightmost column. Doing so results in N ladders derived from a given even numbered ladder of order $N-1$; the Nth ladder has 0 bars in the Nth route. Continue this process for all $N-1!$ ladders of order $N-1$. Doing so results in $(N-1)! * N = N!$ ladders of order N. End of proof. To see an example of the proof please refer to Figure **??**. □

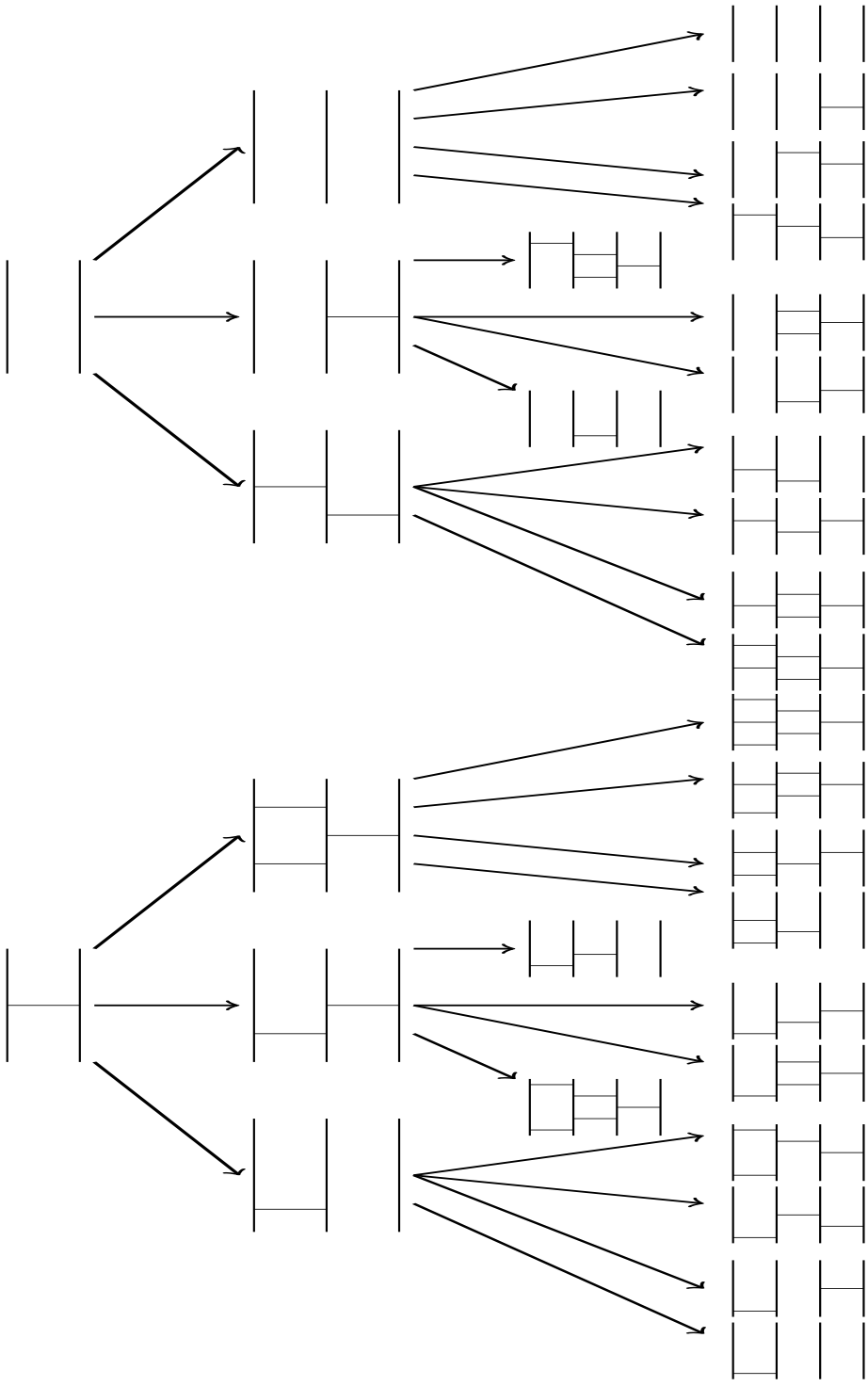


Figure 3.2: $CanL\{\pi_4\}$ generated by the $ModifiedSJT$ algorithm. The algorithm inserts or removes a bar from the previous ladder at the leaf nodes in the tree, the tree is used to prove the veracity of the algorithm

Theorem 3.2.2 *The number of rows required for the ladder data-structure is* $2(N-1)-1$ *and the number of columns required for the ladder is* $N-1$.

Proof. The number of columns is fairly straightforward. Seeing as there are always N elements in π_N, a column represents a gap between lines in the corresponding ladder lottery. Each ladder of order N has N lines, one for each element in π_N. Therefore each ladder of order N has $N-1$ columns.

The number of rows for the ladder data-structure is calculated a follows, given π_N, the minimal number of rows required is when π_N is sorted. In this case there are zero rows because there are zero bars added to the ladder. This ladder is $l(ID)$ and is the first ancestor in $CanL\{\pi_N\}$. When a bar is added to the ladder it can be added to an already existing row or to a new row. If the current state of the ladder is $l(ID)$ then adding the first bar ces the second ladder in $CanL\{\pi_N\}$. Since the bars are being added bottom right to top left, and the first bar to be added belongs to the Nth route, then it must be added to $row = N-1$, $col = N-1$. As bars of the Nth route get continuously added to the ladder, each bar is added a row above the previous bar and to a column to the left of the column of the previous bar. Since no two bars of the Nth route can be on the same row, this will require $N-1$ rows. Note, if they were added to the same row, then the left end point of the right bar would be touching the right end point of the left bar which is disallowed. Once the bars of the Nth element are added, the bars of the $N-1th$ route will be added. The $N-1th's$ first bar will be added to the $N-2$ column, otherwise it would be directly below the first bar of the Nth route, which is a violation. Since the first bar of the $N-1$'s element is added to column $N-2$, then it must be given a new row, otherwise its right end point will be touching the left end point of the first bar of route N. The remaining $N-2$ bars of element $N-1$ will be added bottom right to top left, but none of their end points will touch the end points of element N seeing as they will always be two columns apart from any bar in $N's$ route. The same logic applies to element $N-2$, it will require one extra row for its first bar, in order not to touch the first bar

of element $N-1$, but the remainder of its bars will always be two columns away from the remainder of the bars for $N-1$, etc. Therefore there are $N-1$ rows required for the Nth element and each subsequent element, k requires only one new row. Since $2 \leq k < N$, then there are $(N-2)$ additional rows required for the ladder. Note that element 1 has no bars in its route. Therefore there are $(N-1)$ rows required for element $N's$ bars plus $(N-2)$ rows required for all the remaining $2 \leq k < N$ routes. In conclusion the number of rows required is $(N-1)+(N-2) = 2(N-1)-1$. See figure for the tree of ladders generated by $ModifiedSJT$ for $N = 4$. Note that the maximum number of rows required is $2(N-1)-1 = 2(3)-1 = 5$. □

From Figure **??** it should be clear that the canonical representative from $CanL\pi_N$ when using the $ModifiedSJT$ algorithm is also the root ladder from each $OptL\pi_N$. Recall that the root ladder is the ladder whose bars of a lesser route have not crossed the bars of a ger route. In the case of the $ModifiedSJT$ algorithm, transitioning from l_i to l_{i+1} involves simply inserting a new bar or removing a bar for a given route. Let k be the current route. If a new bar being added belongs to route k, the bar is added below the route of $k+1$ and above the route of $k-1$, therefore adding a bar does not violate the constraint of the root ladder. Let l_i be the current route ladder. Let l_{i+1} be the next ladder in the sequence. Then removing a bar from l_i cannot make l_{i+1} a non-root ladder, because removing a bar from l_i does not allow the bar of a lesser element to cross the bars of a ger element. Thus, the canonical representative for $CanL\pi_N$ is always the root ladder from each $OptL\pi_N$.

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. 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 3.2.3 *Assume a bar is being added. Let i be the current number of bars in the ladder for $route = k[2 \dots N]$. Then the $row = (N-1) + (N - route) - i$.*

Proof. It must be noted that we are listing only root ladders. So when transitioning from l_j to l_{j+1} in $CanL\pi_N$ both are root ladders. With this in mind, one can say that the number of rows required for $route = N$ is $N-1$ seeing as the Nth value can have at most $N-1$ bars in its route, each requiring their own row. Thus, the $(N-1)$ term in the equation in included for the $N-1$ bars of $route = N$. The $(N - route)$ term is added to $(N-1)$ indicating an offset value for the first bar of the $route$; this term calculates the difference in rows between the first bar of $route = N$ and the first bar of $route = k$. If $route = N$ then $offset = 0$, if $route = N-1$ then $offset = 1$, if $route = N-2$ then $offset = 2$, etc. Since bars are added right to left, bottom, up, then the first bar of $route = k$ will be added at $row = (N-1) + (N - route)$. When a bar is added to $route = k$ route, the i value is incremented by one. This value is subtracted in order to effectively move up the ladder as bars are added to $route = k$. Since bars are added bottom to top, row needs to be decremented for each new bar to be added. Refer to Figure **??** for an example of row calculation when adding a bar. □

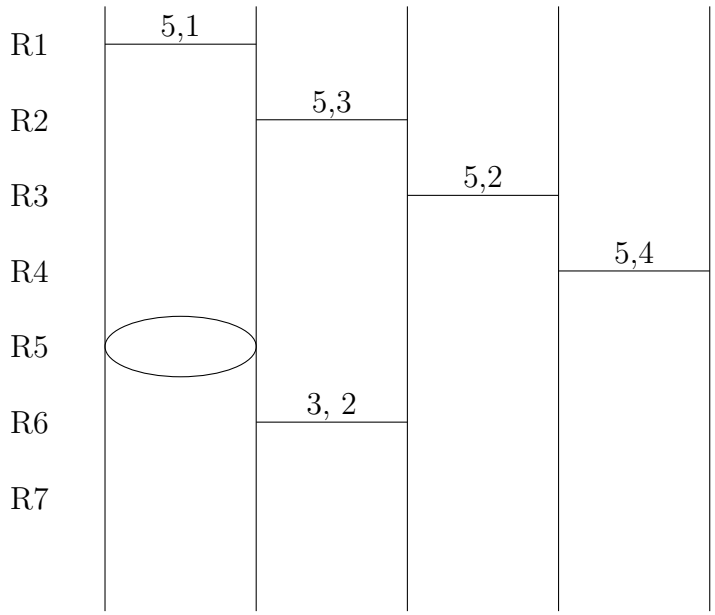


Figure 3.3: The second bar of route 3 goes will go in row 5, column 1. $5 = (5-1) + (5-3) - 1 = (N-1) + (N - route) - i$.

Lemma 3.2.4 *Assume a bar is being added. Let i be the current number of bars in the ladder for $route = k[2 \ldots N]$. Then the $col = route - 1 - i$.*

Proof. The total number of bars required for $route = k$ is $k-1$, each requiring their own column. The columns span from $1 \ldots k-1$. The bars are added right to left and when a bar is added i is incremented by one. Since bars are added right to left, the first bar of $route = k$ is inserted at column $k-1$. This is because the first bar of the kth route is the left child bar of the lowest bar of the $k+1th$ route. Let y be the first bar to be added for $route = k$ and let x be the lowest bar of the $route = k+1$. x is the parent bar of y and y is the left child of x for the following reasons. If y was directly below x, then the ladder would have redundant bars, thus making it non-optimal. If y was to the right of x, then y would either be above x, thus violating the property of the root ladder, or if y were below x and to the right of x then y would be part of the route for $k+1$, yet this is a contradiction seeing as we said y belongs to $k's$ route. Therefore, y must be in a column to the left of x. As bars are added to $k's$ route, i is

incremented for each bar. i is subtracted from the original column, $k-1$, effectively moving to the next column to the left in *ladder*. See figure **??** for an example of column calculation when adding a bar for $k < N$. □

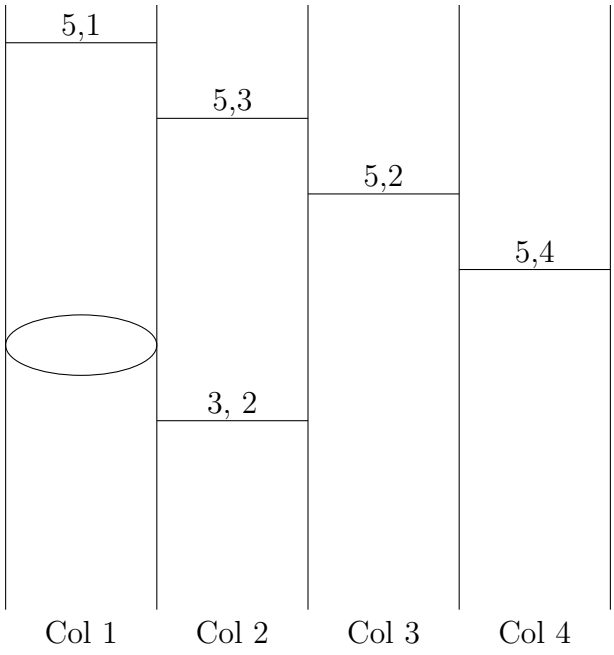


Figure 3.4: The second bar of route $k = 3$ goes will go in column 1. Since one bar has been added, $i = 1$. $col = 1 = 3 - 1 - 1 = route - 1 - i$.

Lemma 3.2.5 *Assume a bar is being removed from* $route = k[2 \dots N]$. *Let* i *represent the number of bars that have currently been removed from* $route = k$. *Then* $row = 2 * (N - route) + (i) + 1$.

Proof. When removing a bar the row is calculated as follows. Keeping in mind bars are removed from top to bottom, left to right. The leftmost bar of the $route = kth$ element is in the same column as the leftmost bar of the $route = k + 1th$ element. Thus, the first bar of the $route = kth$ element must be two rows below the leftmost bar of the $route = k+1th$ element. Therefore, the following recurrence relation holds for calculating the row of the leftmost bar of $route = k$:

$$row(route = k) = \begin{cases} row(route = k + 1) + 2 & \text{if } k < N. \\ 0 & \text{if } k = N. \end{cases} \tag{3.1}$$

The recurrence relation simplifies to $2(N - route)$ which means that the difference between N and $route = k$ multiplied by 2 is the same as the recurrence relation.

Every time a bar is removed from $route = k$, i is incremented indicating a bar has been removed. Since bars are removed from top to bottom, i is added to $2(N - route)$ in order to effectively move down the ladder by i rows starting from $row = 2(N - route)$. The $+1$ is added due to *ladder* being a 1 indexed array; if *ladder* were a 0 indexed array as is true for most languages, the $+1$ would be removed from the equation. See figure **??** for an example of removing a bar. □

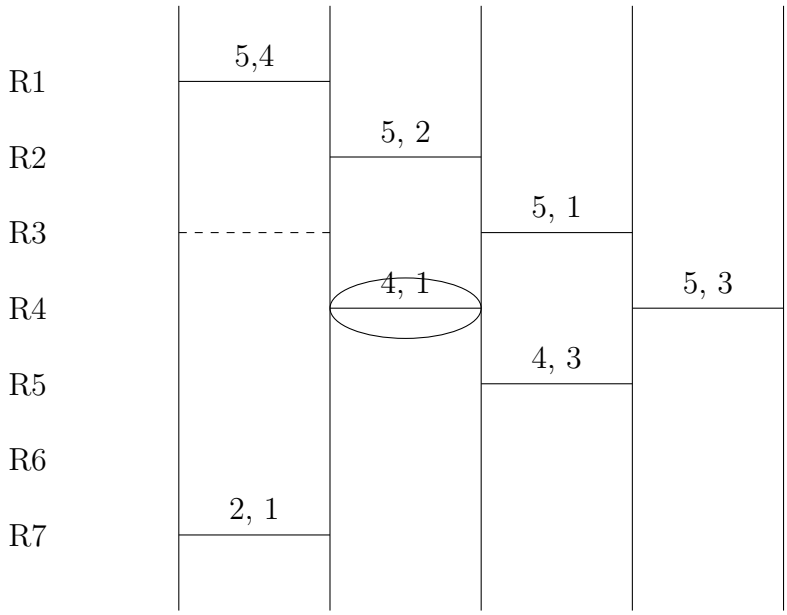


Figure 3.5: The bar to be removed for route $k = 4$ is (4, 1) which is at row 4. The dashed line indicates a bar from route 4 has already been removed. $row = 4 = 2(5 - 4) + 1 + 1 = 2(N - route) + i + 1$.

Lemma 3.2.6 *Assume a bar is being removed from route* $= k[2 \ldots N]$. *Let* i *represent the number of bars that have currently been removed from route* $= k$. *Then* $col(i) + 1$.

Proof. The bars are removed left to right. The first bar to be removed is the leftmost bar belonging to $route = k$ which is always at column 1. i is incremented for each bar removed from the route of k. The +1 is added due to *ladder* being a 1 indexed array; if *ladder* were a 0 indexed array as is true for most languages, the +1 would be removed from the equation. See Figure **??** for an example of column calculation when removing a bar. □

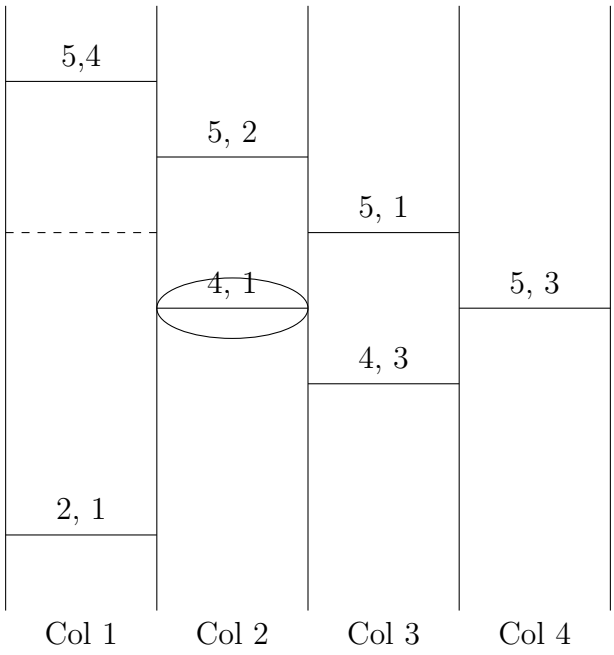


Figure 3.6: The bar to be removed for route $k = 4$ is (4, 1) which is at column 2. The dashed line indicates a bar from route 4 has already been removed. Since one bar from route 4 has been removed, $i = 1$. $column = 2 = 1 + 1 = i + 1$.

3.2.2 The Cyclic Bar Algorithm

Algorithm 13 First part of the algorithm Cyclic Bar

```
1: function CyclicBar(ladder[2(N − 1) − 1][N − 1], currentLimit, maxLimit,
   N, k)
2:     if the number of bars in ladder = currentLimit then
3:         print(Ladder)
4:         return
5:     end if
6:     if currentLimit > maxLimit then
7:         return
8:     end if
9:     if k = N then
10:        m ← 0
11:        row ← k − 1
12:        col ← k − 1
13:        numBars ← current number of bars in ladder
14:        while mumBars < currentLimit AND m < k − 1 do
15:            ladder[row][col] ← 1
16:            row ← row − 1
17:            col ← col − 1
18:            m ← m + 1
19:            numBars ← numBars + 1
20:        end while
21:        if numBars = currentLimit then
22:            Print(Ladder)
23:        end if
24:        remove all bars belonging to k's route.
25:        return
26:    end if
```

Algorithm 14 Cyclic Bar Continued

```
    if k < N then
        count ← 0
        for i ← 0, i < k, i ← i + 1 do
            if the number of bars in ladder = currentLimit then
                break
            end if
            if i = 0 then
                CyclicBar(ladder, currentLimit, maxLimit, N, k + 1)
            else
                row ← (N − 1) + (N − k) − count
                column ← (k − 1) − arr[k]
                ladder[row][column] ← 1
                count ← count + 1
                CyclicBar(Ladder, currentLimit, maxLimit, N, k + 1)
            end if
        end for
        remove all bars from k's route.
    end if
end function
```

Algorithm 15 Driver for the Cyclic Bar Algorithm

```
function CylclicBar Driver(ladder[2(N − 1) − 1][N − 1], N)
    maxLimit ← (N(N − 1))/2
    k ← 2
    for i ← 0, i <= maxLimit, i ← i + 1 do
        CyclicBar(ladder, currentLimit ← i, maxLimit, N, k)
    end for
end function
```

The initial conditions for the algorithm are the following. Let *ladder* be initialized as a two dimensional array with $2(N-1)-1$ rows and $(N-1)$ columns. Let N be initialized to the maximal element in π_N. Let k be initialized to 2. Let the $maxLimit$ be initialized to $(N(N-1))/2$.Let the $currentLimit$ be initialized to zero.

The tree structure is ced as follows. The $currentLimit$ represents the number of bars to be inserted into *ladder*. Once all ladders with $currentLimit$ bars have been ced, the $currentLimit$ is increased by one and the algorithm repeats until $currenLimit > maxLimit$. This ces all ladders in $CanL\pi_N$. The ladders are generated as a forest structure, with each value of $currentLimit$ cing its own tree of ladders. See figure –fig for the forest of ladders for $N = 4$. The forest of ladders is all the ladders in $CanL\pi_N$.

On each recursive call to the function, k is increased by one until $k = N$. When $k = N$ all the remaining bars that need to be added to the ladder are added to $k = N's$ route. The remaining bars equals the $currentLimit$ minus the number of bars in the ladder. Once all of the remaining $k = N's$ bars are added, the algorithm checks if the number of bars in the ladder equals $currentLimit$. If it does, then the ladder is printed, but if it does not then a dead-end is reached seeeing there are not enough bars in the ladder.

When $k < N$ a for loop is implemented for $0 \ldots k-1$ indicating the range for the number of bars to be added for $k's$ route. On each iteration of the for loop a bar is added to $k's$ route followed by a recursive call with k incrementing by one. Once all of the bars for $k's$ route have been added, all the bars from $k's$ route are removed. This process repeats itself until all ladders of order N with $currentLimit$ bars have been added. It should be noted that the row and column calculations are the same as with the $ModifiedSJT$ algorithm.

The forest structure is ced as follows. Simply call the algorithm for the tree structure in a for loop ranging from $0 \ldots N(N-1)/2$. This will increment the current limit for each call to the tree structure resulting in the forest structure. Each combination

of bars into the *ladder* data structure ces the root ladder from each $OptL\pi_N$, thus adding one more ladder to $CanL\pi_N$. Once complete, the tree of ladders terminates, and the *currentLimit* increases, thus cing a new tree in the forest for $CanL\pi_N$. To see the forest ced by the Cyclic Bar Algorithm for $N = 4$ please refer to figure **??**

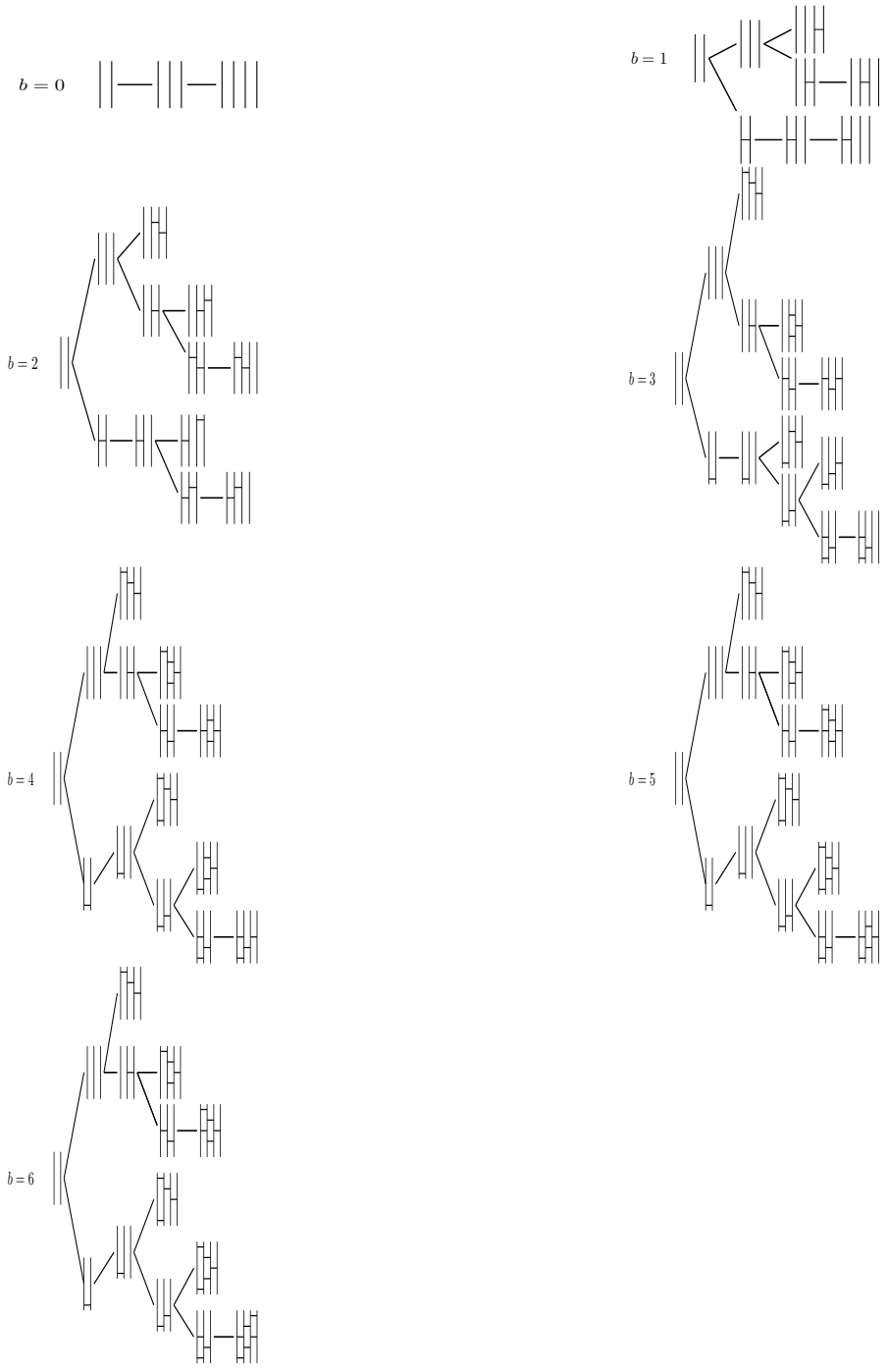


Figure 3.7: The forest for all ladders in $CanL\{\pi_4\}$ generated by the Cyclic Bar Algorithm. The leaf nodes present a possibly correct candiate ladder. If the ladder in the leaf has enough bars, then it is a legitimate ladder in the tree.

It has been stated that the forest ced by the Cyclic Bar algorithm generates $CanL\{\pi_N\}$. This claim has yet to have been proven, so the following theorem will prove this claim.

Theorem 3.2.7 *The forest ced by the Cyclic Bar algorithm generates* $CanL\{\pi_N\}$

Proof. The proof is done by way of a combinatorial proof and induction. Rather than list ladders, we shall list permutations using the same method. Let $List(N, k)$ be the listing of permutations of order N with k inversions. The hypothesis is that $List(N, k) = \sum_{m=0}^{k} List(N-1, m)$ given $N > 1$ and $k \geq 0$. The base case is $N = 2$ and $k = 0$. We know that the identity permutation of order 1 has no inversions. We know that the list containing the identity permutation of order 1 is of length one. We know that the list containing the identity permutation of order 2 is of length one. By appending the value 2 to the only permutation in the list $List(1, 0)$ we get the only permutation in the list $List(2, 0)$. Therefore the base case checks out.

Suppose we have the list of permutations for $List(N-1, 0) \ldots List(N-1, k)$. We want to show that we can insert the Nth element into each of the permutations in each of these lists such that the resulting permutations have N elements with k inversions. Partition k into k' and k''. Note that $k' + k'' = K$. Let k' equal the number of inversions formed by the Nth element. Let k'' equal the number of elements not formed by the Nth element. We can look at the $List(N-1, 0) \ldots List(N-1, k)$ as lists of permutations with k'' inversions. So we write $List(N-1, 0) \ldots List(N-1, k)$ as $List(N-1, k''=0) \ldots List(N-1, k''=k)$. When $k''=0$ we know $List(N-1, 0)$ has one permutation, thus $k' = K$. The Nth element must be positioned k' positions to the left of the Nth position in this one permutation from $List(N-1, 0)$ to form a permutation of order N with k inversions. In general, we can say that for each of the permutations from each of the $List(N-1, k'')$, we know that the Nth element must be positioned $k - k''$ to the left of the Nth position in order to ce a permutation of length N with k inversions. Thus, by exhaustively inserting

N value	k value	k'' value	k' value	L(N-1, k'')	$L(N,k)$
4	2	0	2	$(1,2,3)$	$(1,4,2,3)$
4	2	1	1	$(1,3,2)$	$(1,3,4,2)$
4	2	1	1	$(2,1,3)$	$(2,1,4,3)$
4	2	2	0	$(3,1,2)$	$(3,1,2,4)$
4	2	2	0	$(2,3,1)$	$(2,3,1,4)$

Table 3.2: The table showing all $List(4,2)$ derived from $List(3,0)\ldots List(3,2) + List(4,k')$

the Nth element in all $k - k'' = k'$ positions to the left of the Nth position in all permutations from all $List(N-1,k'')$, we get all permutations of order N with k inversions. Therefore $List(N,k) = List(N,k') + \sum_{k''=0}^{k} List(N-1,k'')$. Seeing as an inversion in a permutation corresponds to a bar in a ladder, then by using this same proof on ladders, we can generate all ladders with k bars. Which is to say that $ladders(N,k) = Ladders(N,k') + \sum_{k''=0}^{k} Ladders(N-1,k'')$. In order to list $CanL\{\pi_N\}$ simply apply this same logic for all k bars for $0 \leq K \leq N(N-1)/2$. To see an example of the above proof for $List(4,2)$ refer to Table **??**.

□

3.3 Results

In the results section, the runtimes of the two algorithms will be provided. The run times are done without printing the ladders. When the ladders are printed, the runtime increases by a substantial amount. The runtime for each algorithm for $N = 10$ will be provided in table **??**. In the analysis section, the table will be further analyzed along with the time and space complexity for each algorithm.

Runtimes for generating $CanL\{\pi_N\}$ in seconds		
N value	Cyclic Bar	Modified SJT
1	0.000000	0.000000
2	0.000000	0.000000
3	0.000000	0.000000
4	0.000000	0.000000
5	0.000000	0.000000
6	0.000000	0.000000
7	0.000000	0.000000
8	0.000000	0.000000
9	0.093750	0.000000
10	0.968750	0.031250
11	12.718750	0.250000
12	174.312500	2.781250

Table 3.3: The table with the runtimes for listing $CanL\{\pi_N\}$ using the Cyclic Bar Algorithm and Modified SJT Algorithm.

3.4 Analysis

From looking at the table in the results section, it is clear that the modified SJT algorithm performs better than the Cyclic Bar algorithm. The reason(s) for this disparity in performance will be analyzed. Following this analysis, areas of application and practical relevance for the Listing Problem will be discussed along with concluding remarks.

3.4.1 Performance Analysis

As $N \geq 9$ there is a noticeable difference between the runtimes of the two algorithms by a sizable order of magnitude. Clearly the modified SJT algorithm performs better than the Cyclic Bar algorithm. The reason(s) for this improved performance are the difference in time complexities between the two algorithms. The time complexity for the modified SJT algorithm is $(N!)N$. The time will be proven in the following

lemma.

Lemma 3.4.1 *The time complexity for the modified SJT algorithm is* $O((N!)N)$

Proof. The $N!$ term is fairly straightforward, the algorithm creates all $N!$ ladders in $CanL\{\pi_N\}$ which accounts for the $N!$ factor. The N term is a result of the second for loop found in the algorithm. The first for loop found in the modified sjt function runs $(N-1)$ times each time the modified SJT function is called. However, on each iteration of this for loop a ladder is listed, therefore the runtime of this for loop is accounted for by the $N!$ factor. However, the second for loop in the helper SJT function runs at worst, $N-1$ times before listing a ladder. This worst case is when the $K=2$ route needs to have a bar inserted or removed. Therefore, this second for-loop accounts for the N factor in the time complexity. Thus, the time complexity of the modified SJT algorithm is $O((N!)N)$. □

Lemma 3.4.2 *The time complexity for the Cyclic Bar algorithm is* $O((N!)N^2)+N^2$.

Proof. The $N!$ term is fairly straightforward, the algorithm creates all $N!$ ladders in $CanL\{\pi_N\}$ which accounts for the $N!$ factor. The N^2 multiple is a result of the for loop that is executed when $2 \leq k < N$. This for loop runs from 1 to k for each value of k. Thus, the for loop is executed $1+2+3+4,...+N-1$ times. This summation is equal to $((N-1)N-2)/2$ which is reduced to N^2. There is also the $+N^2$ term pertaining to backtracking. Once $currenLimit \geq N$, then the algorithm begins to back-track. Each time $currentLimit$ increases from N to $N(N-1)/2$ the number of back-tracks increases by one per $currentLimit$ level. Thus, there are $1+2+3+4,...+((N(N-1))/2-N))+1$ back-tracks required, which is reduced to N^2. Thus, the time complexity is $O((N!)N^2)+N^2$. □

The space complexity is the same for the two algorithms. Both require a two dimensional ladder data structure whose dimensions are $(2(N-1)-1)(N-1)$. Therefore the space complexity for the algorithms is $O(N^2)$.

3.4.2 Application(s)

The applications for generating $CanL\{\pi_N\}$ are currently unknown to me insofar as this problem has yet to be solved to my knowledge. However, if I am to be granted some speculation, I could provide some hypothetical scenarios in which listing $CanL\{\pi_N\}$ could be of interest. The first hypothetical application would be to model an *oblivious sorting system* for $N!$ permutations. An oblivious sorting system is a system such that the sorting operations are done irrespective of the data being passed to the system [**?**]. Recall that a bar in a ladder simply swaps two adjacent elements in a permutation. Due to the static nature of each ladder, the swap operation resulting from two elements in a permutation crossing a bar is unchanging. Seeing as each ladder in $CanL\{\pi_N\}$ sorts the corresponding permutation of order N, one can implement all of $CanL\{\pi_N\}$ for some arbitrary N value and then pass each permutation of order N through its respective ladder from $CanL\{pi_N\}$ resulting in each permutation being ordered. The ladders from $CanL\{\pi_N\}$ only need to be generated once and saved. Once this is done a permutation can be passed to the correct ladder and it can be sorted by having each of its elements pass through the ladder.

Chapter 4

The Minimum Height Problem

4.1 Introduction to the Problem

Let the *height* of a ladder be the number of rows that a ladder has with at least one bar. Let $MinL\{\pi\} \subseteq OptL\{\pi\}$ such that the ladders in $MinL\{\pi\}$ are the shortest ladders from $OptL\{\pi\}$. Therefore, $MinL\{\pi\} \subset OptL\{\pi\}$. Let *a minimal ladder* be a ladder from $MinL\{\pi\}$. The *Minimum Height Problem* asks, given a permutation π, is there an algorithm for generating a minimal ladder from $MinL\{\pi\}$?

Two tangential questions that result from this problem are the following. Let $MinL\{\pi_N\}$ be the set of all $MinL\{\pi\}$ for each permutation of order N. Recall that $OptL\{\pi_N\}$ is the set of all $OptL\{\pi\}$ of order N. Thus, $MinL\{\pi_N\} \subseteq OptL\{\pi_N\}$. The first tangential question is, what are the upper and lower bounds for the heights of ladders in $MinL\{\pi_N\}$? Let *ladders of order N* pertain to ladders derived from some π with N elements. The second tangential question is what ladders of order N have a height of zero or one?

I will address the tangential questions in the introduction. Following the tangential questions, I will provide a number of algorithms for generating one ladder from $MinL\{\pi\}$ in the procedures section; one of which is a heuristic algorithm. In the results section I will provide a table with the heights of the ladder from the heuristic algorithm in comparison to the heights of the ladders in $MinL\{\pi\}$. Finally, in the analysis section there will be a discussion about the efficacy of the heuristic algorithm along with some applications of the algorithm.

4.1.1 Upper and Lower Bounds of the heights of the Ladders in each $MinL\{\pi_N\}$

In this section the upper and lower bounds for the heights of the ladders in $MinL\{\pi_N\}$ will be determined; not the upper and lower bounds for the heights of the ladders in $OptL\{\pi_N\}$. Seeing as $MinL\{\pi_N\} \subseteq OptL\{\pi_N\}$, by determining the lower bound for the height of $MinL\{\pi_N\}$, the lower bound for the height of $OptL\{\pi_N\}$ will also be determined.

Lemma 4.1.1 *The lower bound for the height of a ladder $MinL\{\pi_N\}$ is zero*

Proof. If π_N is the sorted permutation of order N then there are no bars in its ladder. Recall that a bar swaps an adjacent inversion in π. Seeing as there are no adjacent inversions in the sorted permutation of order N, then there are no bars that need to be added to its corresponding ladder. Since a ladder with no bars requires no rows, then the lower bound for the height of a ladder from $MinL\pi_N$ is zero. End of Proof. □

The upper bound for the heights of the ladders in $MinL\{\pi_N\}$ is more difficult to prove than the lower bound. The lower bound is unique seeing as there is only one ladder of order N with zero bars. With the upper bound however, it has yet to be shown if there is an upper bound for $MinL\{\pi_N\}$. Before proving the upper bound for $MinL\{\pi_N\}$ it must be shown how to derive the ladder with minimal height from the root ladder of the reverse permutation of order N. Refer to this ladder as $MinL(Rev(\pi_N))$. Once we have established how to derive $MinL(Rev(\pi_N))$ from the root ladder of the reverse permutation of order N, it will be relatively easy to prove the upper bound for $MinL\{\pi_N\}$.

Let $Rev(\pi_N)$ be the reverse permutation of order N. Let $RootL(Rev(\pi_N))$ be the root ladder for $Rev(\pi_N)$. Recall that the root ladder is the ladder such that no bar of a lesser element has crossed the route of a greater element. $RootL(Rev(\pi_N))$ requires $2(N-1)-1$ rows. See theorem **??**.

In order to create $MinL(Rev(\pi_N))$, one simply needs to take $RootL(Rev(\pi_N))$ and modify it. In order to modify $RootL(Rev(\pi_N))$ correctly, consider what happens when the bars of lesser elements are right swapped above the routes of greater elements. Of course, if this is done to $RootL(Rev(\pi_N))$ then the ladder is no longer $RootL(Rev(\pi_N))$. Nonetheless, when the $N-1th$ route is swapped above the Nth route, this frees up an extra row in the ladder for the $N-2th$ route. This is the row where the last bar of the $N-1th$ element resided before it was swapped above the Nth route. Now, the first bar of the $N-1th$ route will begin in column 2 and end at column $N-1$. Furthermore, a new row will need to be added to the top of the ladder in order to accommodate the first bar of the $N-1th$ route. Now the route of the $N-2th$ element can be raised up a row seeing as its last bar will still be in column $N-3$ and the row/column that was previously occupied by the last bar of the $N-1th$ element will be free. Then the $N-3$ route can be swapped above the route of elements $N-2 \ldots N$. The route of $N-3$ will begin at column 4 and span to column $N-1$. Since a new row was already added above route N for element $N-1$, the first bar of element $N-3$ begins at the same row as the first bar for element $N-1$. By swapping all the $N-Jth$, $1 \leq J < (N-1)$ and $J = 2K+1$, routes above the routes of elements $(N-J)+1 \ldots N$ in $RootL(Rev(\pi_N))$, the ladder is reconfigured to $MinL(Rev(\pi_N))$. The height of $MinL(Rev(\pi_N))$ is N because the Nth element still requires $N-1$ rows, and the $N-1th$ element requires one additional row to be added above the row for the first bar of the Nth element. Please refer to Fig.**??** for an example of modifying $RootL(5,4,3,2,1)$ to $MinL(,4,3,2,1)$. Please refer to Alg.**??** to see the algorithm for creating $MinL(Rev(\pi_N))$ irrespective of $RootL(Rev(\pi_N))$.

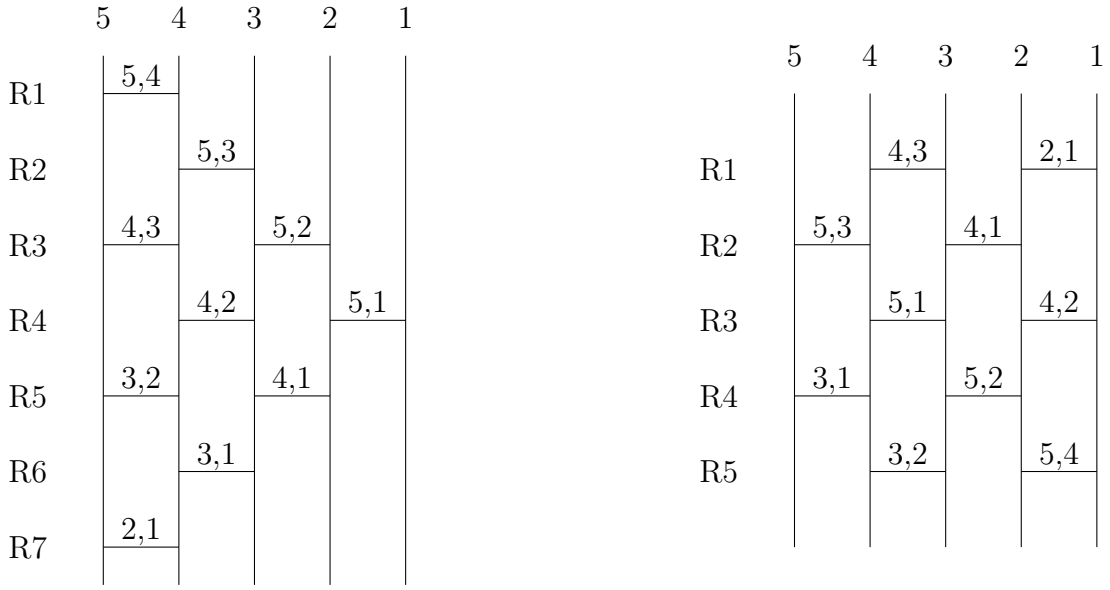


Figure 4.1: The ladder to the left is $Root(5,4,3,2,1)$. The ladder to the left is $MinL(5,4,3,2,1)$. Note that $N = 5 = 2K + 1$, thus by swapping routes 2 and 4 above route 5 whilst leaving route 3 below route 5 in $Root(5,4,3,2,1)$, we get $MinL(5,4,3,2,1)$. There is no way to reduce the height further seeing as route 5 still needs 4 rows and route 4 needs one extra row for its first bar.

Now that $MinL(Rev(\pi))$ has been established, we can prove the upper bound for $MinL\{\pi_N\}$.

Lemma 4.1.2 *The upper bound for $MinL\{\pi_N\}$ is N.*

Proof. We shall use a proof by contradiction. Suppose that the upper bound for the height of $MinL\{\pi_N\}$ was greater than N. (It cannot be less than N because we have already demonstrated that the minimal height of the ladder for the reverse permutation is N). Let $MinL(Rev(N))$ be the minimal ladder for the reverse permutation of order N. Refer to Fig.**??** for an example of $MinL(5,4,3,2,1)$. It will be shown that one $MinL\{\pi\}$ of order N can be derived from $MinL(Rev(N))$. Recall that a bar univerts an inversion in a permutation. By removing bars from $MinL(Rev(N))$, that is effectively removing inversions from $Rev(\pi(N))$. Of course, when a bar is removed

from $MinL(Rev(N))$, the ladder ceases to be $MinL(Rev(N))$. Let k be the number of bars in the current state of the ladder, with $MinL(Rev(N))$, $k = (N(N-1))/2$. For each subsequent ladder, $0 \leq k < (N(N-1))/2$. Thus, to create the minimal ladders with $k = ((N(N-1))/2) - 1$ bars, simply remove one of the correct bars from $MinL(Rev(N))$. Once all the minimal ladders with $k = ((N(N-1))/2) - 1$ bars have been created, simply remove the correct bar from each of these ladders with $k = (N(N-1))/2-1$ bars to get all minimal ladders with $k = ((N(N-1))/2)-2$ bars. This process continues until each minimal ladder of order N has been created. Since bars are only being removed from ladders, no more rows will be added to the ladder. Removing a bar does not necessarily remove a row, but removing a bar definitely does not add a row to the ladder. Earlier we stated that the height of $MinL(Rev(N))$ is N, and at the same time we stated that we could create one minimal ladder order N from each $MinL\{\pi\}$ of order N by deriving it from $MinL(Rev(N))$ through removing bars. Yet at the beginning of the proof, we supposed the upper bound was greater than N which contradicts the claim that by removing bars from $MinL(Rev(N))$ the height of $MinL(Rev(N))$ will not increase. Thus, the upper bound for $MinL_{\{}\pi_N\}$ is N. Please refer to Fig.**??** for the removal sequence which lists $MinL\{\pi_4\}$.

□

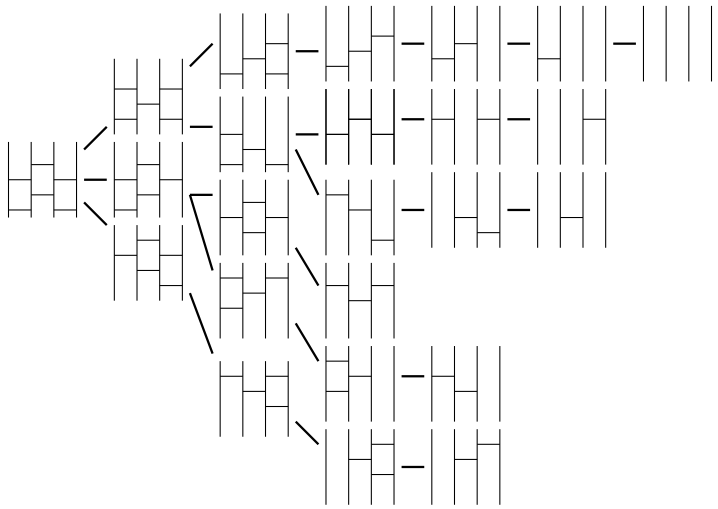


Figure 4.2: Removal sequence of bars from the minimal ladder for $(4, 3, 2, 1)$ resulting in $MinL\{\pi_N\}$

4.1.2 Minimal Ladders of Order N with Heights of Zero or One

There are some ladders of order N which have a height of zero or one. There is only one permutation of order N which results in a minimal ladder with a height of zero, namely the identity permutation. This point has already been proven in the lemma for the lower bound of the minimal height. What is more interesting is ladders of order N with a height of one. One may be tempted to assume that if the identity permutation results in a minimal ladder with a height of zero, then all permutations of order N with exactly one inversion result in minimal ladders with a height of one. Although this is true, it is only partially true. There are permutations of order N with more than one inversion which result in minimal ladders with a height of one. Below will be presented one algorithm, one recurrence relation and one formula pertaining to ladders of order N with a height of one. The algorithm lists all ladders of order N with a height of one. The recurrence relation counts all ladders of order N with a height. The formula is the closed form solution to the recurrence relation. The similarities between ladders of order N with a height of one and other mathematical objects will also be analyzed.

4.1.2.1 Listing Algorithm for all Ladders of Order N with a Height of One

Algorithm 16 Listing Algorithm For All Ladders of Order N with a height of 1

```
1: function GenHeightOne(ladder[1][k = N − 1], col = N − 1)
2:     if col < 1 then
3:         return
4:     end if
5:     ladder[1][Col] ← 1
6:     GenHeightOne(ladder, col − 2)
7:     ladder[1][Col] ← 0
8:     GenHeightOne(ladder, col − 1)
9: end function
```

Let *ladder* be a two dimensional array initialized as the identity ladder of order N. Let *col* be initialized to $N-1$ indicating the current column. When a 1 is inserted at $ladder[1][Col]$ that indicates a bar has been added to row 1, Col. When a 0 is inserted at $ladder[1][Col]$ that indicates a bar has been removed from row 1, Col. Since no two endpoints of two bars can be touching, the function moves two columns to the left on the first recursive call. This ensures that the next bar added will be two columns away from the current bar that was just added. Once the *col* is less than 1 the function returns to the previous value of *col* and removes the bar that was at $ladder[1][col]$. This now frees the column that is one away from the value of *col*. Thus, the function makes a second recursive call, this time reducing *col* by one. Each call to the function produces a unique ladder. To see the tree of all ladders with a height of one for $N = 5$ please refer to Fig.**??**.

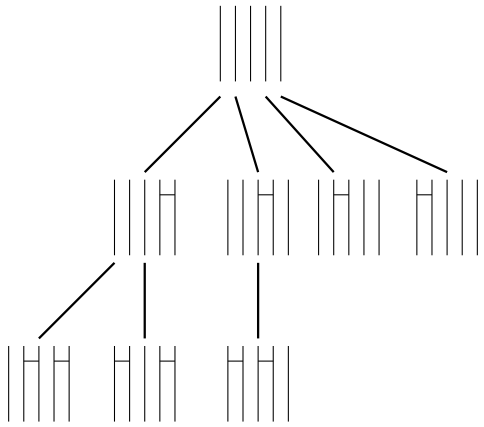


Figure 4.3: All 7 ladders of order 5 with a height of one listed by the function *GenHeightOne*

4.1.2.2 Recurrence Relation for Counting the Number of Ladders of Order N with a Height of One

The recurrence relation of the number of ladders of order N is the same recurrence relation for other combinatorial objects such as the number of binary strings of length $N-1$ with no consecutive $1s$ and at least one 1 [**?**][**?**]. The recurrence relation is used to prove the veracity of the *GenHeightOne* algorithm.

Theorem 4.1.3 *The recurrence relation for the number of ladders of order N with a height of* 1 *is:*

$$\begin{cases} L_{count}(0) = 0 & N = 0 \\ L_{count}(1) = 0 & N = 1 \\ L_{count}(N) = L_{count}(N-1) + L_{count}(N-2) + 1 & N \geq 2 \end{cases}$$

Proof. We shall do a combinatorial proof to demonstrate the above theorem.

Suppose we want to count all binary strings of length N such that there can be no consecutive $1s$ and there must be at least one 1 in the string. Suppose we are counting $1s$ from right to left. Suppose the first 1 in a binary string of length N is at position N, then the second 1 can appear at position $N-2$, thus we have

binary strings of length N with the first 1 appearing at position N and the second one appearing at position $N-2$; let m = the number of binary strings of length N such that there is a 1 at position $N-2$. Next, suppose a binary string of length N has a 0 at position N, then the first 1 can appear at position $N-1$ or position $N-2$. If it appears at position $N-2$ we have binary strings of length N with a 1 at position $N-2$. We already designated this number as m, so we get $2(m)$. Still supposing we are considering binary strings of length N with a 0 at position N, consider all binary strings of length $N-1$ with no consecutive $1s$. Let k = the number of binary strings of length $N-1$ with no consecutive $1s$ and at least one 1. Let the first 1 in the binary string of length N appears at position $N-1$, then we have $2(m)+k$. Still assuming a 0 at position N in binary strings of length N, if there is also a 0 at position $N-1$, then the first 1 can appear at position $N-2$. The number of binary strings of length N with a 1 at position N was designated as m. Thus we have $2(m)+m=3m$ Yet we have already counted m under the conditions that the first 1 in binary strings of length N appears at position $N-2$. Therefore we subtract m from k thus leaving us with j = the number of binary strings of length N with the first 1 appearing at position $N-1$. Now we have $2(m)+j$. T hen consider all binary strings of length N such that from positions $1 \ldots N-1$ there are only $0s$. Therefore, there must be a 1 at position N seeing as we are considering all binary strings of length N with at least one 1. Only one such binary string of length N exists, therefore we add one. We get $2(m)+(k-m)+1 = 2(m)+j+1$ = the number of binary strings of length N with at least one 1 and no consecutive $1s$.

Now consider a ladder, L, with $N+1$ lines. The number of columns in L is N and the height of L is one. Note that the end points of no two bars can be touching which is to say that there can be no adjacent bars on the same row. For example, if there is a bar at row 1, column N then the next consecutive bar in row 1 can appear at most at column $N-2$. Knowing this, we can easily see how this scenario models all binary strings of length N with no consecutive $1s$ and having at least one 1. Let a

bar in l be represented as a 1 in a binary string of length N. Knowing that a ladder with zero bars has a height of zero, it must be the case that l has at least one bar. Thus, we get the same recurrence relation for the number of ladders of order $N+1$ where m is the number of ladders with a bar appearing in column $N-2$, $(k-m)=j$ being the number of ladders with the first bar in column $N-1$ minus ladders with a bar appearing at $N-2$. Lastly is the $+1$ for all ladders of order $N+1$ where the only bar appears at column N. See Fig.**??** for the mapping of binary strings of length $N=5$ with no consecutive $1s$ and at least one 1 to ladders of order 6 with a height of one.

□

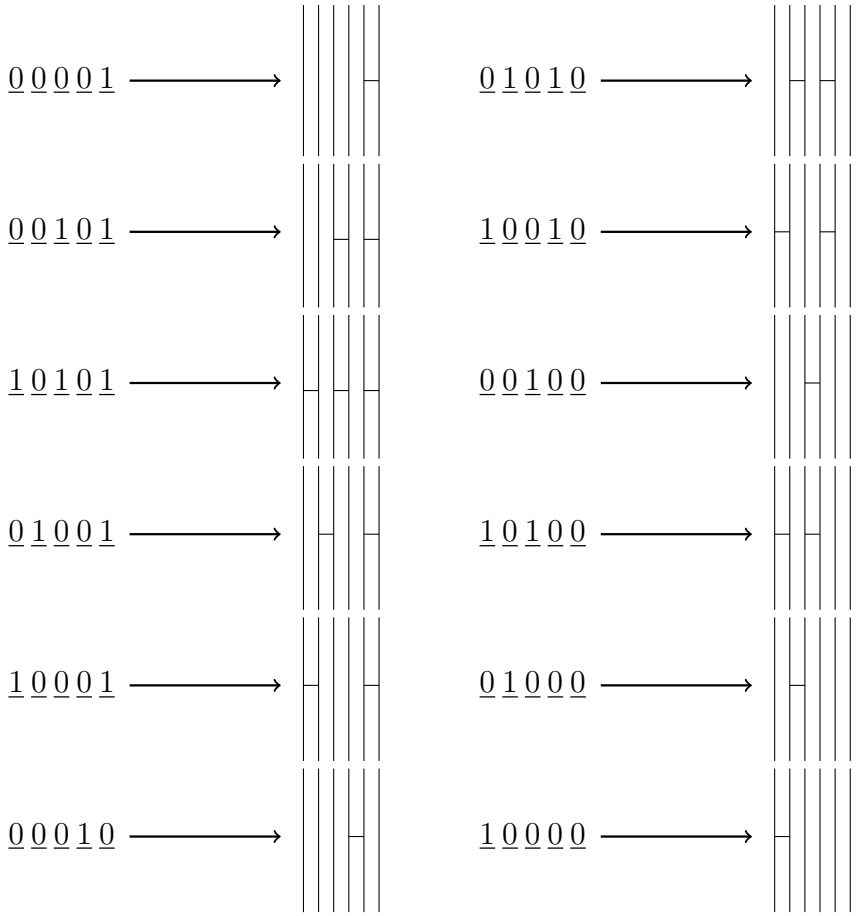


Figure 4.4: All 12 binary strings of length 5 with at least one 1 and no consecutive $1s$ maps to all twelve ladders of order 6 with a height of one. The recurrence relation being $L(6) = 2L(4) + (L(5) - L(4)) + 1 = L(4) + L(5) + 1$

4.1.2.3 Closed form Formula for Ladders of Order N with a Height of One

Before providing the closed form formula for the number of ladders with a height of one, it is important to connect ladders with a height of one to other mathemati-

cal phenomena because ladders with a height follow the same pattern as these other mathematical phenomena [**?**]. One of these phenomena include the number of involutions in the Symmetric Group S_N [**?**]. The connection between the Symmetric Group, S_N and ladders of order N with a height of zero or one will be analyzed followed by the closed form formula.

A *group* is a finite set along with a binary operation on the elements of the set such that the binary operation on two elements in the set produces a result that is also in the set. The stipulations of a group are the following. Firstly, the group must have *closure* which means the result of the binary operation produces a result in the set; this stipulation was already addressed in the definition of a group. Secondly, the group must be associative, meaning the rearrangement of priority of the order of application of the binary operation across $2 \leq K \leq N$ elements in the set does not change the result; associativity means the order of application of the binary operation across multiple elements in the set does not change the result. The third stipulation is that the set has the identity element. The identity element is the element such that when the binary operation is applied to an element, x, with the identity element the result is x. The fourth stipulation is the *inverse element* which is a relation between two elements, x and y such that when the binary operation is applied to x and its inverse element y, the result is x [**?**].

A *symmetric group of order* N/S_N is finite group whose elements are all $N!$ permutations of order N, the binary operation is permutation composition, applying the composition forms a bijection between all elements in the set. When a permutation is written in cycle notation, the *orbit* is defined as the transposition of elements on the identity permutation. For example, let $\pi = (3, 2, 1, 6, 4, 5, 7)$ be written as $(1, 3)(4, 5, 6)$ in cycle notation. There are two orbits, one of size two, namely $(1, 3)$ and one of size three, namely $(4, 5, 6)$ [**?**]. There Let an *involution* be defined as a composition of a permutation with itself such that the result of the composition is the identity permutation [**?**]. For example, $X = \{1, 2, 3\}$. Let

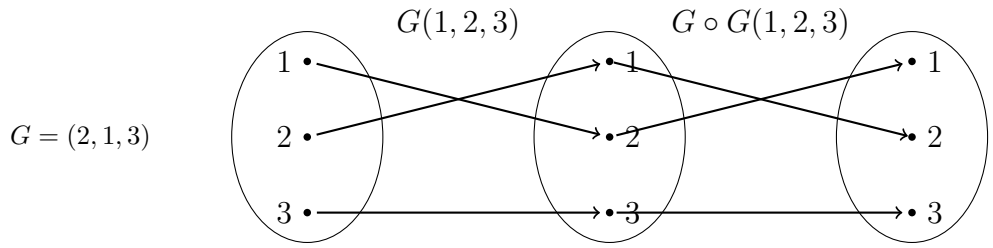


Figure 4.5: The involution $(2,1,3)$ composed with itself when applied to the identity permutation returns the identity permutation

$S_X = S_N = \{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$. The involutions of $S_N = \{(2,1,3),(1,3,2),(1,2,3)\}$. The reason these are the involutions is because when we define a permutation as a bijective function on the identity permutation, we can see the the composition of an involution with itself returns the identity permutation. Let $(1,2,3) = F$, let $(2,1,3) = G$ and let $(1,3,2) = H$ then we have $F \circ F = (1,2,3)$, $G \circ G = (1,2,3)$ and $H \circ H = (1,2,3)$. The orbit(s) of an involution are of size two and the orbits are transitive, meaning the elements composing the orbit(s) are adjacent in the identity permutation. To see an example of the mapping of the composition of $(2,1,3)$ with itself see Fig.**??**

Theorem 4.1.4 *There is a bijectie function between ladders of order N with a height of zero or one and the involution set of S_N.*

Proof. The involution set of S_N consists of all permutations of order N such that when composed with themselves, the result of the composition is the identity permutation. If a permutation is an involution it either has no inversions or for each pair of inversions, the inversion pairs are pairwise disjoint. That is to say, no element in the involution set forms more than 1 inversion. When an involution is applied to the identity permutation, each element in the identity is rotated by one or zero positions. If an element from the identity permutation is rotated two times over over a span of two positions, the element returns to its original position in the identity permutation.

Thus, applying an involution to itself either rotates an element zero times or it rotates an element twice over a span of two positions, thus placing the element in its original position in the identity permutation

A ladder of order N with a height of one consists only of bars such that no element crosses more than one of these bars. Suppose an element x needed to be cross more than one bar. This would mean the route of $x > 1$. If that is the case then the height of the ladder would be greater than one, which contradicts the claim that the ladder has a height of one. Then it must be the case that for all ladders of order N with a height of one, each bar in any of these given ladders uninverts a pair of elements in π exactly once and no two bars have the same element crossing them. It follows that each bar places an element in π to its correct position in the identity permutation. We know that each ladder with a height of zero or one is unique, in that they all sort different permutations, because the only way to get two or more ladders to sort the same permutation is to perform a swap operation on bar(s) in a ladder to get another ladder in $OptL\{\pi\}$. Yet with ladders of height zero or one, no swap operation can be performed.

Let $F(L_N)$ be the representation of a ladder as a function. We have already established that an involution can be thought of as a function. Let G be the representation of an involution as a function. We know that $G \circ G(ID) = ID$. I propose that for each G there is a corresponding $F(L_N)$ such that $F(L_N) \circ G(ID) = ID$ where each $F(L_N)$ and G are unique. This shall be proven by way of contradiction. Suppose there exists a G of order N such that $F(L_N) \circ G(ID) \neq ID$ for every $F(L_N)$. Let this G be known as K. We know that the number of ladders with a height of zero or one of order N equals the number of involutions of order N. We also know that each bar in $F(L_N)$ of order one uninverts an inversion in π We know that each G is unique seeing as the involution set $\subset$ all $N!$ permutations and all $N!$ permutations are unique. Thus, if $F(L_N) \circ K(ID) \neq ID$ for every $F(L_N)$ of a height of zero or one that means either there is a $F(L_N)$ of height zero or one that does not map K to ID

when composed with K or there exists at least two $F(L_N)$ of height zero or one that map the same $G \neq K$ to ID when composed with G. In the first case this would mean that there is some involution, K, that could not be sorted into the identity permutation by any $F(L_N)$ of height zero or one. Yet if that is the case then there is an $F(L_N)$ of height zero or one such that there exists a bar in $F(L_N)$ that does not place the element crossing it into its correct position in ID. But if that is the case then F_{L_N} does not have a height of zero or one which is a contradiction. In the second case, this would mean that there are two $F(L_N)$ with a height of zero or one, let us call them A and B, and some $G \neq K$, such that $A \circ G(ID) = B \circ G(ID) = ID$ and $A \neq B$. Yet we know that the only way for two unique ladders to sort the same permutation is by right/left swapping the bars of one ladder to get the configuration of the bars in the other ladder. Yet a bar cannot be right/left swapped in a ladder with a height of zero or one. Thus, if $A \circ G(ID) = B \circ G(ID) = ID$ it must be the case that $A = B$ which is a contradiction. Therefore, for each G there exists an $F(L_N)$ such that $F(L_N) \circ G(ID) = ID$ where each $F(L_N)$ and G are unique. To see the bijective mapping between ladders of order 4 with a height of zero or one and the involution set of S_4 please refer to Fig.**??** □

So far we have demonstrated that ladders of order N with a height of one are congruent with binary strings of length $N-1$ with no consecutive ones and at least one 1 and the involution set of S_N. The final mathematical phenomena to be discussed is the *Fibonacci sequence*. The Fibonacci sequence is the sequence $0, 1, 1, 2, 3, 5, 8, 13, \ldots$. It is defined by the recurrence relation:[**?**]

$$\begin{cases} Fib(0) = 0 & N = 0 \\ Fib(1) = 1 & N = 1 \\ Fib(N) = Fib(N-1) + Fib(N-2) & N \geq 2 \end{cases}$$

The Fibonacci sequence is considered famous for its occurrence in natural phenomena such as the structure of a pine cone, the number of petals on sunflowers and the spiral of the shells of ammonites which are a prehistoric crustaceans. The Fibonacci sequence was first discovered by an Indian mathematician named Pingala at some time between 450 - 200 BCE [**?**]. It was then introduced to Western cultures in 1202 by the Italian mathematician, Fibonacci. The recurrence relation for the Fibonacci numbers should look familiar to the recurrence relation for the number of ladders of order N with a height of one. The difference between the two is the recurrence relation for the number of ladders of order N with a height of one has an additional $+1$ because of the single ladder of order N in which the first $N-2$ columns have a 0 in row one and the last column has a 1 in row one. Other than the additional $+1$, the sequences are the same. Please refer to Fig.**??** to see the sequences together.

From looking at the sequences in Fig.**??**, it is interesting to note that $L_{count}(N) = Fib(N+1) - 1$. There is a well known equation for the Fibonacci sequence which is the following:

$$Fib(N) = 1/\sqrt{5}((1+\sqrt{5}/2)^n)((1-\sqrt{5}/2)^n))$$

[**?**] From the Fibonacci equation along with the equation $L_{count}(N) = Fib(N+1) - 1$, it is fairly straightforward to derive the equation for $L_{count}(N)$

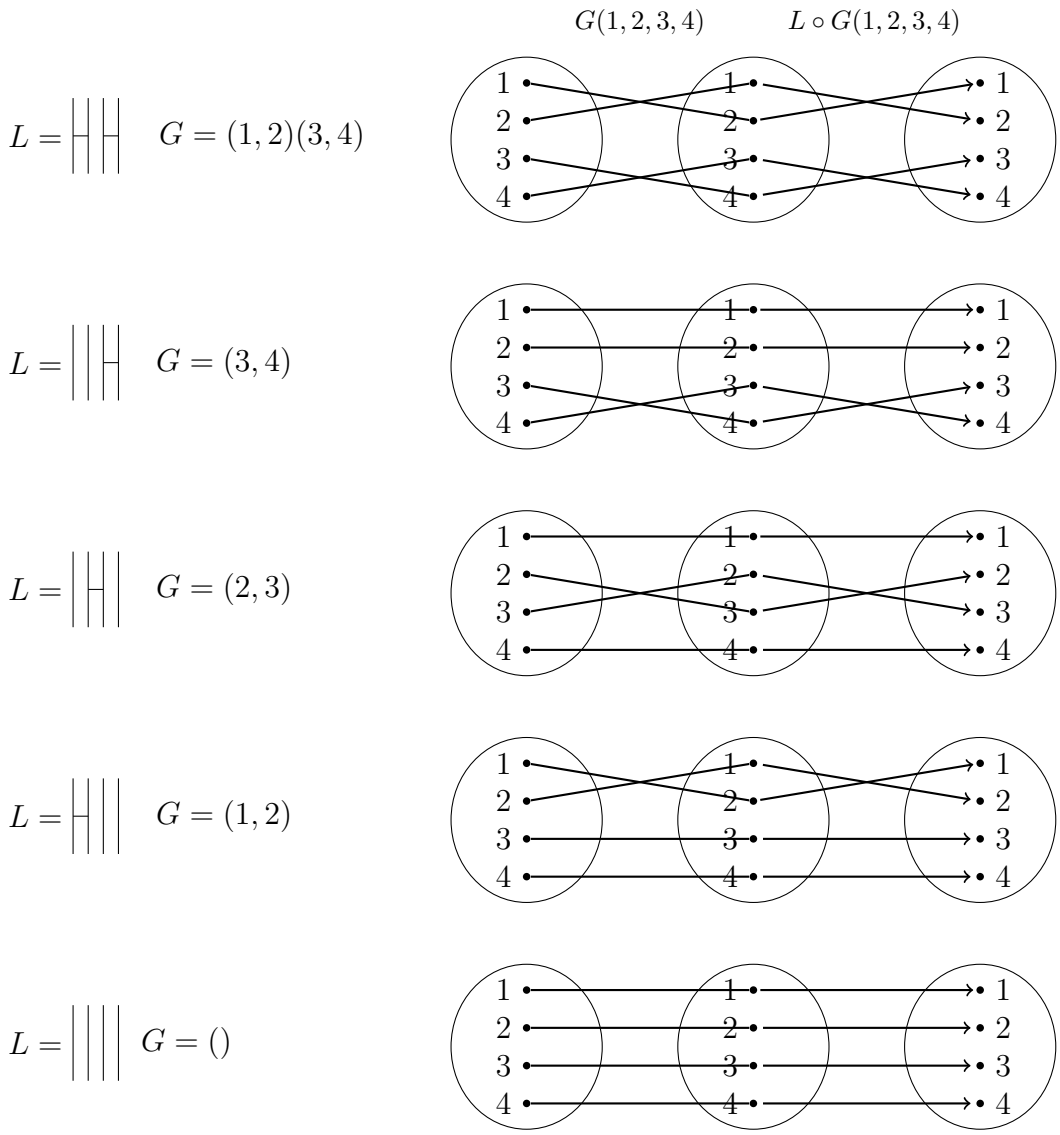


Figure 4.6: All ladders of order 4 with a height of zero or one form a bijection with the involution set of S_4.

$$
\begin{aligned}
Fib: &\quad 00, 01, 01, 02, 03, 05, 08, 13, 21, 34, 55, \ldots \\
L_{count}: &\quad 00, 00, 01, 02, 04, 07, 12, 20, 33, 54, 88, \ldots \\
N =: &\quad 00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, \ldots
\end{aligned}
$$

Figure 4.7: The Fibonacci sequence lined up with the sequence for the number of ladders with a height of one.

$$L_{count}(N) = 1/\sqrt{5}((1+\sqrt{5}/2)^{n+1})((1-\sqrt{5}/2)^{n+1})) - 1$$

The equation is simply the closed form formula for the $N+1th$ Fibonacci number minus 1.

We have examined the congruence between ladders of a height of one with three other mathematical phenomena. The three being binary strings of length $N-1$ with at least one 1 and no consecutive $1s$, the involution set of the symmetric group S_N and the Fibonacci sequence. The similarities between these mathematical phenomena is of theoretical interest because it has been shown that the set containing all mathematical phenomena that follow the sequence $0, 0, 1, 2, 4, 7, 12, 20\ldots$ also contains ladders with a height of one.

4.2 Procedure

In the procedure section a heuristic algorithm is provided for the Minimum Height Problem. Recall that the Minimum Height Problem asks, given some π is there an algorithm for creating a minimal ladder from $MinL\{\pi\}$? Before providing the heuristic algorithm, it must be stated that there is an exact procedure to generate a minimal ladder from each $MinL\{\pi\}$ from $MinL\{\pi_N\}$. Refer to the minimal ladder for the reverse permutation of order N as $MinL(Rev(\pi_N))$. In the introduction of this chapter, there is a description of a removal sequence of bars from $MinL(Rev(\pi_N))$ resulting in one minimal ladder for each $MinL\{\pi\}$ from $MinL\{\pi_N\}$. However, this method for creating a minimal ladder for an arbitrary permutation of order N is inefficient. Using this method on some arbitrary permutation π would first require creating $MinL(Rev(\pi_N))$, then each bar in $MinL(Rev(\pi_N))$ that does not correspond to an inversion in π would need to be removed from the $MinL(Rev(\pi_N))$. The resulting ladder is a minimal ladder from $MinL\{\pi\}$. To see an example of the exact procedure for creating a minimal ladder, given some arbitrary π of order N please refer to Fig.**??**. To see the algorithm for creating $MinL(Rev(\pi_N))$ please refer to Alg.**??**.

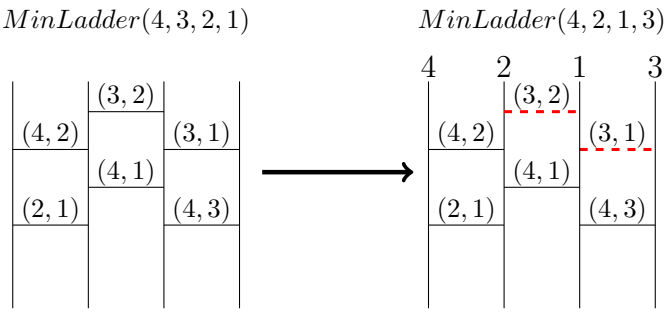


Figure 4.8: Given $\pi = (4, 2, 1, 3)$, the exact procedure first cs cs a min ladder for $(4, 3, 2, 1)$ then removes bars to c a min ladder for $(4, 2, 1, 3)$

4.2.1 Algorithm to create $MinL(Rev(\pi_N))$

Let *ladder* be initialized to an empty two dimensional array. Let N be initialized to $[N]$. Let *Row* be initialized to 2. Let *Col* be initialized to 1. Let *elem* be initialized to N. The goal is to c $MinL(Rev(\pi_N))$. The bars for the Nth element begin at $row = 2$, $col = 1$ and span to $row = 2 + (N - 1)$, $col = (N - 1)$. Each element equal to $N - 2k$ where $0 \leq k \leq floor(N/2)$ indicates an element with the same polarity as N and $\leq N$. We know from the introduction to this chapter that the routes of the same polarity as N remain below the route of N. We also know that the first bar of each of these routes begins at column 1.

To calculate the row of the first bar of the jth element where j has the same polarity as N consider the following. Let the previous element with the same polarity as N be referred to as $j+2$. Let the row of the first bar of the route of $j+2$ be equal to m. Thus, $ladder[m][1]$ is the first bar of the route of element $j+2$. We know that the second bar of the route of $j+2$ goes in $ladder[m+1][2]$. Thus, the first bar of the jth route must begin at $row = m + 2$; it cannot go in $ladder[m][1]$ seeing as the first bar of $j + 2$ occupies this cell. Nor can it go in $ladder[m + 1][1]$ seeing as the second bar of the route of $j+2$ is at $ladder[m+1][2]$. Nor can it go in a $row > m+2$ seeing as the ladder would no longer be minimal. Thus, the first bar of the route of element j is at $ladder[m + 2][1]$ where m is the row of the first bar of the route of element $j + 2$.

Each element equal to $N - (2k + 1)$ where $0 \leq k \leq floor(N/2)$ indicates an element with the opposite polarity as N and $\leq N$. We know from the introduction to this chapter that the routes of elements with the opposite polarity of N are right swapped above the routes of all elements gr than

Algorithm 17 Algorithm for creating $MinL(Rev(\pi_N))$

```
1: function CMinRL(ladder, N, row, col, elem)
2:     if elem = 0 OR elem = 1 then
3:         return
4:     end if
5:     if elem = N then
6:         CMinRL(ladder, N, row ← row + 2, col ← 1, elem ← elem − 2)
7:         CMinRL(ladder, N, row ← 1, col ← 2, elem ← elem − 1)
8:     else
9:         if N − elem = 2k then
10:            CMinRL(ladder, N, row ← row + 2, col ← 1, elem ← elem − 2)
11:        else
12:            CMinRL(ladder, N, row ← 1, col ← col + 2, elem ← elem − 2)
13:        end if
14:    end if
15:    r ← row, c ← col
16:    for i ← 1, i < elem, i ← i + 1 do
17:        ladder[r][c] ← 1
18:        r ← r + 1, c ← c + 1
19:    end for
20: end function
```

themselves. We also know from the introduction that the first bar of each of these routes begin at row 1 in $MinL(Rev(\pi_N))$. The number of bars in each of these elements' routes equals $j-1$ where j is one of these elements. Lastly, we know from the introduction of this chapter, that the last bar of each of these elements' routes end at column $N-1$/the last column in $ladder$.

To calculate the column of the first bar of the jth element where j has the opposite polarity as N consider the following. Let the previous element with the opposite polarity as N be referred to as $j+2$. Let the column of the first bar of the route of $j+2$ be equal to m. Thus, $ladder[1][m]$ is the first bar of the route of element $j+2$. We know that the first bar of the route of element j must begin at row 1. This bar cannot go in $ladder[1][m]$ seeing as this is where the first bar of the route of element $j+2$. Nor can this bar go in $ladder[1][m+1]$ seeing as if it did, its left endpoint would be touching the right endpoint of the first bar of the route of element $j+2$. Nor can it go in $ladder[1][m+>2]$ seeing as if it did, the bars of route j would extend beyond the last/$N-1th$ column in $ladder$.

Lemma 4.2.1 *The column for the first bar of element $j's$ route is $m+2$.*

Proof.

- Let the column of the first bar of element $j+2's$ route be m.
- Let the number of columns in $ladder = N-1$.
- Let the number of bars in $j+2's$ route be equal to $j+2-1 = j+1$.

From the axioms we get the equation $N-1-(j+1)=m$. We need to derive $N-1-(j-1)=m+2$ from $N-1-(j+1)=m$.

$$\begin{aligned} (N-1)-(j+1) &= m \\ (N)-(j) &= m+2 \\ (N-1)-(j) &= m+2-1 = m+1 \\ (N-1)-(j-1) &= m+2-1+1 = m+2 \\ (N-1)-(j-1) &= m+2 \end{aligned} \tag{4.1}$$

Thus, the first bar for $j's$ route is $m+2$. End of proof. To see $MinL((5,4,3,2,1))$ please refer to Fig.**??**. □

Lemma 4.2.2 *The time complexity of $CMinRL$ is $O\binom{n}{2}$*

Proof. For each element, x, in $Rev(\pi_N)$, the function makes a recursive call and the function adds all $x-1$ bars belonging to $x's$ route in the ladder. The total number of bars for the $MinL(Rev(\pi))$ equals the number of inversions for $Rev(\pi_N)$ which is equal to $\binom{n}{2}$. End of proof. □

4.2.2 The Heuristic Algorithm to create $MinL(\pi)$

In the previous section, the algorithm to c $MinL(Rev(\pi))$ was provided. The algorithm is exact, but unfortunately only cs the minimal ladder for the reverse permutation. The following algorithm is a heuristic algorithm for creating a minimal ladder for any permutation. The heuristic algorithm is based on inserting the maximum the number of bars per row of the ladder. Each bar uninverts and inversion, two or more bars on the same row uninvert two or more inversions in parallel. Thinking back to sorting networks, when two or more connectors are directly above/below each other, the connectors swap elements in tandem. The same can be said for bars on the same row of the ladder. Let *ladder* be a ladder lottery for some π. One can say if *ladder* has a height of one, then it sorts π into π_{ID} in one step. If *ladder* has a height of two, then it sorts π into an intermediary π_2 in row 1 then sorts π_2 into π_{ID} in row 2, etc. Define *bar compression* as the average number of bars per row in *ladder*; if the ladder has zero rows and/or zero bars, then the bar compression is undefined. The more bars per row the higher the bar compression, the less bars per row the lower the bar compression. The heuristic algorithm works by maximizing bar compression of *ladder*. It should be intuitive that given a *ladder* with b bars, each bar could be given its own row; in this case the *ladder* would have the least bar compression. This *ladder* is the opposite of the minimal ladder. Thus, for the heuristic algorithm, the goal is to squeeze as many bars in the same row as possible in order to maximize the bar compression of *ladder*.

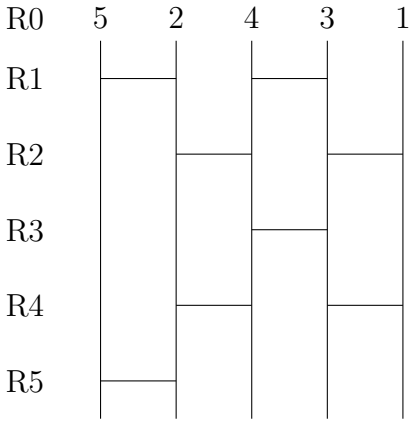
Recall that when inverted elements of π travel through the ladder and cross a bar, the elements are swapped, thus resulting in some intermediate permutation π_k. Define $InvPi(\pi)$ as the permutation of intermediate permutations, beginning at π and ending at the identity permutation. Each $\pi_k \in InvPi(\pi)$ corresponds to a row from a unique *ladder* from $OptL\{\pi\}$. Given some arbitrary π, there can be more than one $InvPi(\pi)$. See table **??**. for two different $InvPi(3,5,4,6,2,1)$.

When creating a $MinL(\pi)$, the goal is to c a *ladder* with the least number of rows, which in turn corresponds to the shortest $InvPi(\pi)$. Let $MinInvPi(\pi)$ be the $InvPi(\pi)$ generated by the heuristic algorithm. $MinInvPi(\pi)$ is not unique. Let $\pi_k \in MinInvPi(\pi)$. Then the recurrence relation for $\pi_k = \pi^{k-1} \rightarrow \tau(\pi_i^{k-1}, \pi_{i+1}^{k-1}) | \pi_i^{k-1} > \pi_{i+1}^{k-1}$ and for any $\tau(\pi_i^{k-1}, \pi_{i+1}^{k-1}) \cap \tau(\pi_j^{k-1}, \pi_{j+1}^{k-1}) =$

$2\ InvPi(3,5,4,6,2,1)$		
π_k	$A = InvPi(3,5,4,6,2,1)$	$B = InvPi(3,5,4,6,2,1)$
π_1	$(3,5,4,6,2,1)$	$(3,5,4,6,2,1)$
π_2	$(3,4,5,6,1,2)$	$(3,4,5,6,2,1)$
π_3	$(3,4,5,1,6,2)$	$(3,4,5,2,6,1)$
π_4	$(3,4,1,5,2,6)$	$(3,4,2,5,6,1)$
π_5	$(3,1,4,2,5,6)$	$(3,2,4,5,1,6)$
π_6	$(1,3,2,4,5,6)$	$(2,3,4,1,5,6)$
π_7	$(1,2,3,4,5,6)$	$(2,3,1,4,5,6)$
π_8	$None$	$(2,1,3,4,5,6)$
π_9	$None$	$(1,2,3,4,5,6)$

Table 4.1: Table for two different InvPi(3,5,4,6,2,1)

$\emptyset$. In simpler terms, for some $\pi_k \in MinInvPi(\pi)$, perform the maximum number of adjacent transpositions on adjacent inversions in π_{k-1} as is possible. In turn, this means the maximum number of bars can be added to the *kth* row in *ladder*. The less rows there are in the *ladder* the smaller the corresponding $InvPi(\pi)$ and the gr the bar compression. To see an example of maximal bar compression and a corresponding $MinInvPi(\pi)$, please refer to Fig.**??**.



$R0 \quad (5,2,4,3,1) \rightarrow (5,2,4,3,1)$

$R1 \quad (5,2,4,3,1) \rightarrow (2,5,3,4,1)$

$R2 \quad (2,5,3,4,1) \rightarrow (2,3,5,1,4)$

$R3 \quad (2,3,5,1,4) \rightarrow (2,3,1,5,4)$

$R4 \quad (2,3,1,5,4) \rightarrow (2,1,3,4,5)$

$R5 \quad (2,1,3,4,5) \rightarrow (1,2,3,4,5)$

Figure 4.9: The ladder on the left has a bar compression of 8/5. The corresponding $InvPi((5,2,4,3,1))$ is on the right.

Algorithm 18 Heuristic algorithm to c a ladder with minimal height

```
1: function HeuristicMinLadder(ladder[N][N − 1], π, N, row ← 1)
2:     if Sorted(π) then
3:         return
4:     end if
5:     if row = 1 then
6:         π_2 ← PreProcessRowOne(π, N)
7:         for i ← 1, i ≤ N, i ← i + 1 do
8:             if p_i ≠ π'_i then
9:                 ladder[1][i] ← 1
10:                i ← i + 1
11:            end if
12:        end for
13:        HeuristicMinLadder(Ladder, π ← π^2, N, row ← row + 1)
14:    else
15:        for i ← 1, i < N, i ← i + 1 do
16:            if p_i > π_{i+1} then
17:                Swap(πi, π_{i+1})
18:                ladder[row][i] ← 1
19:                i ← i + 1
20:            end if
21:        end for
22:        HeuristicMinLadder(ladder, π, N, row ← row + 1)
23:    end if
24: end function
```

Algorithm 19 Algorithm to return the second permutation from $InvPi(\pi)$ which will result in the maximal bar compression

1: **function** PREPROCESSROWONE(π, N)
2: $\pi' \leftarrow \pi$
3: **for** $p_i > \pi_{i+1} \cdots > \pi_{i+2k+1} \in \pi$. **do**
4: $\tau(\pi_i, \pi_{i+1}), \tau(\pi_{i+2}, \pi_{i+3}) \ldots \tau(\pi_{i+2k}, \pi_{i+2k+1})$
5: **end for**
6:
7: $\pi'' \leftarrow \pi$
8: $\pi''' \leftarrow \pi$
9: **for** $\pi'_i > \pi'_{i+1} \cdots > \pi_{i+2k} \in \pi'$ **do**
10: $\tau(\pi''_i, \pi''_{i+1}), \tau(\pi''_{i+2}, \pi''_{i+3}) \ldots \tau(\pi''_{i+2k-2}, \pi''_{i+2k-1})$
11: **end for**
12: **for** $\pi'_i < \pi'_{i-1} \cdots < \pi_{i-2k} \in \pi'$ **do**
13: $\tau(\pi'''_i, \pi'''_{i-1}), \tau(\pi'''_{i-2}, \pi'''_{i-3}) \ldots \tau(\pi'''_{i-2k+2}, \pi'''_{i-2k+1})$
14: **end for**
15: **if** π'' and π''' are equally zig-zaggy **then**
16: return π'''
17: **else**
18: return $ZigZag(\pi'', \pi''')$ where $ZigZag$ returns the permutation that is most zig-zaggy.
19: **end if**
20: **end function**

Let *ladder* be initialized to the empty ladder. Let π be some arbitrary permutation of order N. Let π_k be a permutation from $InvPi(\pi)$. The function $PreProcessRowOne$ transposes as many adjacent inversions in π as possible. This corresponds to adding as many bars to row one *ladder* as is possible. Define a *decreasing subsrting of* π *(DSS for short)* as follows: given some value k $2 \leq k \leq N$, a decreasing substring of length k is defined as $p_i > \pi_{i+1} > \pi_{i+2} \cdots > \pi_k$. A DSS in π can be even or odd; the polarity of the DSS is defined by the length of the substring. For example, the DSS $(3, 2, 1)$ has odd polarity whereas the DSS $(4, 3, 2, 1)$ has even polarity. A DSS terminates when ceases to be an adjacent inversion. For example, given $(3, 2, 1, 4)$ in π, the DSS is $(3, 2, 1)$ which has odd polarity. $PreProcessRowOne$ returns the π_2 from some $MinInvPi(\pi)$; the

first permutation being $\pi_1 = \pi$.

There are two criteria for π_2. The first is π_2 is a transformation of π such that π has undergone as many adjacent transpositions as possible. The second criteria for π_2 is that π_2 is as zig-zaggy as possible, given that it has undergone the maximum amount of adjacent transpositions. *Maximal zig-zagginess* is defined in four cases.

$$\text{Max zig-zag} = \begin{cases} \pi_1 < \pi_2 > \pi_3 < \pi_4 \ldots \pi_{N-1=2k} > \pi_{N=2k+1} & \text{if } N = 2k+1 \text{ and } \pi_1 < \pi_2 \\ \pi_1 > \pi_2 < \pi_3 > \pi_4 \ldots \pi_{N-1=2k} < \pi_{N=2k+1} & \text{if } N = 2k+1 \text{ and } \pi_1 > \pi_2 \\ \pi_1 < \pi_2 > \pi_3 < \pi_4 \ldots \pi_{N-1=2K-1} < \pi_{N=2k} & \text{if } N = 2k \text{ and } \pi_1 < \pi_2 \\ \pi_1 < \pi_2 > \pi_3 < \pi_4 \ldots \pi_{N-1=2K-1} < \pi_{N=2k} & \text{if } N = 2k \text{ and } \pi_1 > \pi_2 \end{cases}$$

When a DSS is of even length, then there are $k/2$ adjacent inversions that can be uninverted in tandem, where k is the length of the DSS. For example, given DSS $(4, 3, 2, 1)$, $\tau(4, 3)$ and $\tau(2, 1)$ are done in tandem. When a DSS is of odd length, the maximum number of adjacent inversions that can be uninverted in tandem is $floor(k/2)$ where k is the length of the DSS. A choice needs to be made in terms of whether or not the first element in the DSS will be transposed or the kth element in the substring will be transposed. For example, given the DSS $(5, 4, 3, 2, 1)$, either $\tau(5, 4), \tau(3, 2)$ or $\tau(4, 3)\tau(2, 1)$ are legitimate options. The first step of $PreProcessRowOne$ is to perform $k/2$ $\tau(\pi_i, \pi_{i+1})$ in tandem for all even lengthed DSSs. Then, for all odd lengthed DSSs in π, $PreProcessRowOne$ performs $floor(k/2)$ $\tau(\pi_i, \pi_{i+1}), \ldots \tau(\pi_{k-2}, \pi_{k-1})$ resulting in a candidate permutation π''. $PreProcessRowOne$ then performs $floor(k/2)$ $\tau(\pi_k, \pi_{k-1}) \ldots \tau(\pi_3, \pi_2)$ resulting in a second candidate permutation π'''. This results in two candidate permutations for π_2. In order to choose between π'' and π''', the algorithm then checks for which of the two have a better zig-zag pattern; a better zig-zag pattern is a relation between two permutations such that if one permutation is closer to maximal zig-zagginess than the other, then it has a better zig-zag pattern. The reason the algorithm looks for better zig-zagginess is because the more zig-zaggy π_k is, the more adjacent 2 lengthed DSSs there are in π_k. The more adjacent 2 lengthed DSSs there are, the more pairwise disjoint adjacent inversions there are in π_k. The more pairwise disjoint adjacent inversions there are in π_k, the more bars can be added to *ladder* at row k. The result is likely to be $MinL(\pi)$ and the shortest lengthed $InvPi(\pi)$.

Lemma 4.2.3 *Given j adjacent inversions in π_k, the more of these inversions that are pairwise disjoint, the more bars can be added to the kth row.*

Proof. We shall use proof by induction. Let m be the number of elements it takes to c j adjacent inversions. Let n be the number of bars that can be added to the kth row of ladder. Inductive Hypothesis:

$$\text{m,n=:}\begin{cases} 2j, j \text{ if j adjacent inversions are pairwise disjoint} \\ j+1, ceil(j/2) \text{ if j adjacent inversions are not pairwise disjoint. Bars added right to left} \\ j+1, floor(j/2) \text{ if j adjacent inversions are not pairwise disjoint. Bars added left to right} \end{cases}$$

Base case 1: let $\pi = (4,3,2,1)/j = 2$. $(4,3) \cap (2,1) = \emptyset$ and $m = 2j = 4$ and $n = 2 = j$.
Base case 2: let $\pi = (3,2,1)/j = 2$. $(3,2) \cap (2,1) = \{2\}$ and $m = j+1 = 3$ and $n = 1 = ceil(j/2)$.
Base case 3: let $\pi = (3,2,1)/j = 2$. $(3,2) \cap (2,1) = \{2\}$ and $m = j+1 = 3$ and $n = 1 = floor(j/2)$.

We need to show that for $j+1$, $m = 2(j+1)$ and $n = j+1$ when the $j+1th$ adjacent inversion is pairwise disjoint. We also need to show that for $j+1$, $m = j+1+1$ and $n = ceil/floor(j+1/2)$ when the $j+1th$ adjacent inversion is not pairwise disjoint.

Suppose the $j+1th$ adjacent inversion is pairwise disjoint and suppose the first j adjacent inversions are also pairwise disjoint, then this would require two more elements to form an inversion in π. The reason being is that if the $j+1th$ inversion was formed by one more element in π then the $j+1th$ inversion would not be pairwise disjoint. Let this element be referred to as x. We shall prove by contradiction that if x, on its own, forms the $j+1th$ adjacent inversion in π then inversion $j+1$ cannot be pairwise disjoint. Let $inv(a,b)$ be the jth inversion in π where $a > b$. If one were to insert x to the left a and $x > a$ then $inv(x,a) \cap inv(a,b) \neq \emptyset$. If one were to insert x to the right of b and $x < b$ then $inv(a,b) \cap inv(b,x) \neq \emptyset$. Therefore, we have a contradiction. Thus, if adjacent inversion $j+1$ is a pairwise disjoint, then 2 more elements are required to make the $j+1th$ adjacent inversion. Therefore, $m = 2(j+1) = 2j+2$ where the $+2$ accounts for the two more elements required to make the $j+1th$ inversion. $n = j+1$ seeing as j inversions are pairwise disjoint, then the j bars at row k in *ladder* are at least two columns away from every other bar corresponding to one of the j inversions. Since the $j+1th$ adjacent inversion is also pairwise disjoint from all other j inversions, then the bar corresponding to this inversion is also placed at least two columns away from the other bars in the kth row. Thus, $n = j+1$.

Suppose the $j+1th$ adjacent inversion is not pairwise disjoint and suppose the first j inversions are also not pairwise disjoint, then the $j+1th$ adjacent inversion would require one more element to

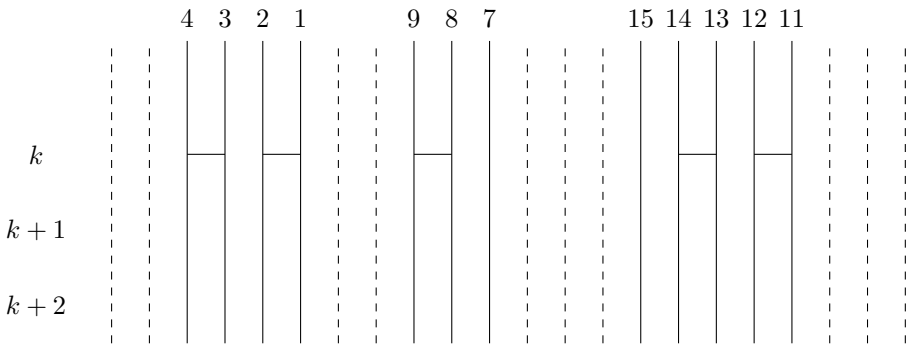
form an inversion in π. We shall do a direct proof to show that one more element is required. Let $inv(a, b)$ be the jth inversion in π where $a > b$. Let x be the element to form the $j + 1th$ inversion in π. If one were to insert x to the left a and $x > a$ then $inv(x, a) \cap inv(a, b) \neq \emptyset$. If one were to insert x to the right of b and $x < b$ then $inv(a, b) \cap inv(b, x) \neq \emptyset$. Therefore, in both cases, when x forms the $j + 1th$ inversion, the $j + 1th$ inversion is not pairwise disjoint, which is exactly what we are trying to prove. Since the xth element forms the $j + 1th$ inversion then $m = j + 1 + 1 = j + 2$ where the additional $+1$ comes from the xth forming the $j + 1th$ inversion.

Next, suppose that a is the leftmost element of the first j inversions. Let the position of a in $\pi = i$. Thus, $p_i = a$. Also, suppose that the first $j/2$ bars are added to $row = k$ going left to right and element x is directly to the left of a in π forming the $j+1th$ inversion. Seeing as element p_i and p_{i+1} have a bar on row k then elements x and a cannot have a bar on row k, thus $n = floor(j+1/2)$. Next, suppose that the first $j/2$ bars are added to $row = k$ going right to left, then it is possible to place a bar on row k for elements x and a, thus $n = ceil(j + 1/2)$. Next suppose that b is the rightmost element for the first j inversions. Let the position of b in $\pi = l$. Thus, $p_l = b$. Also, suppose that the first $j/2$ bars are added to $row = k$ going right to left and element x is directly to the right of b in π forming the $j + 1th$ inversion. Seeing as element p_{l-1} and p_l have a bar on row k then elements x and b cannot have a bar on row k, thus $n = floor(j + 1/2)$. Next, suppose that the first $j/2$ bars are added to $row = k$ going left to right, then it is possible to place a bar on row k for elements x and b, thus $n = ceil(j + 1/2)$.

Clearly $n = j > n = floor/ceil(j/2)$, therefore, the more adjacent inversions that are pairwise disjoint, the more bars can be added to *ladder* in row k. To see an example of the above proof please refer to Fig.**??**. End of proof.

□

From looking at Fig.**??**, one notices that when uninverting adjacent pairwise disjoint inversions, the result is a better zig-zag pattern. E.g. uninverting $(4, 3) \cap (2, 1)$ from $(4, 3, 2, 1)$ results in $(3, 4, 1, 2)$ which is more zig-zaggy than uninverting just the $(4, 3)$ or just the $(2, 1)$ which would result in $(3, 4, 2, 1)$ or $(4, 3, 1, 2)$ respectively. Given $(15, 14, 13, 12, 11)$ uninverting $(12, 11) \cap (14, 13)$ resulted in $(15, 13, 14, 11, 12)$ which is more zig-zaggy than if we were to uninvert $(15, 14)$ and $(12, 11)$ which would result in $(14, 15, 13, 11, 12)$. In sum, the heuristic algorithm is based on two assumptions. The first assumption is in order to c $MinL(\pi)$, $PreProcessRowOne$ performs the maximum number of transpositions of adjacent inversions in π. This leads to multiple candidate permutations.



DSS	j	m	n
$(4, 3, 2, 1)$	$2\ \{(4, 3)\} \cap \{(2, 1)\} = \emptyset$	4	2
$(9, 8, 7)$	$2\ \{(9, 8)\} \cap \{(8, 7)\} = \{8\}$	3	1
$(15, 14, 13, 12, 11)$	$4\ \{(15, 14)\} \cap \{(14, 13)\} \cap \{(13, 12)\} \cap \{(12, 11)\} = \{14, 13, 12, 11\}$	5	2

Figure 4.10: Figure demonstrating that the more pairwise disjoint adjacent inversions there are in π_k the more bars can be added to *ladder* at $row = k$

Once done, to determine which candidate permutation is the best option for π_2, determine which candidate is most zig-zaggy. This permutation is the permutation for π_2 in $MinInvPi(\pi)$. Then, from π_2, $HeuristicMinLadder$ uninverts as many adjacent inversions in every subsequent π_k in $MinInvPi(\pi)$. Bars are added to the *ladder* accordingly. Once complete, the resulting ladder is likely to be a $MinL(\pi)$.

4.3 Results

In the results section the heights of 105 ladders corresponding to 105 random permutations generated by the heuristic algorithm will be compared with the heights of 105 $MinL(\pi)$ corresponding to the same 105 random permutations generated by a brute force algorithm. To see the table for the results of the heuristic algorithm please see MinLadders.txt in the Appendix. For each test, the permutation is listed above the heights of the brute force ladder and the heuristic ladder. The experiment was conducted by running 15 tests on 15 unique, random permutations of order $N = [4 \dots 10]$. The accuracy of the heuristic algorithm is over 88%; 92 of the 105 test cases succeeded. When the heuristic algorithm was incorrect, it was off by only one row. Below you will see a table with the cases where the heuristic algorithm created an incorrect ladder. In the analysis section, Table **??**

will be analyzed and concluding remarks on the minimum height problem will be made.

4.3.1 Table with Results

Error Table of Heuristic Min Ladder		
Permutation	BF. Height	Heur. Height
2 4 5 3 1	4	5
5 3 1 6 2 4	4	5
7 3 1 2 5 6 4	6	7
6 5 3 1 4 7 2	6	7
5 6 1 3 7 2 4	5	6
7 3 4 1 5 6 2	6	7
3 6 4 1 7 5 2	5	6
6 1 3 4 7 8 5 2	6	7
3 5 8 2 7 6 1 4	6	7
9 3 2 4 6 8 5 1 7	8	9
7 5 1 8 9 6 2 3 4	7	8
3 1 5 6 9 10 7 2 8 4	6	7
10 2 8 1 4 9 3 6 5 7	9	10

Table 4.2: Table of the mismatches between the heuristic and the brute force algorithm

4.4 Analysis

In the analysis section we will be analyzing the errors resulting from the heuristic algorithm, discussing applications for the solutions provided to the Minimum Height Problem and discussing open problems regarding the Minimum Height Problem.

4.4.1 Analysis of Errors

Depending on the permutation, it is not always the case that making π^2 as zig-zaggy as possible results in a ladder with minimal height. Given the permutation $(2, 4, 5, 3, 1)$,
$PreProcessRowOne(2, 4, 5, 3, 1)$ returns $\pi^2 = (2, 4, 3, 5, 1)$. However, $\pi^2 = (2, 4, 3, 5, 1)$ does not

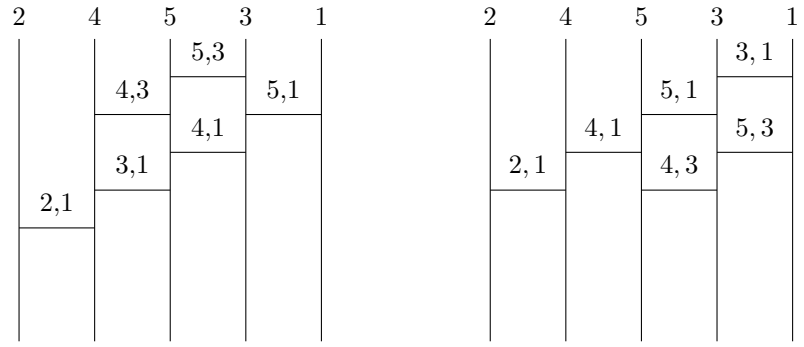


Figure 4.11: *HeuristicAlgorithm* ladder on the left, shortest ladder is on the right.

lead to a ladder with minimal height. Rather, the π^2 which leads to a ladder with minimal height is $(2, 4, 5, 3, 1)$. Notice how $(2, 4, 5, 3, 1)$ is less zig-zaggy than $(2, 4, 3, 5, 1)$. To see the ladder resulting from the *HeuristicAlgorithm* in comparison to the truly shortest ladder for the permutation $(2, 4, 5, 3, 1)$ please refer to Fig.**??**.

The assumption behind the *HeuristicAlgorithm* is that the more bars that are inserted into a given row, the shorter the ladder will be. Thus, the *HeuristicAlgorithm* is a greedy algorithm. Given a current row in the ladder, continue to insert as many bars into that row as possible. Once done, move to the next row. However, the greedy approach only leads to a locally optimal solution; each row is locally optimized given the current state of π^k. This does not necessarily lead to a globally optimal insofar as the resulting ladder may not be a $MinL(\pi)$. To see a table containing ladders from the *HeuristicAlgorithm* in comparison to the ladders created by brute force please refer to Fig.**??** in the appendix.

4.4.2 Open Problems Related to the Minimum Height Problem

- Is there a deterministic and efficient algorithm for creating $MinL(\pi)$ given some arbitrary π?

4.4.3 Applications

The *HeuristicAlgorithm* can be applied to other problems of compression. Most notable is creating the shortest $InvPi(\pi)$. Still thinking of what to add here. Not sure what this problem can be applied to.

Chapter 5

Evaluation

Chapter 6

Summary and Future Work

Conclude your thesis with a re-cap of your major results and contributions. Then outline directions for further research and remaining open problems.

Appendix A

Appendix

```
Input Permutation:  4  3  2  1 Brute Force: 4  Custom: 4
Input Permutation:  2  1  4  3 Brute Force: 1  Custom: 1
Input Permutation:  4  2  3  1 Brute Force: 3  Custom: 3
Input Permutation:  1  3  4  2 Brute Force: 2  Custom: 2
Input Permutation:  4  2  1  3 Brute Force: 3  Custom: 3
Input Permutation:  2  1  3  4 Brute Force: 1  Custom: 1
Input Permutation: 3  1  4  2 Brute Force: 2  Custom: 2
Input Permutation: 3  2  1  4 Brute Force: 3  Custom: 3
Input Permutation: 2  4  3  1 Brute Force: 3  Custom: 3
Input Permutation: 1  2  3  4 Brute Force: 1  Custom: 1
Input Permutation: 3  4  2  1 Brute Force: 4  Custom: 4
Input Permutation: 2  3  4  1 Brute Force: 3  Custom: 3
Input Permutation: 2  4  1  3 Brute Force: 2  Custom: 2
Input Permutation: 3  1  2  4 Brute Force: 2  Custom: 2
Input Permutation: 1  2  4  3 Brute Force: 1  Custom: 1
Input Permutation: 3  1  4  2  5 Brute Force: 2  Custom: 2
Input Permutation: 1  5  4  3  2 Brute Force: 4  Custom: 4
Input Permutation: 4  2  1  3  5 Brute Force: 3  Custom: 3
Input Permutation: 3  2  1  5  4 Brute Force: 3  Custom: 3
Input Permutation: 1  2  4  5  3 Brute Force: 2  Custom: 2
Input Permutation: 2  4  1  5  3 Brute Force: 2  Custom: 2
Input Permutation: 1  4  3  5  2 Brute Force: 3  Custom: 3
```

```
Input Permutation: 5  4  1  3  2 Brute Force: 5  Custom: 5
Input Permutation: 1  3  5  2  4 Brute Force: 2  Custom: 2
Input Permutation: 4  3  5  2  1 Brute Force: 5  Custom: 5
Input Permutation: 2  3  5  4  1 Brute Force: 4  Custom: 4
Input Permutation: 2  3  4  5  1 Brute Force: 4  Custom: 4
Input Permutation: 4  3  5  1  2 Brute Force: 4  Custom: 4
Input Permutation: 2  4  5  3  1 Brute Force: 4  Custom: 5
Input Permutation: 5  3  4  1  2 Brute Force: 4  Custom: 4
Input Permutation: 4  3  2  1  6  5 Brute Force: 4  Custom: 4
Input Permutation: 5  1  3  2  6  4 Brute Force: 4  Custom: 4
Input Permutation: 6  4  3  2  5  1 Brute Force: 5  Custom: 5
Input Permutation: 3  2  4  5  1  6 Brute Force: 4  Custom: 4
Input Permutation: 5  1  3  4  2  6 Brute Force: 4  Custom: 5
Input Permutation: 6  3  2  1  4  5 Brute Force: 5  Custom: 5
Input Permutation: 2  5  4  3  6  1 Brute Force: 5  Custom: 5
Input Permutation: 1  5  6  3  4  2 Brute Force: 4  Custom: 4
Input Permutation: 5  6  3  4  2  1 Brute Force: 6  Custom: 6
Input Permutation: 2  6  1  3  4  5 Brute Force: 4  Custom: 4
Input Permutation: 6  4  1  2  3  5 Brute Force: 5  Custom: 5
Input Permutation: 5  3  1  6  2  4 Brute Force: 4  Custom: 5
Input Permutation: 3  5  6  1  2  4 Brute Force: 4  Custom: 4
Input Permutation: 2  4  1  3  5  6 Brute Force: 2  Custom: 2
Input Permutation: 2  3  1  5  4  6 Brute Force: 2  Custom: 2
Input Permutation: 4  3  1  2  7  5  6 Brute Force: 4  Custom: 4
Input Permutation: 3  6  4  5  7  2  1 Brute Force: 7  Custom: 7
Input Permutation: 7  3  1  2  5  6  4 Brute Force: 6  Custom: 7
Input Permutation: 7  1  5  4  6  3  2 Brute Force: 6  Custom: 6
Input Permutation: 6  5  7  2  4  1  3 Brute Force: 6  Custom: 6
```

```
Input Permutation: 4  2  7  5  3  1  6 Brute Force: 5  Custom: 5
Input Permutation: 6  5  3  1  4  7  2 Brute Force: 6  Custom: 7
Input Permutation: 5  6  1  3  7  2  4 Brute Force: 5  Custom: 6
Input Permutation: 7  3  4  1  5  6  2 Brute Force: 6  Custom: 7
Input Permutation: 1  3  2  5  7  4  6 Brute Force: 2  Custom: 2
Input Permutation: 6  7  4  1  3  5  2 Brute Force: 6  Custom: 6
Input Permutation: 7  1  6  5  4  2  3 Brute Force: 6  Custom: 6
Input Permutation: 3  6  4  1  7  5  2 Brute Force: 5  Custom: 6
Input Permutation: 2  5  1  3  6  7  4 Brute Force: 3  Custom: 3
Input Permutation: 5  6  7  4  1  3  2 Brute Force: 7  Custom: 7
Input Permutation: 8  3  7  6  2  1  4  5 Brute Force: 7  Custom: 7
Input Permutation: 1  7  4  5  6  8  2  3 Brute Force: 6  Custom: 6
Input Permutation: 5  7  8  4  6  3  1  2 Brute Force: 8  Custom: 8
Input Permutation: 2  6  8  4  3  7  5  1 Brute Force: 7  Custom: 7
Input Permutation: 6  1  7  8  3  5  4  2 Brute Force: 7  Custom: 7
Input Permutation: 6  1  3  4  7  8  5  2 Brute Force: 6  Custom: 7
Input Permutation: 8  1  6  2  5  4  7  3 Brute Force: 7  Custom: 7
Input Permutation: 6  4  3  2  5  1  7  8 Brute Force: 5  Custom: 5
Input Permutation: 2  3  6  5  1  7  8  4 Brute Force: 5  Custom: 5
Input Permutation: 3  5  8  2  7  6  1  4 Brute Force: 6  Custom: 7
Input Permutation: 2  8  6  5  1  3  7  4 Brute Force: 6  Custom: 6
Input Permutation: 4  1  3  2  7  8  5  6 Brute Force: 3  Custom: 3
Input Permutation: 6  3  7  1  8  2  5  4 Brute Force: 5  Custom: 5
Input Permutation: 5  2  6  7  1  3  8  4 Brute Force: 5  Custom: 5
Input Permutation: 7  6  5  8  2  4  1  3 Brute Force: 7  Custom: 7
Input Permutation: 1  9  5  7  6  4  2  3  8 Brute Force: 7  Custom: 7
Input Permutation: 2  4  3  8  7  6  9  1  5 Brute Force: 7  Custom: 7
Input Permutation: 2  5  8  9  4  6  1  3  7 Brute Force: 6  Custom: 6
```

```
Input Permutation: 9  3  2  4  6  8  5  1  7 Brute Force: 8  Custom: 9
Input Permutation: 5  8  1  6  3  9  2  4  7 Brute Force: 6  Custom: 6
Input Permutation: 8  7  2  6  3  5  1  9  4 Brute Force: 8  Custom: 8
Input Permutation: 7  5  1  8  9  6  2  3  4 Brute Force: 7  Custom: 8
Input Permutation: 2  8  7  5  4  3  1  6  9 Brute Force: 7  Custom: 7
Input Permutation: 5  2  8  7  6  1  3  9  4 Brute Force: 6  Custom: 6
Input Permutation: 7  1  2  3  4  9  8  6  5 Brute Force: 6  Custom: 6
Input Permutation: 6  3  5  1  2  9  7  8  4 Brute Force: 6  Custom: 6
Input Permutation: 2  4  9  3  6  5  1  8  7 Brute Force: 7  Custom: 7
Input Permutation: 2  4  1  3  7  9  5  6  8 Brute Force: 3  Custom: 3
Input Permutation: 9  3  1  8  5  2  6  4  7 Brute Force: 8  Custom: 8
Input Permutation: 4  9  8  1  6  7  3  5  2 Brute Force: 8  Custom: 8
Input Permutation: 7  2  1  4  8  3  6  10  9  5 Brute Force: 6  Custom: 6
Input Permutation: 2  4  5  6  8  3  10  1  9  7 Brute Force: 7  Custom: 7
Input Permutation: 4  9  10  3  6  8  7  2  1  5 Brute Force: 9  Custom: 9
Input Permutation: 8  7  5  4  3  2  6  9  10  1 Brute Force: 9  Custom: 9
Input Permutation: 2  5  7  1  6  9  4  8  3  10 Brute Force: 6  Custom: 6
Input Permutation: 8  4  3  1  6  9  10  7  5  2 Brute Force: 8  Custom: 8
Input Permutation: 3  1  5  6  9  10  7  2  8  4 Brute Force: 6  Custom: 7
Input Permutation: 9  2  4  3  1  10  5  6  8  7 Brute Force: 8  Custom: 8
Input Permutation: 7  10  6  4  1  5  3  9  8  2 Brute Force: 9  Custom: 9
Input Permutation: 4  7  2  5  10  9  6  8  1  3 Brute Force: 8  Custom: 8
Input Permutation: 9  4  6  2  8  3  5  10  7  1 Brute Force: 9  Custom: 9
Input Permutation: 10  2  8  1  4  9  3  6  5  7 Brute Force: 9  Custom: 10
Input Permutation: 1  2  10  9  5  7  3  8  6  4 Brute Force: 8  Custom: 8
Input Permutation: 10  6  3  2  9  8  5  1  7  4 Brute Force: 9  Custom: 9
Input Permutation: 6  8  9  3  4  7  2  1  5  10 Brute Force: 8  Custom: 8
```

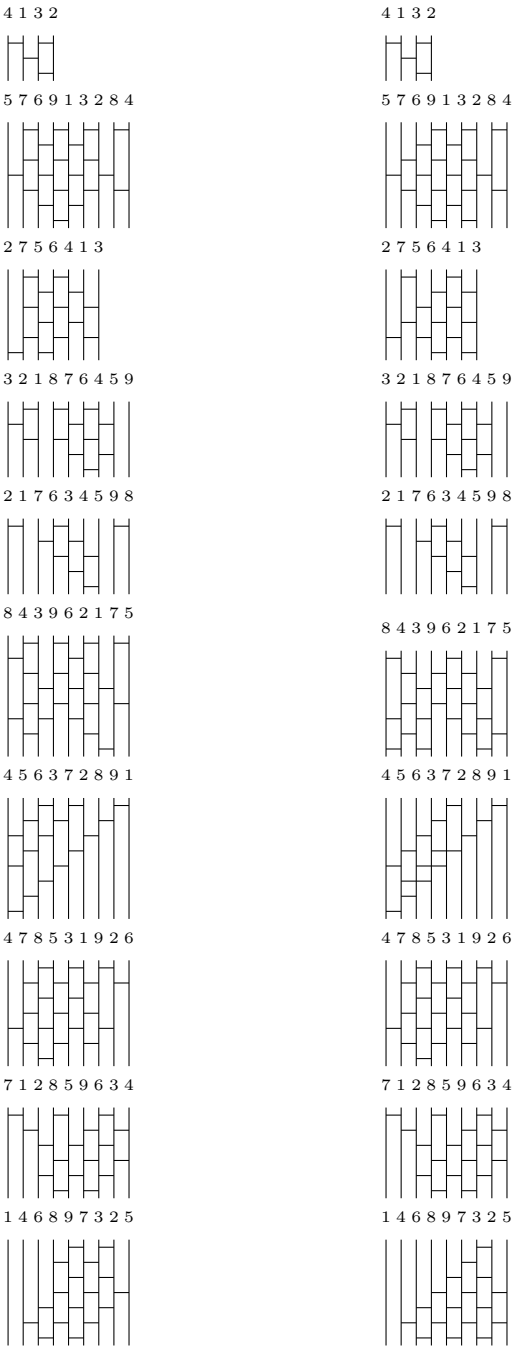


Figure A.1: Heuristic algorithm on the left, Brute Force on the right