# 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

MSc

in

Computer Science

Guelph, Ontario, Canada

# ABSTRACT

## Amidakuji: Counting, Listing and Optimization Algorithms

Patrick Di Salvo
University of Guelph, 2020

Advisor:
Dr. Joseph Sawada

Amidakuji, or ladder lotteries in English, are an abstract mathematical object which correspond to permutations. 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 multiple solutions to a problem pertaining to ladder lotteries; this problem is known as The Canonical Ladder Listing Problem. The solutions to this problem come in the form of novel algorithms, recurrence relations and formulas. The potential applications for ladder lotteries is also discussed in this thesis, along with a discussion of open problems related to ladder lotteries.

# 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 [34]. 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. We call such a tracing for a given student a *path*. For example, the path of Ryu is highlighted in red in Figure 1.1. The prize at the bottom of the ladder lottery is their prize [34]. See Figure 1.1 for an example of a ladder lottery.

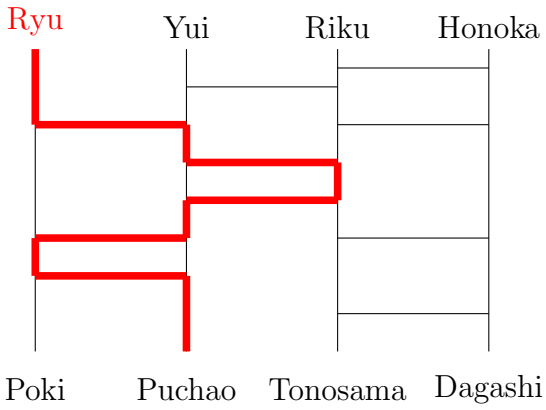



**Figure 1.1:** A ladder lottery where Ryu gets Puchao, Yui gets Dagashi, Riku gets Tonosama and Honoka gets Poki

The game of Amidakuji has an interesting history. Amida is the Japanese name for Amithaba, the supreme Buddha of the Western Paradise. Amithaba was a Buddha from India and there was a cult based around him. The cult of Amida, otherwise known as Amidism, believed that by worshiping Amithaba, they would enter into his Western 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 [18]. Amidakuji began in Japan during the Muromachi period, which spanned from 1336 to 1573 [18]. 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. Kuji is the Japanese word for lottery. Hence, the game was termed Amidakuji. In English, Amidakuji translates to ladder lottery.

An interesting property about a ladder lottery is that it is associated with a *permutation*. A *permutation* is a unique ordering of objects. For the purposes of this thesis, the objects of a permutation are $1, 2, \ldots, n$. Consider a permutation $\pi = (p_1, p_2, \ldots, p_n)$. A pair of elements, $p_i$ and $p_j$, form an *inversion* if $p_i > p_j$ and $i < j$. For example, given $\pi = (4, 3, 5, 1, 2)$, its set of inversions is
$Inv = \{(4, 3), (4, 2), (4, 1), (3, 2), (3, 1), (5, 1), (5, 2)\}$. A *transposition* is a swap of two elements in $\pi$. An *adjacent transposition* is defined as a swap of two adjacent elements in $\pi$. Given $\pi_1 = (4, 3, 5, 1, 2)$ and $\pi_2 = (3, 4, 5, 1, 2)$, they differ by the adjacent transposition of the elements $(4, 3)$. A ladder lottery corresponds to a permutation when:

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

2. At the bottom of the ladder lottery are the $n$ elements of $\pi$ in strictly ascending

order. For each element $x$ in $\pi$, $x$ goes down its path and ends up in the bottom $xth$ line.

### 1.0.1 Representation

In this thesis, ladder lotteries are represented as a two dimensional array. Let $n$ be the number of elements in $\pi$. Let a *column* be a gap between lines in the ladder lottery. Each ladder lottery represented as a two dimensional array has $n-1$ columns. Let a *row* be defined as a the horizontal span from column 1 to column $n-1$ containing at least one bar. Let a *ghost row* be defined as the horizontal span from column 1 to column $n-1$ containing zero bars. Let the *height* of the ladder be defined as the number of rows plus the number of ghost rows in the ladder. If two ladders are equivalent, then they sort $\pi$ with the exact same bars. If two ladders are identical, then they sort $\pi$ with the same exact same bars and with the same number of rows. In Figure 1.2, all three ladders are equivalent but only the leftmost ladder and middle ladder are identical with two rows. The rightmost ladder has three rows.

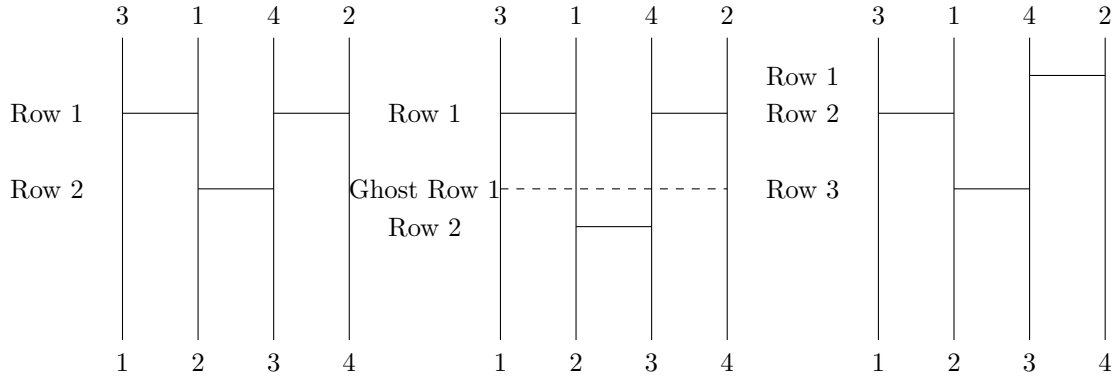



**Figure 1.2:** Three equivalent ladders. Leftmost and middle ladders are identical.

### 1.0.2 Optimal Ladder Lotteries

Let $k$ denote the number of inversions for some permutation. An *optimal ladder lottery* is a special case of ladder lottery, corresponding to $\pi$, in which there are $k$

bars in the ladder; one for each *inversion* in $\pi$ such that each bar uninverts a single inversion in $\pi$ exactly once [34]. An optimal ladder lottery sorts $\pi$ in ascending order using exactly $k$ bars by applying $k$ adjacent transpositions on $k$ inversions in $\pi$. For example, given $\pi = (4, 3, 5, 1, 2)$ an optimal ladder lottery associated with $\pi$ would have seven bars; one for each inversion in $\pi$. For each bar, two elements in $\pi$ that form an inversion cross the given bar. Once all elements have crossed their respective bars, $\pi$ is sorted in ascending order. 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 $\pi$. To see an example of two ladder lotteries associated with $\pi = (4, 3, 5, 1, 2)$, one optimal and one non-optimal, please refer to Figure 1.3. Let $OptL\{\pi\}$ be defined as the set of optimal

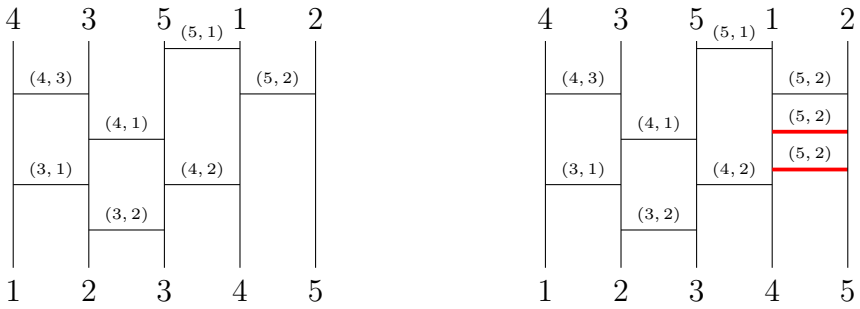



**Figure 1.3:** Two ladders for the permutation $(4, 3, 5, 1, 2)$. The left ladder is an optimal ladder and the right ladder is not. The bold bars in the right ladder are redundant, thus the right ladder is non optimal.

ladder lotteries that form an equivalence class, insofar as all ladders in $OptL\{\pi\}$ sort $\pi$ in ascending order with the least number of bars. The first discussion of $OptL\{\pi\}$ is in the paper Efficient Enumeration of Ladder Lotteries and its Application [34]. In this paper, the authors provide an algorithm known as FindAllChildren which generates $OptL\{\pi\}$; the details of the algorithm are discussed in Chapter 2. To see $OptL\{(4, 3, 5, 1, 2)\}$ please refer to Figure 1.4. Given that there are $n!$ permutations of order $n$, each of them have their own $OptL\{\pi\}$. In Table A.1 found in the appendix, the number of ladders in each of the 120 $OptL\{\pi\}$ of order 5 is presented.

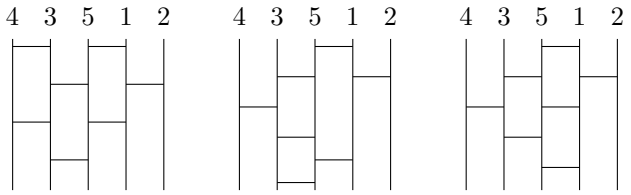



**Figure 1.4:** All the optimal ladders in $OptL\{(4,3,5,1,2)\}$

### 1.0.3 Combinatorial Generation

Our research on ladder lotteries pertains to research in *combinatorial generation*. Combinatorial generation is a subfield of theoretical computer science which lists all instances of combinatorial objects such as permutations, combinations, sets, subsets and graphs. Gray codes are a special type of combinatorial generation in which a constant amount of constant change is required to get from one object to the next. For example, a single swap operation, a single rotation, or a single bit flip is used to get from one object to the next. The term "Gray Code" comes from the Binary Reflected code for binary strings given by Frank Gray [11]. There are many known algorithms for listing permutations of order $n$. Throughout the course of this thesis, a number of such algorithms were researched [16, 3, 6, 12, 13, 8, 35, 28, 20, 1, 15]. Each of these algorithms will be reviewed in Chapter 2. To see all 24 ladders listed in lexicographic order please refer to Figure 1.5.

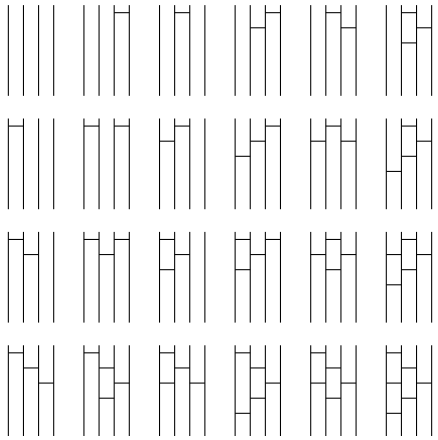


**Figure 1.5:** All 24 ladders listed in lexicographic order.

## 1.1 Thesis Statement

This thesis considers a combinatorial generation problem related to ladder lotteries. The 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 one optimal ladder per permutation? In other words, is there an easy way to transition from one ladder to the next until all $n!$ ladders have been listed? In this thesis, 'N' of the aforementioned permutation listing algorithms are modified in order to list $n!$ ladder lotteries, each of which corresponds to one of the $n!$ permutations of order $n$. The first of these algorithms is a modification of the Steinhaus-Johnson-Trotter algorithm; SJT for short [13].

## 1.2 Contributions

For the Canonical Ladder Listing Problem, this thesis provides a number of algorithms for listing all $n!$ ladders. These algorithms are termed Algorithm 4 which can be found in Chapter 3, ModifiedSJT which can be found in Chapter 3, Algorithm 5, and CyclicBar which can be found in Chapter 3 Algorithm 3. This thesis also provides a number of theorems and lemmas relating to The Canonical Ladder Listing Problem. Furthermore, we define the *canonical representation of the root ladder* which can be found in Chapter 3.

## 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 [34]. Since this paper, a number of problems related to ladder lotteries have been solved. In Table 1.1 the reader will find a table of solved problems related to ladder lotteries. In Chapter

2 a more comprehensive analysis of these solved problems will be provided.

| Table of Known Results Related to Ladder Lotteries | | |
|---|---|---|
| Name of Problem | Description | Source |
| Enumeration Problem | Generates $OptL\{\pi\}$ | [34] 2010 |
| Ladder Lottery Realization Problem | Determines time complexity for creating a ladder lottery given a multi-set of bars | [31] 2018 |
| The Reconfiguration Problem | Determine the length of the path between two ladders in $OptL\{\pi\}$ | [30] 2017 |
| Enumeration Problem given $n$, $k$ | Generates all ladders with $n$ lines and $k$ bars; includes non-optimal ladders | [32] 2014 |
| The Coding Problem | Provides a binary string encoding for a ladder lottery | [29] 2012 |
| Counting and Random Generation Problem | Provides a solution for counting ladder lotteries of a given type as well as randomly generating a ladder lottery | [33] 2017 |

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

## 1.4 Overview of Thesis

This thesis is broken down into several sections. In Chapter 2, a literature review of ladder lotteries will be provided along with background information pertinent to this thesis. In Chapter 3, The canonical configuration of the root ladder is provided. Chapter 3 is broken down into four subsections. The first subsection is the introduction to the problem, the second is a methodology subsection, the third is a results subsection and the fourth is a conclusion/analysis subsection. In Chapter 4, The ModifiedSJT algorithm is provided. Chapter 4 is broken down into four

subsections. The first subsection is the introduction to the algorithm, the second is a methodology subsection, the third is a results subsection and the fourth is a conclusion/analysis subsection. In Chapter 5, The K-Bar Generation algorithms are provided. The first subsection is the introduction to the algorithms, the second is a methodology subsection, the third is a results subsection and the fourth is a conclusion/analysis subsection.

# Chapter 2

## Background and Literature Review

In Chapter 2 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 Chapter 3 and Chapter 4. We have already defined $\pi$ in the introduction and we have already discussed a ladder lottery as a concept. Let $n$ be the number of elements in $\pi$. 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 $\pi$ is in ascending order. The number of rows in ladder is $\binom{n}{2}$ if $\pi$ 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 auxiliary 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 [34]. In this paper the authors present the first algorithm for generating $OptL\{\pi\}$ for some arbitrary permutation $\pi$. 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 $\pi$, in $\mathcal{O}(1)$ per ladder [34]. The authors refer to this algorithm as FindAllChildren which can be found in Algorithm 1.

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

```
1: function FindAllChildren(ladder, cleanLevel, n)
2:     currentRoute ← n
3:     while currentRoute ≥ cleanLevel do
4:         going top left to bottom right
5:         for bar ∈ currentRoute do
6:             row ← row of bar in ladder
7:             col ← col of bar in ladder
8:             lowerNeighbor ← 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 ← currentRoute − 1
16:    end while
17:    currentRoute ← cleanLevel − 1
18:    for bar ∈ currentRoute do
19:        row ← row of bar in ladder
20:        col ← col of bar in ladder
21:        lowerNeighbor ← 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
```

One will note that two helper functions, RightSwap and LeftSwap, are required to complete FindAllChildren. The details of these algorithms are not found in the paper [34]. Thus, these algorithms and their details can be found in the Appendix in Algorithm 13 and Algorithm 14. We define a *local swap operation* as either a right swap or a left swap. The details of which are found in the Appendix. FindAllChildren is the first algorithm for generating $OptL\{\pi\}$.
FindAllChildren enumerates $OptL\{\pi\}$ as a tree of ladders.W

FindAllChildren was used in this thesis to generate the sample data that aided in finding solutions for the Gray Code Problem in Chapter 3 and the Minimum Height Problem in Chapter 4. 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 2.1. Each $OptL\{\pi\}$ has a unique ladder

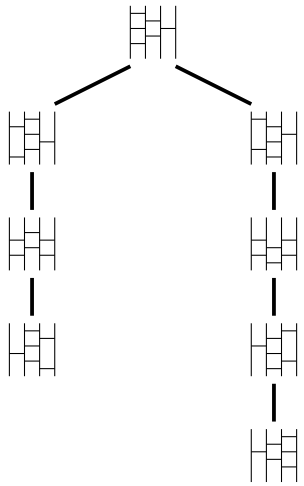


**Figure 2.1:** The tree structure of $OptL\{(4,3,2,1)\}$ generated by FindAllChildren

which is the root of the tree generated by FindAllChildren. This ladder is known as the *root ladder*. Unlike every other ladder in the tree, the root ladder cannot be derived from a local swap operation. Thus, the root ladder must be created by another algorithm other than FindAllChildren. In Chapter 3, the algorithm for creating the root ladder along with further information regarding the root ladder is provided. To see the root ladder of $OptL\{(4,5,6,3,1,2)\}$ please refer to figure Figure 2.2. In Chapter 3, section titled The Root Ladder in Detail, we discuss the details of the root ladder in relation to the Canonical Ladder Listing Problem.

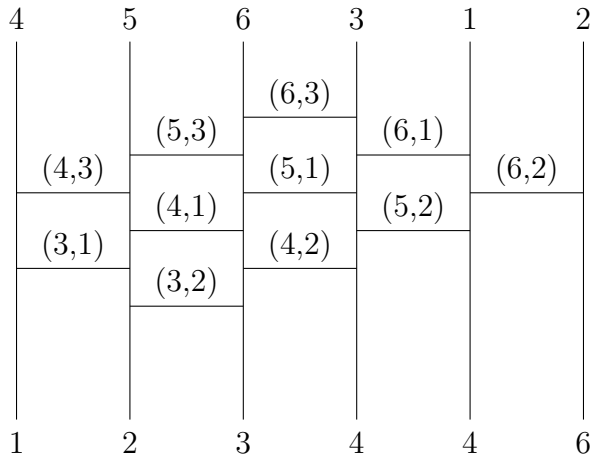



**Figure 2.2:** The root ladder for $OptL\{(4,5,6,3,1,2)\}$. Notice how no bar associated with a lesser element is above bars associated with a greater element. Therefore, the clean level of this ladder is 1, thus making it the root ladder.

In concluding the section on the enumeration problem, we have analyzed the original paper along with making additions to the algorithm FindAllChildren by providing four essential algorithms for the completion of FindAllChildren. The algorithms are Algorithm 4, Algorithm 13, Algorithm 14 and Algorithm 15 which can be found in the Appendix. In concert, these five algorithms solve the enumeration problem which lists $OptL\{\pi\}$ in $O(1)$ time per ladder.

## 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 [31]. The puzzle is known as the ladder lottery realization problem [31]. 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, $\pi$, and a multi-set of bars, is there a *non-optimal* ladder lottery for $\pi$ that uses every bar in the multi-set the number of times it appears in the multi-set [31]. For an example of an affirmative solution to the ladder lottery realization problem, see Figure 2.3.

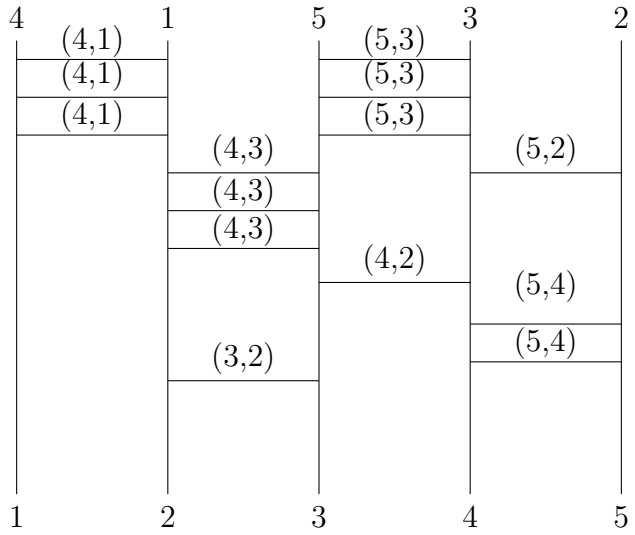



**Figure 2.3:** An affirmative solution to the Ladder Lottery Realization Problem given a starting permutation $(4, 1, 5, 3, 2)$ and the multi set of bars $\{(4, 1)^3, (4, 3)^3, (4, 2)^1, (5, 4)^2, (5, 3)^3, (5, 2)^1, (3, 2)^1\}$

The authors prove that the ladder lottery realization problem in NP-Hard by reducing the ladder lottery realization to the One-In-Three 3SAT problem, which has already been proven to be NP-Hard [31]. 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 [27]. 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 [31]. 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 [31].

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 [31]. 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 [31]. This is because the elements that cross the bar will be uninverted when then be inverted again. Therefore $\pi$ 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 reconfig-*

*uration 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 $L_m$ [30]? 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* [30]. 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 $L_m$ [30]. The second of the concepts is the *improving triple* [30]. 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 $L_m$ [30]. 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 $L_m$ [30].

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 $L_i$ [30]. 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 $L_m$ [30].

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 $n$ [30]. That is to say, given some arbitrary $\pi$ 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 [30]. 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 performed. The length of the reconfiguration sequence between the root ladder and terminating ladder in $OptL\{\pi = (n, n-1, \ldots, 1)\}$ is $n\binom{(n-1)}{2}$ [30]. 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 [32].

## 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 [29]. Coding discrete objects as binary strings is an appealing theme because it allows for compact representation of them for a computer [29].

### 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 $RC(l)$ be the route encoding

for some arbitrary ladder in $OptL\{\pi\}$. For an example of the route encoding for the root ladder of $(3, 2, 5, 4, 1)$ refer to Figure 2.4. The number of bits needed for $RC(l)$ belongs to $\mathcal{O}(n^2)$.

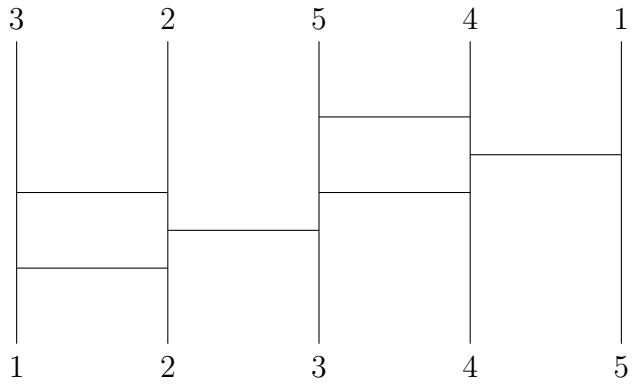



**Figure 2.4:** The route encoding for the following ladder lottery is 11<u>00</u>01<u>00</u>11<u>00</u>01<u>00</u>0000

### 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 $i$ [29]. 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 [29]. 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 [29]. Encoding and decoding can be done in $\mathcal{O}(n + b)$ time [29]. 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 [29]. 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 $LC(i = n - 1)$ [29]. The second improvement 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 $r_y$. 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 $i + 1$ [29]. If $l_x$ and $l_y$ on line $i$ have no $r_z$ between them, then there must be at least one $l_{z\prime}$ 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 $i$ [29]. 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. Improvement three is based off of saving some bits for right end points/$0s$ in $LC(i = n - 1)$. Since line $n$ has no left end points, then there must be some right endpoints between any two

consecutive bars connecting lines $n-1$ and line $n$. Knowing this, then given two bars, $x$ and $y$ with $l_x/l_y$ on line $n-1$ and $r_x/r_y$ on line $n$, there must be at least one bar, $z$, with its $r_z$ between $l_x$ and $l_y$ on line $n-1$. Thus, for every 1 in $LC(i = n-1)$ except the last 1 a 0 must immediately proceed any 1. Since this 0 is implied, it can be removed from $LC(i = n-1)$ [29]. 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 11$\underline{0}$10101$\underline{0}$00101010$\underline{0}$000. By applying improvement one, we get 11$\underline{0}$1010110$\underline{0}$0010101$\underline{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 11$\underline{0}$100110$\underline{0}$010010. Notice how the second, and eighth 1 were removed because they are implied by the successive $0s$. By applying improvement three to the result of improvement two we get 11$\underline{0}$100110$\underline{0}$01010. Notice how the last 0 was removed from improvement two. This is because the 0 is implied in $LC(i = n-1)$ due to the configuration between of bars connecting lines $n-1$ and line $n$. The $IC(l(4,2,3,1))$ for the ladder in Figure 2.5 is $IC(l(4,2,3,1)) = 11\underline{0}100110\underline{0}01010$.

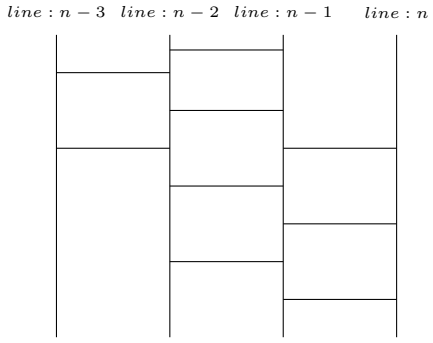



**Figure 2.5:** A ladder used to illustrate all three improvements $IC(l)$.
$IC(l) = 11\underline{0}100110\underline{0}01010\underline{0}$

## 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 [33]. 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 $S_{n,b}$ [33]. 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 [33].

### 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 [33]. 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.[33]

## 2.7 Permutations

Ladder lotteries and permutations are intricately related to each other. Each $\pi$ has an $OptL\{\pi\}$ such that each ladder form $OptL\{\pi\}$ sorts $\pi$. 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 [16]. Since this book, many algorithms for enumerating $n!$ permutations have been created. During the research for the Listing Problem, twelve of these enumeration algorithms were investigated [16, 3, 6, 12, 13, 8, 35, 28, 20, 1, 15].

The first is the lexicographic algorithm which orders all $n!$ permutations from smallest to largest. The lexicographic algorithm generates each permutation with an amortized time of $O(n^2log(n))$ [6]. The second of which is the colexicographic algorithm which orders all $n!$ permutations from largest to smallest. The colexicographic algorithm generates each permutation with an amortized time of $O(n^2log(n))$ [3]. The third of which is Zak's algorithm which generates all $n!$ permutations by reversing a suffix in one permutation to get to the next permutation. The time complexity of Zak's algorithm is amortized time of $O(n^2)$ [35]. The fourth of which is Heap's algorithm which is a decrease and conquer algorithm. The time complexity of Heap's algorithm is CAT, constant amortized time of $O(1)$ per permutation [12]. The fifth algorithm is the Steinhaus-Johnson-Trotter algorithm which generates $S_n$ by performing adjacent swap operations on the permutation resulting in the next permutation. Thus, each permutation in $S_n$ differs from it predecessor by a single swap operation [13]. To go from any two successive permutations, the time complex-

ity is $O(1)$ [13]. The sixth algorithm is the algorithm using star transpositions that always swaps the first element of the permutation with some other element. It was discovered by Gideon Ehrlich and is described as Algorithm E in Knuth's book [16]. The seventh algorithm is the derangement ordering in which no two consecutive permutations have any elements in the same position. It was first discussed by Savage in [20]. The eighth algorithm is the Single Track listing algorithm. Each column in the list of permutations is a cyclic shift of the first column [1]. The computation for each successive permutation is CAT [1]. The ninth algorithm is the Single Track Gray Code listing algorithm. The properties of the Single Track listing algorithm hold. Furthermore, any two successive permutations differing by at most two transpositions [1]. The tenth listing algorithm is found in Knuth's book. At each step, it either rotates the permutation to the left by one or swaps the first two elements [15]. The problem as to whether such a listing algorithm exists for all $n$ is posed as an open problem in Knuth's book [15]. It was solved by Sawada and Williams in their paper A Hamiltonian Path for the Sigma-Tau Problem [21]. The eleventh algorithm is Corbett's algorithm which rotates a prefix of the first possible length in $n$,2,$n-1$,3,$n-2$,4, etc. [28]. The twelfth algorithm is Effler and Ruskey's algorithm which lists permutations by groups of $k$ inversions. constant amortized time algorithm for generating permutations of order $n$ with $k$ inversions [8] meaning To see the listings for all the aforementioned algorithms other than the Effler-Ruskey algorithm, refer to 2.6.

In Chapter 1, it was stated that Algorithms [13][8] were the most conducive for The Canonical Ladder Listing Problem. These are the Steinhaus-Johnson-Trotter and Effler-Ruskey algorithms respectively. The reason that the SJT algorithm is beneficial is because it generates each permutation in $O(1)$ per ladder. In Chapter 3, the SJT algorithm is modified to create ladders instead of permutations while still maintaining the same efficiency. The reason that the Effler-Ruskey algorithm is beneficial is because it has a nice ordering property such that the permutations are listed by the number of inversions. In Chapter 3, we order all $n!$ ladders by

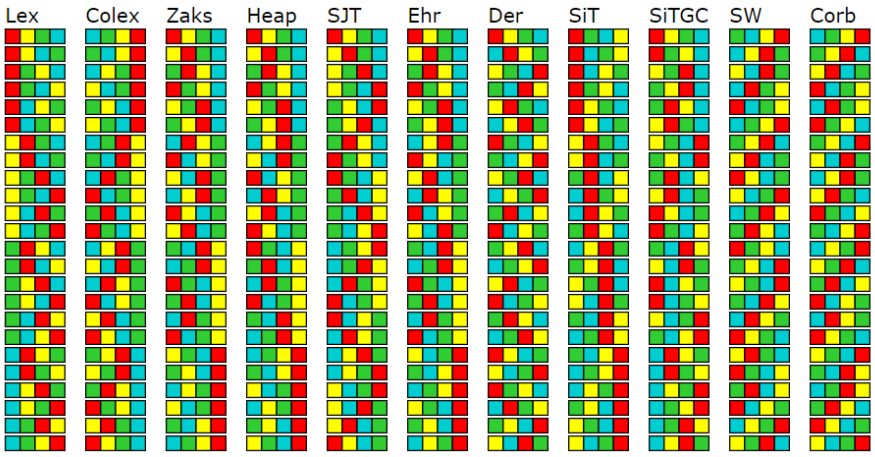



**Figure 2.6:** Eleven permutation listing algorithms

$0, 1, 2, \ldots, \binom{n}{2}$ bars where each bar corresponds to an inversion in a permutation.

### 2.7.1 Steinhaus-Johnson-Trotter Algorithm

The Steinhaus-Johnson-Trotter algorithm generates $S_n$ by performing adjacent swap operations on the permutation resulting in the next permutation. Thus, each permutation in $S_n$ differs from it predecessor by a single swap operation [13]. 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 $\pi$ of order $n-1$ in descending order if $\pi$ of order $n-1$ is an even permutation. The *nth* element is inserted into all positions of $\pi$ of order $n-1$ in ascending order if $\pi$ of order $n-1$ is an odd permutation [13]. For $\pi$ 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 $(2, 1, 3)$. Initialize $\pi$ to the identity permutation, initialize *currentElement* to 2, initialize $n$

to $p_{max}$. Let *direction* be a Boolean array of size $n$ initialized to *true*/*right* for indices. If *currentElement* is greater than $n$, print $\pi$ and return. Otherwise, begin a for loop that runs $i = [1 \dots n-1]$ times. In the for loop, first make a recursive call with *currentElement* increasing by one. If *direction*[*currentElement*] is *right*, then swap *currentElement* in $\pi$ with its left neighbor. Else if *direction*[*currentElement*] is $left$ then swap the *currentElement* in $\pi$ 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* in *direction* for the next time *currentElement* is to be swapped. To see the Steinhaus-Johnson-Trotter algorithm please refer to Algorithm 2.

**Algorithm 2** SJT Algorithm for listing $S_n$

```
1:  function SJT(π, currentElement, n, direction)
2:      if currentElement > n then
3:          if SORTED(π) then
4:              PRINT(π)
5:          end if
6:          return
7:      end if
8:      for i from 1 to currentElement − 1 do
9:          SJT(π, currentElement + 1, n)
10:         index ← index of currentElement in π
11:         if direction[currentElement] = right then
12:             SWAP(p_index, p_index−1)
13:         else
14:             SWAP(p_index, p_index+1)
15:         end if
16:         PRINT(π)
17:     end for
18:     SJT(π, currentElement + 1, n, direction)
19:     if direction[currentElement] = right then
20:         direction[currentElement] ← left
21:     else
22:         direction[currentElement] ← right
23:     end if
24: end function
```

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

### 2.7.2 Effler-Ruskey Algorithm

In the paper A CAT Algorithm for Generating Permutations with a Fixed Number of Inversions, written by Effler and Ruskey, the authors provide a constant amortized time algorithm for generating all permutations of order $n$ with $k$ inversions [8] meaning the time it takes to generate each permutation is bounded by a constant. 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 [8]. Let the initial conditions of the algorithm be the following. Let $\pi$ be the empty permutation of order $n$. Let $n$ be initialized as the elements $[1 \dots n]$. Let $k$ be initialized to an integer between $[0 \dots \binom{n}{2}]$. Let $list$ be initialized to the list of $n$ elements in strictly descending order. Let $i$ be the index of element $x$ in $list$. $n - i$ indicates the number of inversions formed with $x$ when inserting $x$ into position $n$ in $\pi$. The algorithm works as follows. Going right to left in the $list$, each element, $x$, in $list$ is inserted into $p_n$ only if inserting $x$ into position $n$ in $\pi$ does not exceed the current number of inversions, $k$. I.E. only if $k - (n - i) \geq 0$. If $x$ is inserted into $\pi$ at position $n$ then $x$ is removed from $list$, a recursive call is made such that $n$ is reduced by 1 and $k$ is reduced by $(n - i)$. When the function returns from its recursive call, element $x$ is placed back into the list at its original position. To see the algorithm please refer to Algorithm 3.

**Algorithm 3** Generate all permutations with $k$ inversions

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

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

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

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

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

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

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

## 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 2.7.

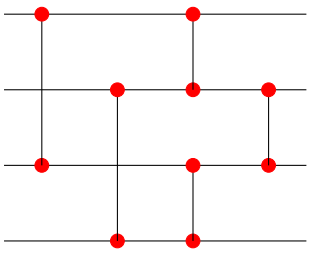


**Figure 2.7:** Complete Sorting Network for $n = 4$.

Sorting networks were first studied in 1954 by Armstrong, Nelson and O'Connor. Sorting networks can be implemented either in hardware or in software [16]. Donald Knuth describes how the comparators for binary integers can be implemented as

simple, three-state electronic devices [16]. Batcher, in 1968, suggested using them to construct switching networks for computer hardware, replacing both buses and the faster, but more expensive, crossbar switches [2]. 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 [19].

Sorting networks are intricately related to ladder-lotteries. Let a *minimum sorting network* be defined as a sorting network such that for any arbitrary comparator, $c$, on wire $i$, $c$ connects to line $i+1$ or $i-1$. Furthermore, the number of comparators in a minimum sorting network is equal to the number of inversions in $\pi$. 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 [14].

### 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 2.1 [23].

According to Dumitrescu and Mandal, in their paper New Lower Bounds For The Number of Pseudoline Arrangements published in 2018, they have devised the current best known lower bound for this sequence to be $bn \geq cn^2 - O(nlogn)$ for some constant $c > 0.2083$. In particular, $bn \geq 0.2083n^2$ for large values of $n$ [7]. Where $bn$ is the bound for a given value $n$. In the paper, Coding and Counting Arrangements of Pseudolines by Felsner and Valtr, written in 2011, the authors demonstrate the best known upper bound for this sequence is $bn \leq 2^{0.657n^2}$ [9].

Seeing as there is yet to be a closed form solution for this sequence, new values

| 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 $\lvert OptL\{Rev(\pi)\}\rvert$

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 [34]. In the paper, Counting Primitive Sorting Networks by $\pi$DDs, written by Kawahara, Minato, Saitoh and Yoshinaka, the authors were the first to calculate for $n = 13$ with a data structure they have termed $\pi$DD [14]. 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 [14]. The data structure resembles a digraph with two sink nodes; one sink node is labelled the zero sink node and the other is labelled the one sink node [14].

# Chapter 3

## The Canonical Ladder Listing 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*. This problem asks, given all $n!$ permutations of order $n$, is there an efficient algorithm for listing one optimal ladder per permutation? In other words, is there an easy way to transition from one ladder to the next until all $n!$ ladders have been listed? The problem is stated as follows: Let $S_n$ be the set of all $n!$ permutations of order $n$. Let the *canonical representative* be the root ladder from each permutation's corresponding $OptL\{\pi\}$. Let $L_n$ be the set of all $n!$ root ladders. Refer to Figure 3.1 for the root ladder for $(3, 5, 4, 1, 2)$. A *change* is defined as the insertion or deletion

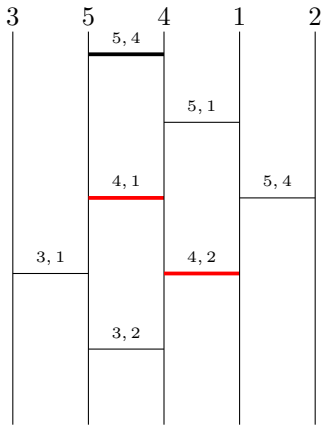



**Figure 3.1:** The root ladder for $(3, 5, 4, 1, 2)$

of at least one bar from a ladder in $L_n$. The goal is to list $L_n$ by changing $l_i \in L_n$ to get to $l_{i+1} \in L_n$.

In this thesis, two listing algorithms are used to list $L_n$. The first of these listing algorithms is a modification of the STEINHAUS-JOHNSON-TROTTER permutation listing algorithm [13]. This algorithm is termed MODIFIEDSJT. The second listing algorithm is influenced by Effler and Ruskey's algorithm [8]. This second ladder listing algorithm is termed the CYCLICBAR algorithm. Both of these algorithms will be described, explained and analyzed throughout the remainder of the chapter.

## 3.1 The Root Ladder in Detail

In order to understand the root ladder, there are a number of key concepts and definitions that first need to be discussed. Let the *route* of an element be the sequence of bars the element travels along in order to reach its final position in the sorted permutation. The bars are read from top to bottom. For every bar, two elements cross the bar, therefore *bar association* is the association of a bar with the greater of the two elements that cross it. Let *route association* be the sequence of bars associated with element $x \in \pi$. The bars of the route of element 4 in $(3, 5, 4, 1, 2)$ are $(5, 4), (4, 1), (4, 2)$. However, $(5, 4)$ makes up the unassociated part of the route of 4 whereas $(4, 1), (4, 2)$ make up the associated part of the route of 4. We write a bar as $(a, b)$ in a ladder $l$, where $a > b$. Let $R((a, b))$ be the row of bar $(a, b) \in l$. Suppose we have two bars, $(a, b)$ and $(c, d)$. $(a, b)$ is said to be *above* $(c, d)$ if $R((a, b)) < R((c, d))$. Divide the route of $x$ into the associated and unassociated parts of its route. Let $w > x > y > z$ be any four elements in $\pi$ and let $p_w < p_x < p_y < p_z$. We know that $(w, x)$ is part of the unassociated part of $x's$ route and $(x, y), (x, z)$ are part of the associated part of $x's$ route. We say the *clean level* of $l$ is the maximum value of $x$ which the following properpty holds :$\forall(y, z)(w, y) \in l : R(y, z) > R(x, z)^R(w, y) < R(x, y)$. To see a ladder with a clean level of 3, refer to Figure 3.2. A unique property of the root ladder is that there is no element $x \in \pi$ such that an associated bar in its route is above an unassociated bar in its route. Therefore, we say the root

ladder has a clean level of 1. The root ladder is the only ladder in $OptL\{\pi\}$ with a clean level of 1. Given the route associated with $x$, each bar in $x's$ route requires its

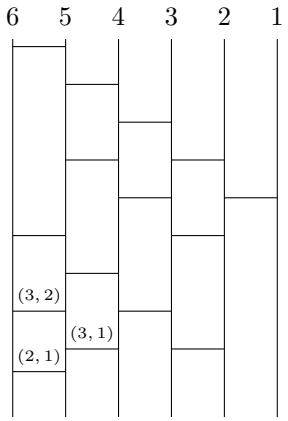



**Figure 3.2:** A ladder with a clean level of 3. All bars of the form $(a, 3)$ where $a > 3 > c$ are above $(3, c)$ and all of the bars of the form $(c, d)$ where $3 > c > d$ are below $(3, d)$ 3 is the maximum element in $\pi$ from $1 \ldots n-1$ where the above property holds.

own row in the root ladder. In other words, no two bars in the route of $x$ share the same row. If $(x, y)$ exists and $(x, z)$ exists then $(x, y)$ is to the left of $(x, z)$ if and only if $R((x, y)) < R((x, z))$. However, given $x$ and $y$ where $x \neq y$, a bar associated with $x$ can share a row with a bar associated with $y$ in the root ladder. Let $l_i$ be line $i$ in the ladder. The *direction* of an element refers to an element crossing a bar connecting $l_i$ to $l_{i-1}$ or $l_{i+1}$. If an element crosses a bar going from from $l_i$ to $l_{i-1}$ then said element has a right-to-left direction. If an element crosses a bar from $l_i$ to $l_{i+1}$ then said element has a left-to-right direction. A *switch in direction* refers to an elements direction switching from left-to-right or from right-to-left as said element travels along its path. If the maximum number of direction switches for any element is $> 1$ then the ladder is not the root ladder of $OptL\{\pi\}$. In 3.1, elements 1 and 2 have a right-to-left direction, element 3 has a left-to-right direction, element 4 starts with a right-to-left direction and then switches to a left-to-right direction, and element 5 has a left-to-right direction. In the root ladder, as $x$ travels along the associated component of its route, the direction must be left-to-right. If $x$ travels along the unassociated component of its route, the direction must be right-to-left.

Although unique, the root ladder can be drawn in many different ways due to the fact of ladder equivalency and ladder identity which were discussed in Chapter 1. For example, if one refers to Figure 3.3, one can see the root ladder for $(6, 1, 3, 2, 4, 5)$ drawn in three different ways. Let the *canonical configuration of the root ladder* be the representation of the root ladder with a height of $2(n-1)-1$. In Figure 3.3, the leftmost ladder is the canonical configuration of the root ladder because it has a height of $9 = 2(6-1)-1$. The reason $2(n-1)-1$ is chosen as the height of the canonical configuration of the root ladder is because any root ladder of order $n$ requires at most $2(n-1)-1$ rows by Theorem 3.1.1 and Corollary 3.1.2. Therefore, when $2(n-1)-1$ is used as the height of the ladder, the two dimensional array used for solving the Canonical Ladder Listing Problem has static dimensions of $2(n-1)-1$ rows and $(n-1)$ columns.

**Theorem 3.1.1** *The number of rows required for the root ladder corresponding to the descending permutation of order $n$ is $2(n-1)-1$.*

*Proof.* Let $\pi$ be the descending permutation of order $n$. Let $Route(x)$ be the route associated with element $x$ in the corresponding root ladder of $\pi$. Let $R(x)$ be the row of the uppermost, leftmost bar of the route associated with $x \in \pi$. Let $R\prime(x)$ be the row of the bottommost, rightmost bar of the route associated with $x \in \pi$. Let $C(x)$ be the column of the uppermost, leftmost bar of the route associated with $x \in \pi$. Let $C\prime(x)$ be the column of the bottommost, rightmost bar of the route associated $x \in \pi$. For each $x \in \pi$, the number of inversions formed with $x$ such that $x$ is the greater of the two elements in the inversion is $x-1$. Thus, in the corresponding root ladder associated with $\pi$, $Route(x)$ has $x-1$ bars. Furthermore, no bar in $Route(x)$ shares a row with any other bar in $Route(x)$ in the root ladder by the fourth property. Therefore, $Route(x)$ requires $x-1$ rows. In the root ladder, $C(x) = 1$ for each $x$ in $\pi$. This is true due to the fifth property, along with the fact that the number of bars in $Route(x)$ is $x-1$. Since $Route(x) = x-1$, $C(x) = 1$ and $x-1$ rows are

required for $Route(x)$, then $C'(x) = x - 1$. If $x = n$ then $R(x) = 1$, $C(x) = 1$, $R\prime(x) = n - 1$, and $C\prime(x) = n - 1$. If $1 < x < n$ then $R(x) = 2 + R(x + 1)$, $C(x) = 1$, $R\prime(x) = 1 + R\prime(x + 1)$ and $C\prime(x) = x - 1$. There are $n - 2$ elements between 1 and $n$. For all $1 < x < n$, $R\prime(x)$ is located one row below $R\prime(x + 1)$, therefore one additional row needs to be added to the ladder to accommodate the bottommost bar of the route associated with $x$. Thus, $n - 2$ rows need to be added to the ladder for each element $x$. The $nth$ element requires $n - 1$ rows for its route. Thus, there are $(n - 1) + (n - 2) = 2(n - 1) - 1$ rows required for the root ladder corresponding to the descending permutation. □

**Corollary 3.1.2** *Let ladder be the data structure for the root ladder for any permutation of order n. Let r be the number of rows required for ladder. Let c be the number of columns required for ladder.* $0 \leq r \leq 2(n - 1) - 1$ *and* $c = (n - 1)$.

*Proof.* By Theorem 3.1.1, we know that $r = 2(n - 1) - 1$ for the root ladder corresponding to the descending permutation of order $n$. The number of bars required for said ladder is $\binom{n}{2}$. Let $b$ be the number of bars required for any root ladder of order $n$. We know that when $\binom{b=n}{2}$, $r = 2(n - 1) - 1$. By removing bars from the root ladder for the descending permutation of order $n$, we can derive any other root ladder for any other permutation of order $n$. Removing bars from the root ladder corresponding to the descending permutation of order $n$ does not necessarily remove a row from *ladder*, however, removing bars from *ladder* certainly does not add any more rows to *ladder*. Therefore, $0 \leq r \leq 2(n - 1) - 1$. $c = (n - 1)$ seeing as there are $n$ lines for any ladder of order $n$ and a column is defined by a pair of lines. □

**Lemma 3.1.3** *If x has bars associated with its route, then the bottommost bar of the route of x is associated with the route of x in the root ladder.*

*Proof.* We shall do a proof by contradiction. Let $w > x > y \in \pi$. Let $p_w < p_x <_p y$. Let $(w, x)$ be the bottommost bar of $x's$ route. We know $x$ crosses $(w, x)$ from

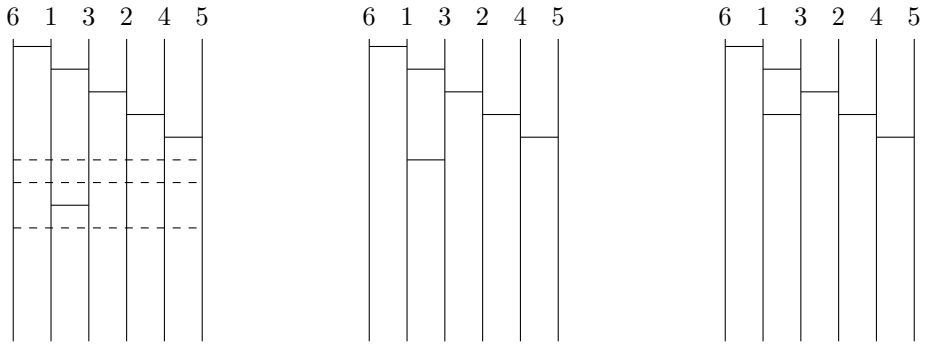



**Figure 3.3:** All three root ladders are equivalent. The left and middle ladder are identical, with 6 rows each. The height of the left ladder is 9, the height of the middle ladder is 6, and the height of the right ladder is 5.

the right-to-left. $x$ has route association, therefore $(x, y)$ is a bar and we know $x$ crosses $(x, y)$ from left-to-right. Since $(w, x)$ and $(x, y)$ exist, so does $(w, y)$. From our assumption, we assumed $(w, x)$ was the bottommost bar of $x's$ route. Therefore, $R((x, y)) < R((w, x))$. $R((w, y)) < R((x, y))$ due the root ladder having a clean level of 1. Therefore, $R((w, y)) < R((w, x))$. Yet, $p_w < p_x < p_y$ Therefore, $R((w, x)) < R((w, y))$. Contradiction. Therefore, the bottommost bar of the route of $x$ must be associated with $x$.

□

**Corollary 3.1.4** *If $x$ has bars associated with its route, then the column of the bottommost bar of the route of $x$ is column $x - 1$.*

*Proof.* Let $b$ be the bottommost bar of the route of $x$. From Lemma 3.1.3 we know that $b$ is associated with $x$. Therefore, $x$ crosses $b$ from left-to-right. We also know that $x$ must end up at the bottom of line $x$. Let $c$ be the $x - 1$ column in the root ladder. Let $l_x$ be line $x$ in the ladder. Conceptually, $c$ is the gap between $l_{x-1}$ and $l_x$. Since $x$ crosses $b$ from left to right and $x$ must end at line $x$, $b$ must be located at column $x - 1$. □

We use Corollary 3.1.2, Corollary 3.1.4, and the properties of the root ladder to derive a naive algorithm for creating the canonical configuration of any root ladder

corresponding to any permutation of order $n$. In order to view said algorithm, please refer to Algorithm 4. The initial conditions of Algorithm 4 are the following. Let *ladder* be initialized to an empty ladder. Let $\pi$ be an arbitrary permutation of order $n$. Let $x$ be the current element in $\pi$ initialized to $n$.

**Algorithm 4** The algorithm for creating the root ladder of $OptL\{\pi\}$

```
1:  function CreateRoot(ladder, π, x)
2:      if x = 1 then
3:          return
4:      end if
5:      index ← p_x
6:      numBars ← x − index
7:      col ← x − 1
8:      row ← (n − 1) + (n − x)
9:      for i ← 1 to numBars do
10:         ladder[row][col] ← 1
11:         row ← row − 1
12:         col ← col − 1
13:     end for
14:     π ← π − x
15:     CreateRoot(ladder, π, n, x − 1)
16: end function
```

Bars associated with the route of the $xth$ element are added, starting from the bottommost bar to the topmost bar. Once all the bars for the associated route of the $xth$ element have been added, the function makes a recursive call. Just prior to the recursive call, element $x$ is removed from $\pi$; all the elements to the left of element $x$ in $\pi$ maintain their same index, and all the elements to the right of element $x$ in $\pi$ are moved one index to the left in $\pi$. On the recursive call, $x$ is decremented by 1; this ensures that $x$ is always the maximal element in $\pi$. The local variable $numBars$ represents the number of bars associated with the route of $x$. On each recursive call $x$ is always the largest element in $\pi$, thus the number of inversions associated with $x$, such that $x$ is the larger element of said inversion, equals $x$ minus the position of element $x$ in $\pi$. Therefore, the number of bars associated with the route of $x$ equals $x - p_x$.

The local variable *col* represents the column to place the bottommost bar associated with the route of $x$. By Corollary 3.1.4, it has been shown that $col = x - 1$. The local variable *row* represents the row to place the bottommost bar associated with the route of $x$. When $x = n$, the maximum number of bars associated with $x$ for any permutation of order $n$ is $n - 1$. Therefore, the bottommost bar of route $x$, when $x = n$, is by default placed at row $n-1$. Once *row* is initialized to $(n-1)$, *row* will be incremented by an offset value of $(n - x)$. Conceptually, the offset increment means that the row for the bottommost bar associated with the route of $x$ is one greater than the row for the bottommost bar associated with element $x + 1$. To see the resulting state of *ladder* for each recursive call to CreateRoot with $\pi = (5, 7, 3, 4, 1, 2, 6)$ please refer to Figure 3.4

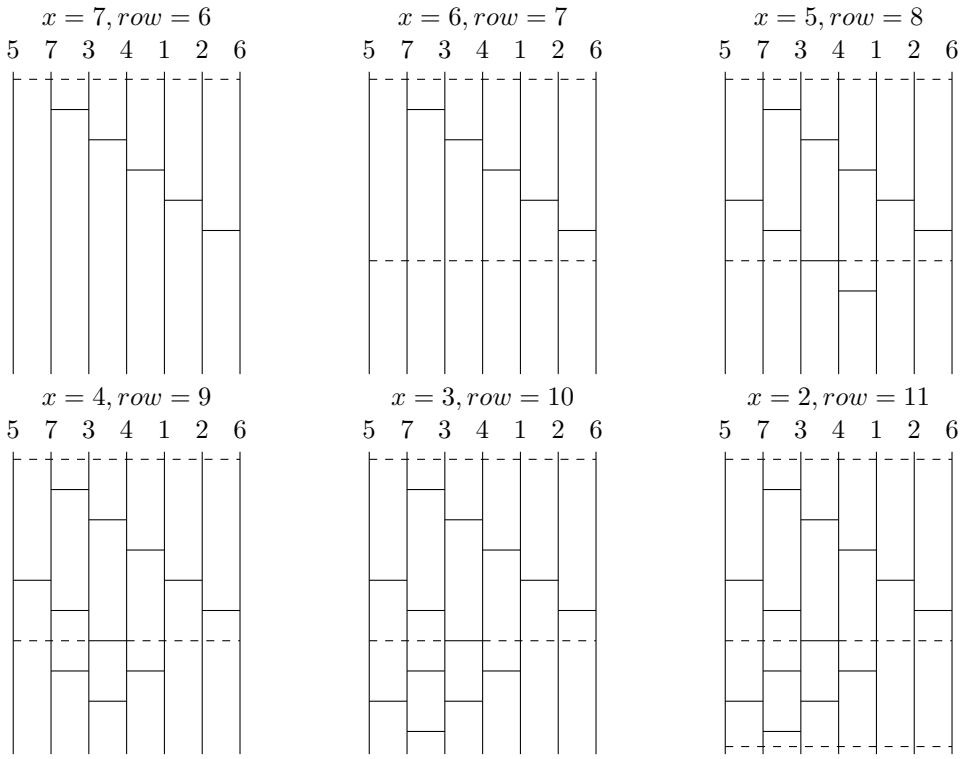



**Figure 3.4:** The ordering of the state of the ladder when creating the root ladder for $(5, 7, 3, 4, 1, 2, 6)$

**Lemma 3.1.5** *The time complexity for CreateRoot is* $O(n^2)$

*Proof.* The for-loop runs from some arbitrary index to $n$ on each function call. Thus, we get $O(n)$. The following recursion holds, $CreateRoot(n-k) = CreateRoot((n-k)+1)+n = CreateRoot((n-k)+2)+n-1\ldots$. The recurrence relation is reduced to $n(n+1)/2 = O(n^2)$. □

We could use the naive algorithm to list $L_n$, however, doing so is rather inefficient. In the proceeding sections of Chapter 3, we provide the ModifiedSJT and CyclicBar algorithms in order to list $L_n$. ModifiedSJT and CyclicBar require additional auxiliary functions. We define Inv(x) as follows: $|\forall y \in \pi : y < x, p_y > p_x|$. For example, given $(3, 4, 1, 5, 6, 2)$, Inv(1)=0, Inv(2)=0, Inv(3)=2, Inv(4)=2, Inv(5)=1, Inv(6)=1. Given Inv(x), we can define the function GetCoordinates. In order to define GetCoordinates, let $x > y \in \pi$. Given $p_x$ and $p_y$,

suppose a transposition were applied to $x$ and $y$. Prior to the transposition of $x$ and $y$, if $p_x = p_y - 1$ then GetCoordinates returns the row and column of the bar $(x, y)$ to be removed from the ladder. If $p_x = p_y + 1$, then GetCoordinates returns the row and column of bar $(x, y)$ to be added to the ladder.

## 3.2 Modified Steinhaus-Johnson-Trotter Algorithm

In order to generate $L_n$, we could use the naive algorithm for creating a root ladder which runs in $O(n^2)$ per ladder. However, doing so is rather inefficient seeing as $|L_n| = n!$. With that being said, we modify the Steinhaus-Johnson-Trotter algorithm to produce root ladders rather than permutations. One of the main results of this thesis is ModifiedSJT produces $L_n$ in Gray Code order in CAT. With the ModifiedSJT algorithm, we make a minimal amount of change between subsequent root ladders in $L_n$ in $O(1)$ time. At each step, we compute the location of the bar to where it will be added to or where it will be removed from.

For the initial conditions of ModifiedSJT assume the following. Let be the *ladder* be initialized a two dimensional binary array of only $0's$. 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 $n$ be the maximum element. Let *direction* be a one indexed array set to left for all indexes. Let left indicate that a bar is to be added to the ladder; bars are added left to right. Let right indicate a bar is to be removed from the ladder; bars are removed right to left. Let *route* be initialized to 2. On each recursive call *route* is incremented by 1 from $2, 3 \ldots n$.

**Algorithm 5** Modification of the SJT algorithm for listing $L_n$

```
1: function ModifiedSJT(n, ladder, route, direction)
2:     if route > n then
3:         if number of bars in ladder equals 0 then
4:             Print(ladder)
5:         end if
6:         return
7:     end if
8:     for i from 0 to route − 1 do
9:         if i = 0 then
10:            ModifiedSJT(n, ladder, route + 1, direction)
11:        else
12:            if direction[route] = left then
13:                row ← (n) + (n − route) − (i);
14:                col ← route − i
15:                ladder[row][col] ← 1
16:            else
17:                col ← i
18:                row ← 2(n − route) + (i)
19:                ladder[row][col] ← 0
20:            end if
21:            Print(ladder)
22:            ModifiedSJT(n, ladder, route + 1, direction)
23:        end if
24:    end for
25:    if direction[route] = left then
26:        direction[route] ← right
27:    else
28:        direction[route] ← left
29:    end if
30: end function
```

The principles of the algorithm are the following. Given $route = k$, add or remove a bar for route $k$, then add or remove all bars for $route = k+1$. Once all bars for route $k+1$ have been added or removed, proceed to add or remove the next bar from $route = k$. Repeat until all $k-1$ bars have been added or removed from route $k$. If $direction[route]$ is left, then bars of $route$ will be added to $ladder$ from right to left, bottom to top, until no more bars to $route$ can be added. If $direction[route]$ is right, then bars will be removed from $ladder$, right to left, top to bottom, until no more bars from $route$ can be removed. Once all the bars for $route$ have been added or removed, then $direction[route]$ is reversed, indicating that the opposite operation will be applied to the bars of the route when $route$ is next processed. The maximum number of bars for a given $route = k$ is $k-1$ and the minimum number of bars is 1. On each recursive call, $route$ is incremented by 1. When $route$ is greater than $n$, print the ladder and return.

**Lemma 3.2.1** ModifiedSJT *produces* $L_n$

*Proof.* Since the algorithm is a modification of the Steinhaus-Johnson-Trotter algorithm, a similar proof for the SJT algorithm can be applied to the ModifiedSJT algorithm. Suppose we want to list all $n!$ ladders of order $n$. Suppose we have all $n-1!$ ladders of order $n-1$, then for each ladder of order $n-1$ add a new column to the right; this results in $n-1$ columns seeing as the number of columns is one less than the number of elements. For each of the $n-1!$ ladders with $n-1$ columns add $0 \ldots n-1$ bars beginning at column $n-1$ and ending at column 1. Doing so results in $(n-1)!n = n!$ ladders of order $n$. To see an example of the proof please refer to Figure 3.5. □

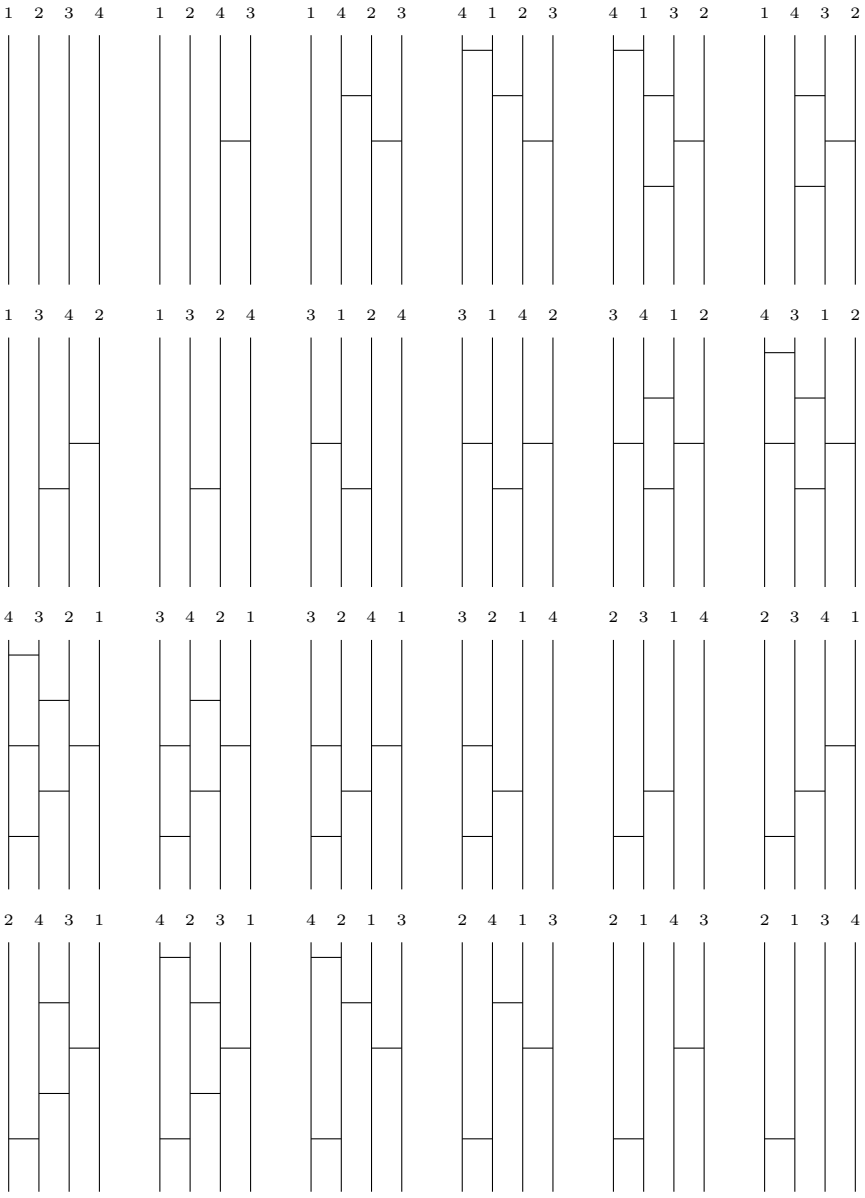



**Figure 3.5:** $L_4$ generated by the ModifiedSJT algorithm. The algorithm inserts or removes a bar between any two successive ladders.

From looking at Figure 3.5 it should be clear that the canonical representative from $L_n$ when using the ModifiedSJT algorithm is also the root ladder from each $OptL\{\pi\}$. Recall that the root ladder is the ladder with a clean level of one. In the case of the ModifiedSJT algorithm, transitioning from one ladder to the next involves simply inserting a new bar or removing a bar. 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 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 greater element. Thus, the canonical representative for $L_n$ is always the root ladder from each $OptL\{\pi\}$. The number of rows required for the ladder data structure is $2(n-1)-1$. See Theorem 3.1.1 for details.

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 3.2.2** *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 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 $L_n$ 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 3.6 for an example of row calculation when adding a bar. □

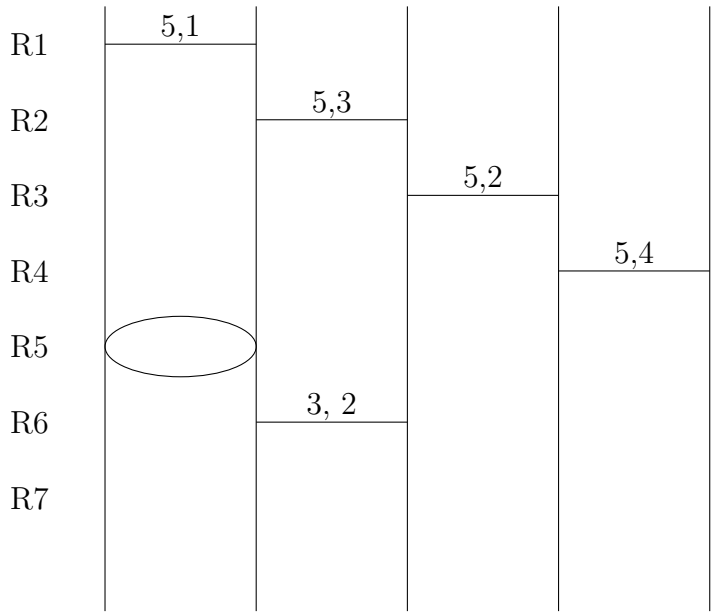



**Figure 3.6:** The second bar of route 3 goes will go in row 5, column 1. $5 = (5-1) + (5-3) - 1 = (n-1) + (n - route) - i$.

**Lemma 3.2.3** *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 3.7 for an example of column calculation when adding a bar for $k < n$. □

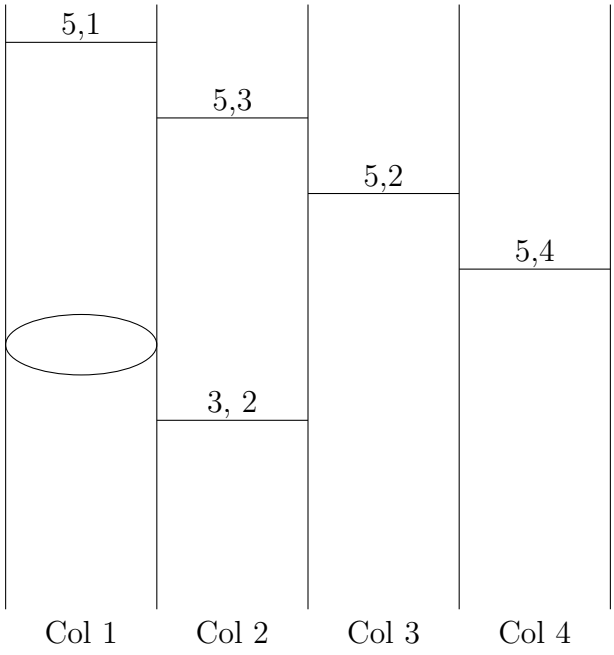



**Figure 3.7:** The second bar of route $k = 3$ goes will go in column 1. Since one bar has been added, $i = 1$. $col = 1 = 3 - 1 - 1 = route - 1 - i$.

**Lemma 3.2.4** *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* $row = 2 * (n - route) + (i) + 1$.

*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 $route = k$:

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

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 3.8 for an example of removing a bar. □

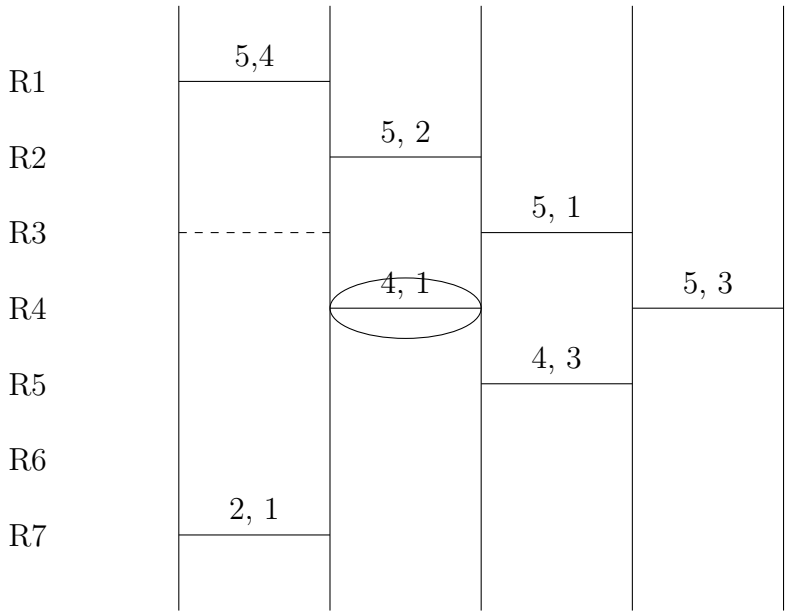



**Figure 3.8:** The bar to be removed for route $k = 4$ is (4, 1) which is at row 4. The dashed line indicates a bar from route 4 has already been removed. $row = 4 = 2(5-4)+1+1 = 2(n-route)+i+1$.

**Lemma 3.2.5** *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 3.9 for an example of column calculation when removing a bar. □

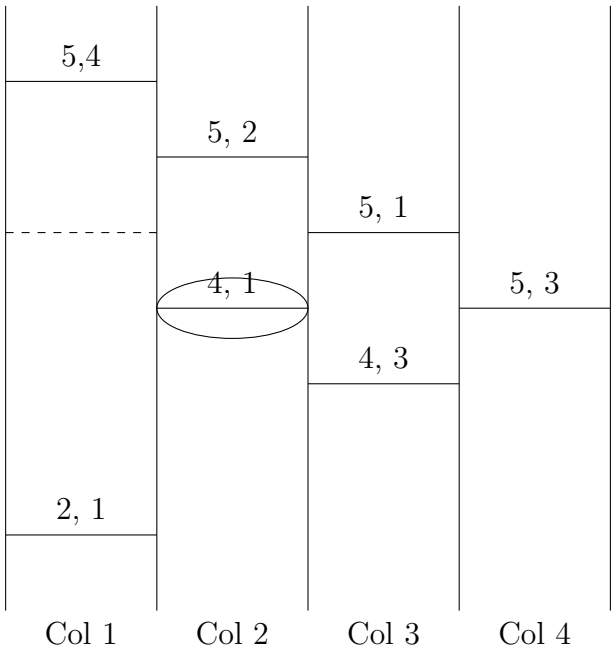



**Figure 3.9:** The bar to be removed for route $k = 4$ is (4, 1) which is at column 2. The dashed line indicates a bar from route 4 has already been removed. Since one bar from route 4 has been removed, $i = 1$. $column = 2 = 1 + 1 = i + 1$.

### 3.2.1 The Cyclic Bar Algorithm

The CyclicBar algorithm is influenced by the CAT algorithm for generating all permutations with $k$ inversions [8]. The algorithm lists $L_n$ by listing all ladders with $n$ lines and $x$ bars before listing all ladders with $n$ lines and $x + 1$ bars for $x = 0, 1, \ldots, \binom{n}{2}$. The initial conditions of CyclicBar are the following: Let be the *ladder* be initialized a two dimensional binary array of only $0's$. 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 *currentLimit* be the current number of bars to be added to *ladder*; *currentLimit* is equivalent to aforementioned $x$. Let $maxLimit = \binom{n}{2}$. Let $n$ be the number of lines in *ladder*. Let $k$ be the current route initialized to 2.

**Algorithm 6** First part of the algorithm Cyclic Bar

```
1: function CyclicBar(ladder, currentLimit, maxLimit, n, k)
2:     if k = n then
3:         m ← 0
4:         row ← k − 1
5:         col ← k − 1
6:         numBars ← current number of bars in ladder
7:         while numBars < currentLimit and m < k − 1 do
8:             ladder[row][col] ← 1
9:             row ← row − 1
10:            col ← col − 1
11:            m ← m + 1
12:            numBars ← numBars + 1
13:        end while
14:        if numBars = currentLimit then
15:            Print(ladder)
16:            remove upper leftmost bar from route k = n
17:        end if
18:        return
19:    end if
```

**Algorithm 7** Cyclic Bar Continued

```
20:    if k < n then
21:        count ← 0
22:        for i from 0 to k − 1 do
23:            if i = 0 then
24:                CyclicBar(ladder, currentLimit, maxLimit, n, k + 1)
25:            else
26:                row ← (n − 1) + (n − k) − count
27:                column ← (k − 1) − arr[k]
28:                ladder[row][column] ← 1
29:                count ← count + 1
30:                CyclicBar(ladder, currentLimit, maxLimit, n, k + 1)
31:            end if
32:        end for
33:        remove all k − 1 bars from k's route.
34:    end if
35: end function
```

**Algorithm 8** Driver for the Cyclic Bar Algorithm

```
1: function CyclicBarDriver(ladder, n)
2:     maxLimit ← (n choose 2)
3:     k ← 2
4:     for i from 0 to maxLimit do
5:         CyclicBar(ladder, i, maxLimit, n, k)
6:     end for
7: end function
```

The CyclicBar algorithm produces a tree of ladders with *currentLimit* bars in each ladder where *currentLimit* is a value between $0, 1, \ldots, \binom{n}{2}$. The parent to child relation of the tree structure is defined as follows. Let $x$ be the parent ladder of $y$ and let $y$ be the child of $x$. Let $k$ be the route of the bars to be added to $x$. Let $k-1$ be the route of the bars to be added to $y$. To go from $x$ to $y$, relocate 1 or more bars from $k$ to $k-1$ starting from the top leftmost bar of $k$. To go from $y$ to $x$, relocate 1 or more bars from $k-1$ to $k$ starting from the bottom rightmost bar of $k-1$. 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 seeing 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.

The CyclicBarDriver algorithm creates a forest of ladders where each tree in the forest is one of the trees produced from the CyclicBar algorithm. The

forest structure is created as follows. Simply call the algorithm for the tree structure in a for loop ranging from $0 \ldots \binom{n}{2}$. 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 produces the root ladder from each $OptL\{\pi_n\}$, thus adding one more ladder to $L_n$. Once complete, the tree of ladders terminates, and the *currentLimit* increases, thus producing a new tree in the forest for $L_n$. To see the forest produced by the CyclicBar and CyclicBarDriver algorithms for $n = 4$ please refer to Figure 3.10

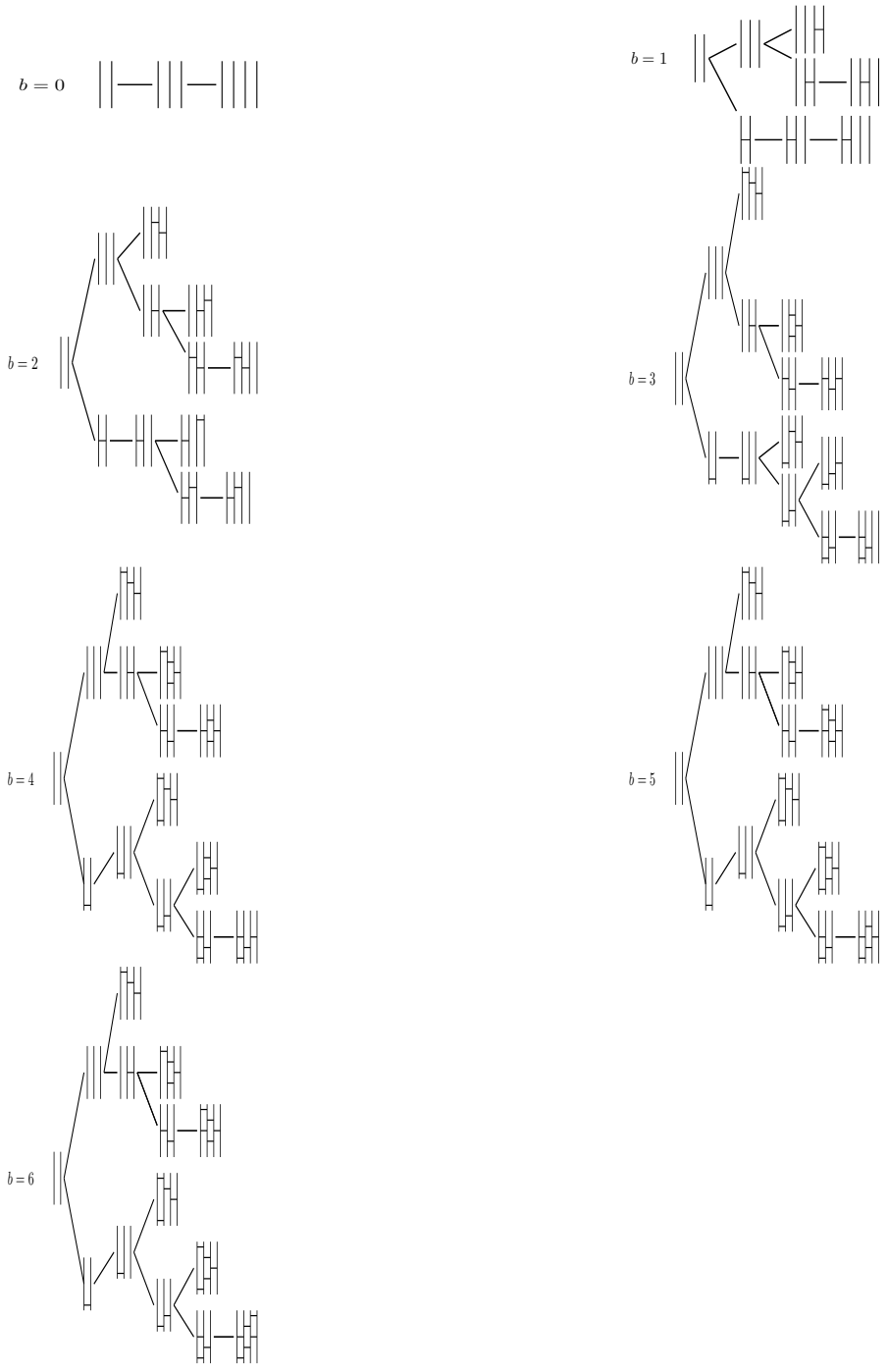



**Figure 3.10:** The forest for all ladders in $L_4$ generated by the Cyclic Bar Algorithm. The leaf nodes present a possibly correct candidate ladder. If the ladder in the leaf has enough bars, then it is a legitimate ladder in the tree.

It has been stated that the forest produced by the Cyclic Bar algorithm generates $L_n$. This claim will be proven below.

**Theorem 3.2.6** *The forest produced by* CyclicBar *generates* $L_n$

*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 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

| $n$ 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 3.1:** The table showing all $List(4,2)$ derived from $List(3,0) \ldots List(3,2) + List(4,k')$

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 $L_n$ 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 3.1.

□

## 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 $n = 12$ will be provided in Table3.2. In the analysis section, the table will be further analyzed along with the time and space complexity for each algorithm.

| Runtimes for generating $L_n$ 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 3.2:** The table with the runtimes for listing $L_n$ 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 MODIFIEDSJT algorithm performs better than the —CYCLICBAR algorithm. The reasons 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 MODIFIEDSJT algorithm performs better than the CYCLICBAR algorithm. The reasons for this improved performance are the difference in time complexities between the two algorithms. The time complexity for the MODIFIEDSJT algorithm is constant amortized time per ladder. The time will be proven in the following lemma.

**Lemma 3.4.1** *The amortized time for* ModifiedSJT *is* $O(1)$ *per ladder.*

*Proof.* Each iteration of the for loop inserts or removes a bar, thus creating a new ladder. This is done in constant time per addition or removal of a bar. □

**Lemma 3.4.2** *The amortized time for the* CyclicBar *algorithm is* $O(n^2)+n^2)$ *per ladder.*

*Proof.* The first $n^2$ term is a result of the for loop that is executed for $k = 2, 3, \ldots n$. 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 second $n^2$ 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 amortized time per ladder is $O(n^2)+n^2)$. □

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 Applications

The applications for generating $L_n$ 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 $L_n$ 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 [10]. 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 $L_n$ sorts the corresponding permutation of order $n$, one can implement all of $L_n$ for some arbitrary $n$ value and then pass each permutation of order $n$ through its respective ladder from $L_n$ resulting in each permutation being sorted. The ladders from $L_n$ only need to be generated once and saved to disk or implemented in hardware.

# 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 ladders of shortest height from $OptL\{\pi\}$. Let *a minimal ladder* be a ladder from $MinL\{\pi\}$. The *Minimum Height Problem* asks, given a permutation $\pi$, is there an algorithm for creating 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 $\pi$ 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 $\pi_n$ is the sorted permutation of order $n$ then there are no bars in its ladder. Recall that a bar swaps an adjacent inversion in $\pi$. 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. □

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 Figure 4.1 for an example of modifying $RootL(5,4,3,2,1)$ to $MinL(5,4,3,2,1)$. 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

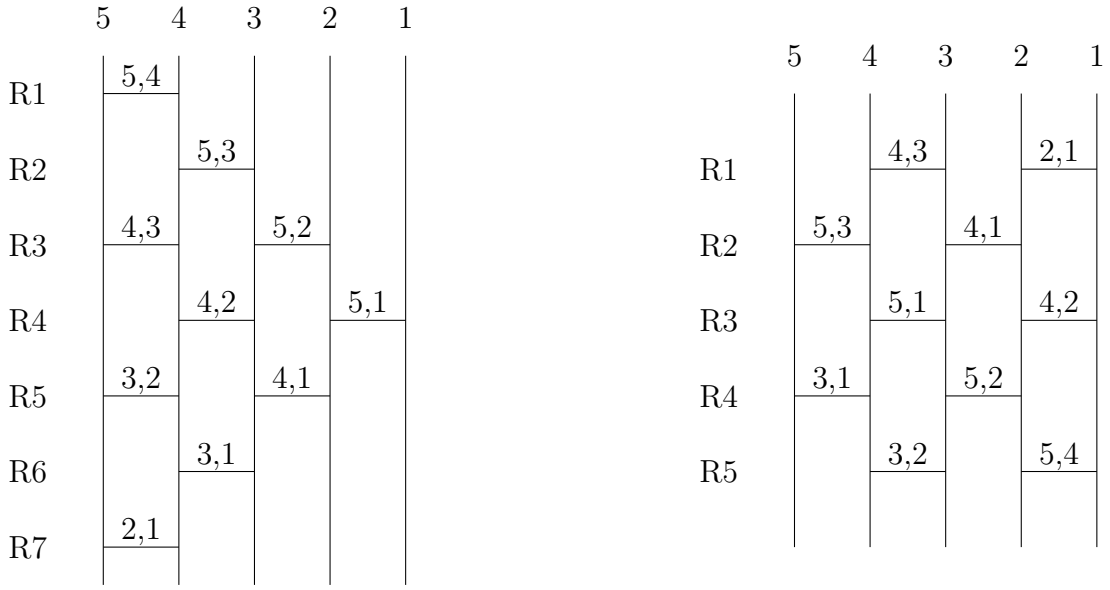



**Figure 4.1:** The ladder to the left is $Root(5, 4, 3, 2, 1)$. The ladder to the left is $MinL(5, 4, 3, 2, 1)$. Note that $n = 5 = 2K + 1$, thus by swapping routes 2 and 4 above route 5 whilst leaving route 3 below route 5 in $Root(5, 4, 3, 2, 1)$, we get $MinL(5, 4, 3, 2, 1)$. There is no way to reduce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

permutation is $n$). Let $MinL(Rev(n))$ be the minimal ladder for the reverse permutation of order $n$. Refer to Figure 4.1 for an example of $MinL(5, 4, 3, 2, 1)$. It will be shown that for each permutation of order $n$, one of its minimal ladders can be derived from $MinL(Rev(n))$. Recall that a bar uninverts 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 Figure 4.2 for the removal sequence which lists $MinL\{\pi_4\}$.

□

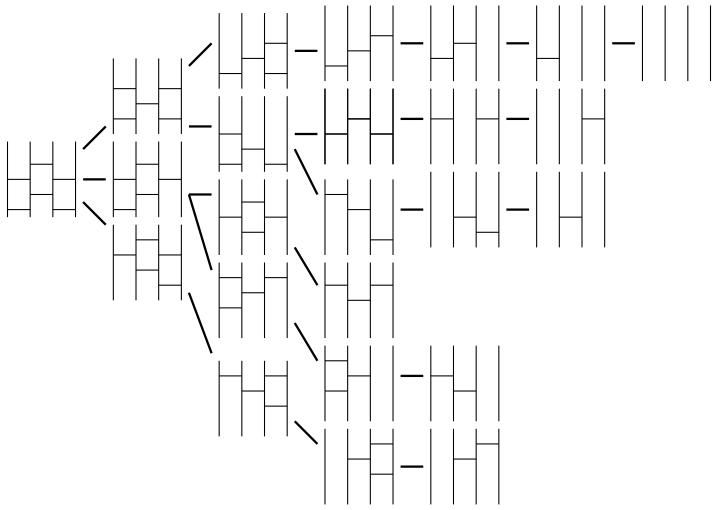


**Figure 4.2:** Removal sequence of bars from the minimal ladder for $(4, 3, 2, 1)$ resulting in $MinL\{\pi_n\}$

### 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

Let *ladder* be a two dimensional array with $n-1$ columns and 1 row. Lat *col* be initialized to $n-1$. Algorithm 9 generates all ladders of order $n$ with a height of one.

**Algorithm 9** 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
```

In GenHeightOne, 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 column to the left of the current column. 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 Figure 4.3.

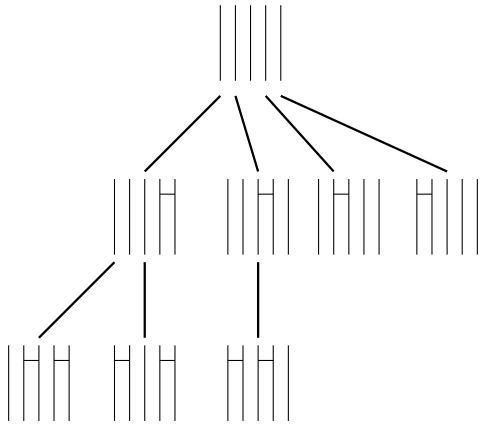


**Figure 4.3:** All 7 ladders of order 5 with a height of one listed by the function GenHeightOne

#### 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 [25][26]. 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 Figure 4.4 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.

□

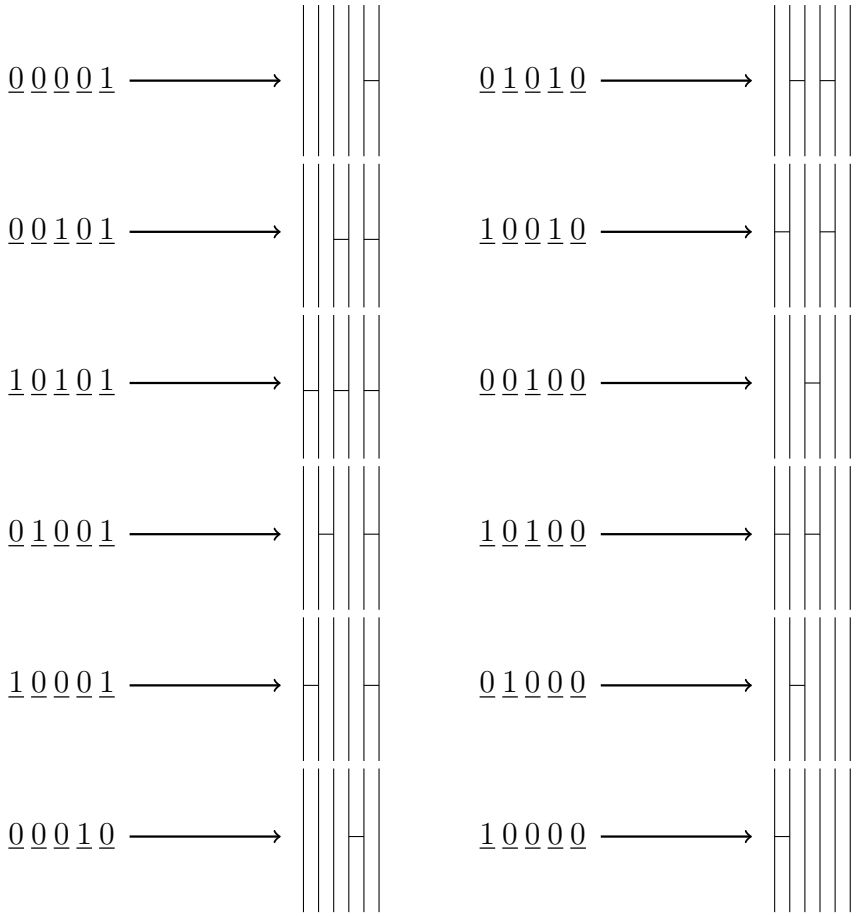



**Figure 4.4:** All 12 binary strings of length 5 with at least one 1 and no consecutive 1$s$ maps to all twelve ladders of order 6 with a height of one. The recurrence relation being $L(6) = 2L(4) + (L(5) - L(4)) + 1 = L(4) + L(5) + 1$

#### 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 mathematical phenomena because ladders with a height follow the same pattern as these other mathematical phenomena [24]. One of these phenomena include the number of in-

volutions in the Symmetric Group $S_n$ [4]. 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 $x$ [17].

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 $(4, 5, 6)$ [5]. 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 [4]. For example, $X = \{1, 2, 3\}$. Let $S_X = S_n = \{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}$. The involutions of $S_n = \{(2, 1, 3), (1, 3, 2), (1, 2, 3)\}$. The reason these are the involutions is because

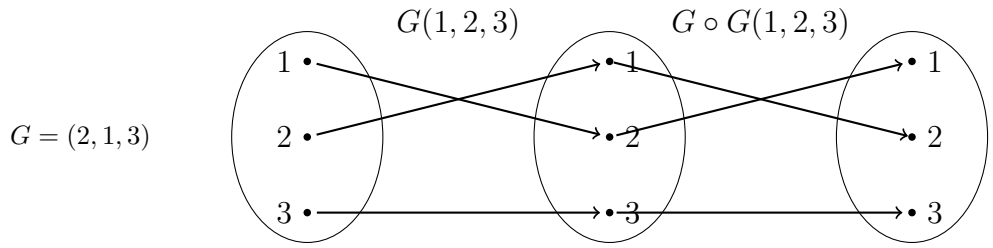



**Figure 4.5:** The involution $(2,1,3)$ composed with itself when applied to the identity permutation returns the identity permutation

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 Figure 4.5

**Theorem 4.1.4** *There is a bijective 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 any permutation is composed with itself, the result of the composition is the identity permutation. If a permutation is an involution then 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 permutation 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 $\pi$ exactly once and no two bars have the same element crossing them. It follows that each bar places an element in $\pi$ 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 $G \circ G(\pi_{ID}) = \pi_{ID}$. I propose that for each $G$ there is a corresponding $F(L_n)$ such that $F(L_n) \circ G(\pi_{ID}) = \pi_{ID}$ 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 $F(L_n) \circ G(\pi_{ID}) \neq \pi_{ID}$ 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 $\pi$ We know that each $G$ is unique seeing as the involution set $\subset$ all $n!$ permutations and all $n!$ permutations are unique. Thus, if $F(L_n) \circ K(\pi_{ID}) \neq \pi_{ID}$ 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 $pi_{ID}$ 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 $\pi_{ID}$ 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 $A \circ G(\pi_{ID}) = B \circ G(\pi_{ID}) = \pi_{ID}$ 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 $A \circ G(\pi_{ID}) = B \circ G(\pi_{ID}) = \pi_{ID}$ it must be the case that $A = B$ which is a contradiction. Therefore, for each $G$ there exists an $F(L_n)$ such that $F(L_n) \circ G(\pi_{ID}) = \pi_{ID}$ 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 Figure 4.6. □

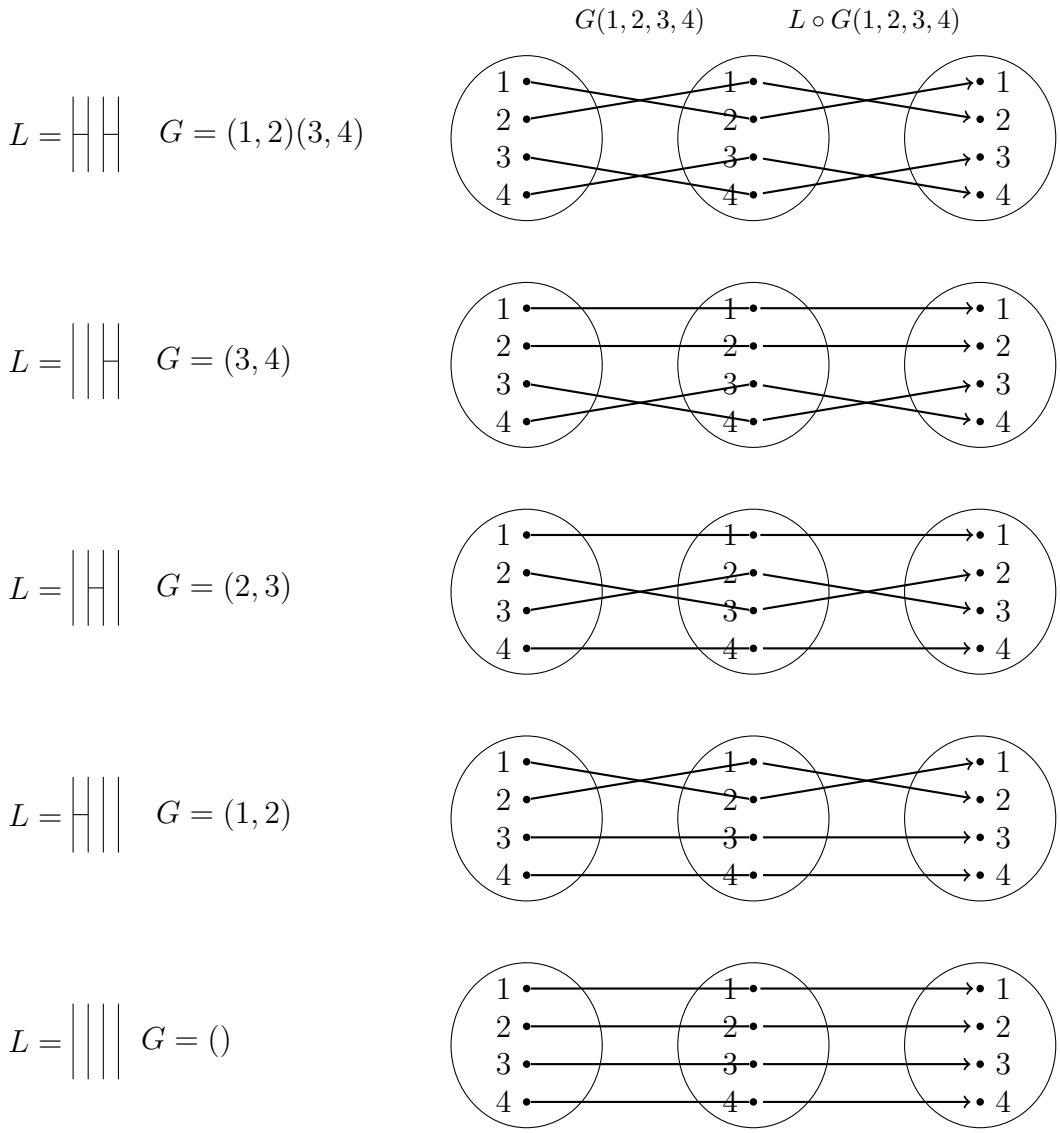



**Figure 4.6:** All ladders of order 4 with a height of zero or one form a bijection with the involution set of $S_4$.

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:[24]

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

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 [22]. 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 Figure 4.7 to see the sequences together.

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

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

From looking at the sequences in Figure 4.7, 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:

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

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

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

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 $\pi$ 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 $\pi$ would first require creating $MinL(Rev(\pi_n))$, then each bar in $MinL(Rev(\pi_n))$ that does not correspond to an inversion in $\pi$ 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 $\pi$ of order $n$ please refer to Figure 4.8. To see the algorithm for creating $MinL(Rev(\pi_n))$ please refer to Algorithm 10.

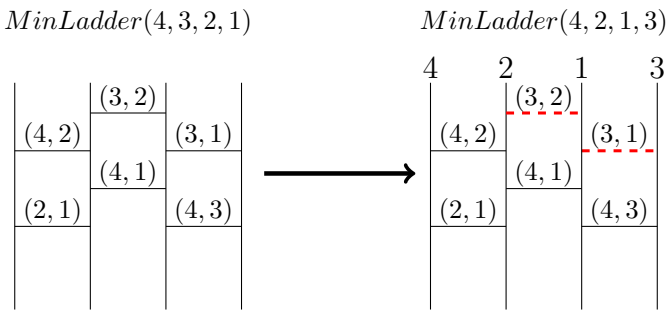



**Figure 4.8:** Given $\pi = (4, 2, 1, 3)$, the exact procedure first creates a min ladder for $(4, 3, 2, 1)$ then removes bars to create a min ladder for $(4, 2, 1, 3)$

### 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 create $MinL(Rev(\pi_n))$. 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 $0 \leq k \leq \lfloor (n/2) \rfloor$ 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

**Algorithm 10** 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 from 1 to elem − 1 do
17:        ladder[r][c] ← 1
18:        r ← r + 1, c ← c + 1
19:    end for
20: end function
```

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 $0 \leq k \leq \lfloor (n/2) \rfloor$ 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 greater than 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 $m+2$. To see $MinL((5,4,3,2,1))$ please refer to Figure 4.1. □

**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 $\binom{n}{2}$. □

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

In the previous section, the algorithm to create $MinL(Rev(\pi))$ was provided. CMinRL is exact, but unfortunately only creates 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. One can say if a ladder has a height of one, then it sorts $\pi$

into $\pi_{ID}$ in one step. If a ladder has a height of two, then it sorts $\pi$ into an intermediary $\pi_2$ in row 1 then sorts $\pi_2$ into $\pi_{ID}$ in row 2, etc. Define *bar compression* as the average number of bars per row in a 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 a 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 a ladder.

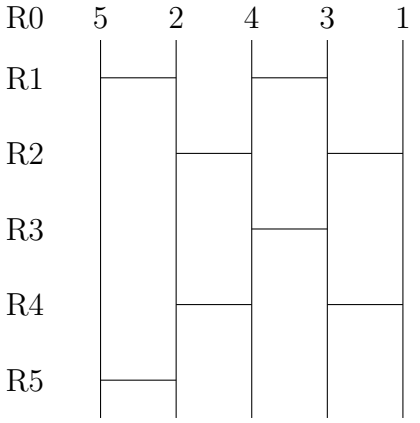
Recall that when inverted elements of $\pi$ travel through the ladder and cross a bar, the elements are swapped, thus resulting in some intermediate permutation $\pi_k$. Define $InvPi(\pi)$ as the permutation of intermediate permutations, beginning at $\pi$ and ending at the identity permutation. Each $\pi_k \in InvPi(\pi)$ corresponds to a row from a unique $ladder \in OptL\{\pi\}$. Given some arbitrary $\pi$, there can be more than one $InvPi(\pi)$. See table 4.1. for two different $InvPi(3,5,4,6,2,1)$.

| 2 $InvPi(3,5,4,6,2,1)$ | | |
|---|---|---|
| $\pi_k$ | $A = InvPi(3,5,4,6,2,1)$ | $B = InvPi(3,5,4,6,2,1)$ |
| $\pi_1$ | $(3,5,4,6,2,1)$ | $(3,5,4,6,2,1)$ |
| $\pi_2$ | $(3,4,5,6,1,2)$ | $(3,4,5,6,2,1)$ |
| $\pi_3$ | $(3,4,5,1,6,2)$ | $(3,4,5,2,6,1)$ |
| $\pi_4$ | $(3,4,1,5,2,6)$ | $(3,4,2,5,6,1)$ |
| $\pi_5$ | $(3,1,4,2,5,6)$ | $(3,2,4,5,1,6)$ |
| $\pi_6$ | $(1,3,2,4,5,6)$ | $(2,3,4,1,5,6)$ |
| $\pi_7$ | $(1,2,3,4,5,6)$ | $(2,3,1,4,5,6)$ |
| $\pi_8$ | *none* | $(2,1,3,4,5,6)$ |
| $\pi_9$ | *none* | $(1,2,3,4,5,6)$ |

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

When creating a $MinL(\pi)$, the goal is to create 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. Let $\pi^k \in MinInvPi(\pi)$. Then the recurrence relation for $\pi^k = \pi^{k-1} \rightarrow \tau(p_i^{k-1}, p_{i+1}^{k-1}) | p_i^{k-1} > p_{i+1}^{k-1}$ and for any $\tau(\pi_i^{k-1}, \pi_{i+1}^{k-1}) \cap \tau(\pi_j^{k-1}, \pi_{j+1}^{k-1}) = \emptyset$. In simpler terms, for some $\pi^k \in MinInvPi(\pi)$, perform the maximum number of adjacent transpositions on adjacent inversions in $\pi^{k-1}$ as is possible. In turn, this means the maximum number of bars can be added to the $kth$ row in the ladder. The less rows there are in the ladder the smaller the corresponding $InvPi(\pi)$ and the greater the bar compression. To see an example of maximal bar compression and a corresponding $MinInvPi(\pi)$, please refer to Figure 4.9.




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

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

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

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

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

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

**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.

The following algorithm is a heuristic algorithm for creating a minimal ladder for any arbitrary permutation of order $n$. Let *ladder* be initialized to an $n$ by $n-1$ empty two dimensional array. Let $row$ be initialized to 1. Let $\pi$ be an arbitrary permutation of order $n$. The algorithm consists of two functions. One is HeuristicMinLadder and the other is PreProcessRowOne. Define a *decreasing substring of* $\pi$ *(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 $\pi$ 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 $\pi$, the DSS is $(3, 2, 1)$ which has odd polarity. Let $S$ be the set of all $DSS_{2k} \in \pi$. Let $T$ be the set of all $DSS_{2k+1} \in \pi$. $S$ and $T$ are only used in PreProcessRowOne. *Maximal zig-zagginess* is defined as follows:

$$\text{Max zig-zag} = \begin{cases} p_1 < p_2 > p_3 < p_4 \ldots p_{n-1=2k} > p_{n=2k+1} & \text{if } n = 2k+1 \text{ and } p_1 < p_2 \\ p_1 > p_2 < p_3 > p_4 \ldots p_{n-1=2k} < p_{n=2k+1} & \text{if } n = 2k+1 \text{ and } p_1 > p_2 \\ p_1 < p_2 > p_3 < p_4 \ldots p_{n-1=2K-1} < p_{n=2k} & \text{if } n = 2k \text{ and } p_1 < p_2 \\ p_1 < p_2 > p_3 < p_4 \ldots p_{n-1=2K-1} < p_{n=2k} & \text{if } n = 2k \text{ and } p_1 > p_2 \end{cases}$$

**Algorithm 11** Heuristic algorithm to create a ladder with minimal height

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

**Algorithm 12** Algorithm to return the second permutation from $InvPi(\pi)$ which will result in the maximal bar compression

**function** PREPROCESSROWONE($\pi$, $n$, $S$, $T$)
    $\pi_a \leftarrow \pi$
    $\pi_b \leftarrow \pi$
    **for each** $DSS_{2k} \in S$ **do**
        $\tau(pa_i, pa_{i+1}), \tau(pa_{i+2}, pa_{i+3}) \ldots \tau(pa_{i+2k-2}, pa_{i+2k-1}$
        $\tau(pb_i, pb_{i+1}), \tau(pb_{i+2}, pb_{i+3}) \ldots \tau(pb_{i+2k-2}, pb_{i+2k-1}$
        Where $\tau(pa_i, pa_{i+1})/\tau(pb_i, pb_{i+1})$ uninverts an adjacent inversion in $\pi_a/\pi_b$ corresponding to the inversion in the $DSS_{2k} \in S$
    **end for**
    **for each** $DSS_{2k+1} \in T$ **do**
        From left to right $\tau(pa_i, pa_{i+1}), \tau(pa_{i+2}, pa_{i+3}) \ldots \tau(pa_{i+2k-2}, pa_{i+2k-1})$
        Where $\tau(pa_i, pa_{i+1})$ uninverts an adjacent inversion in $\pi_a$ corresponding to the inversion in the $DSS_{2k+1} \in T$
    **end for**
    **for each** $DSS_{2k+1} \in T$ **do**
        From right to left $\tau(pb_i, pb_{i-1}), \tau(pb_{i-2}, pb_{i-3}) \ldots \tau(pb_{i-2k-3}, pb_{i-2k-2})$
        Where $\tau(pb_i, pb_{i-1})$ uninverts an adjacent inversion in $\pi_b$ corresponding to the inversion in the $DSS_{2k+1} \in T$
    **end for**
    **if** $\pi_a$ and $\pi_b$ are equally zig-zaggy **then**
        return $\pi_b$
    **else if** $\pi_a$ is more zig-zaggy than $\pi_b$ **then**
        **return** $\pi_a$
    **else**
        **return** $\pi_b$
    **end if**
**end function**

The more zig-zaggy the resulting permutation from PreProcessRowOne is, the more bars are added to row 1 of the ladder. PreProcessRowOne returns the $\pi_2$ from some $MinInvPi(\pi)$; the first permutation being $\pi_1 = \pi$. There are two criteria for $\pi_2$. The first is $\pi_2$ is a transformation of $\pi$ such that $\pi$ has undergone as many adjacent transpositions as possible. The second criteria for $\pi_2$ is that $\pi_2$ is as zig-zaggy as possible, given that it has undergone the maximum amount of adjacent transpositions. 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 $\lfloor (k/2) \rfloor$ 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 $k/2$ $\tau(p_i, p_{i+1})$ in tandem for all even length $DSS \in S$. Then, for all odd length $DSS \in T$, PreProcessRowOne performs $\lfloor k/2 \rfloor$ $\tau(pa_i, pa_{i+1}), \ldots \tau(pa_{k-2}, pa_{k-1})$ resulting in a candidate permutation $\pi_a$. PreProcessRowOne then performs $\lfloor k/2 \rfloor$ $\tau(pb_k, pb_{k-1}) \ldots \tau(pb_3, pb_2)$ resulting in a second candidate permutation $\pi_b$. This results in two candidate permutations for $\pi_2$ in $MinInvPi(\pi)$. In order to choose between $\pi_a$ and $\pi_b$, 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 a permutation is, the more adjacent 2 length DSS there are in said permutation. The more adjacent 2 lengthed DSS there are, the more pairwise disjoint adjacent inversions there are in said permutation. The more pairwise disjoint adjacent inversions there are in a permutation, the more bars can be added to ladder

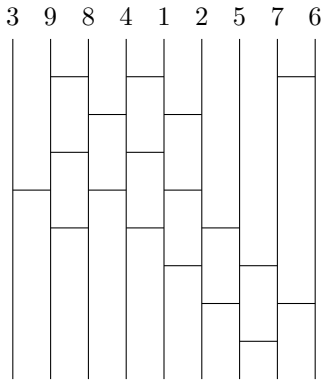



**Figure 4.10:** The resulting ladder from HEURISTICMINLADDER

at the row corresponding to the permutation. The result is likely to be $MinL(\pi)$ and the shortest length $InvPi(\pi)$. Once $\pi_2$ has been selected from PREPROCESSROWONE, HEURISTICMINLADDER adds the corresponding bars to the ladder. Then, HEURISTICMINLADDER continues to perform the maximal number $\tau(p_i, p_{i+1})$ in each subsequent $\pi_k$ until $\pi_k = \pi_{ID}$. HEURISTICMINLADDER adds bars to the $kth$ row accordingly. If any intermediate $\pi_k$ has a $DSS_{2k+1}$ then HEURISTICMINLADDER swaps from left to right. To see $MinL((3, 9, 8, 4, 1, 2, 5, 7, 6))$ please refer to Figure 4.10. To see an example of the resulting $pi_2$ from PREPROCESSROWONE given $\pi = (4, 3, 9, 8, 7, 6, 12, 11, 5, 2, 1, 14, 13, 10)$ please refer to Example 4.2.3.

**Example 4.2.3** *Example demonstrating* PREPROCESSROWONE

1. *Let* $\pi = (4, 3, 9, 8, 7, 6, 12, 11, 5, 2, 1, 14, 13, 10)$
2. *Let* $S = \{(4, 3), (9, 8, 7, 6)\}$
3. *Let* $T = \{(12, 11, 5, 2, 1), (14, 13, 10)\}$
4. *Step 1:* PREPROCESSROWONE *assigns* $\pi$ *to two candidate permutations* $\pi_a$ *and* $\pi_b$.
5. *Step 2: Perform adjacent transpositions for each* $DSS \in S$ *in* $\pi_a, \pi_b$.
6. *After applying step 2:* $\pi_a = (3, 4, 8, 9, 6, 7, 12, 11, 5, 2, 1, 14, 13, 10)$

7. *After applying step 2:* $\pi_b = (3, 4, 8, 9, 6, 7, 12, 11, 5, 2, 1, 14, 13, 10)$

8. *Step 3a): Perform adjacent transpositions for each* $DSS \in T$ *in* $\pi_a$ *going left to right beginning at the leftmost element of each* $DSS \in T$.

9. *After applying step 3a:* $\pi_a = (3, 4, 8, 9, 6, 7, 11, 12, 2, 5, 1, 13, 14, 10)$

10. *Step 3b): Perform adjacent transpositions for each* $DSS \in T$ *in* $\pi_b$ *going right to left beginning at the rightmost element of each* $DSS \in T$.

11. *After applying step 3b:* $\pi_b = (3, 4, 8, 9, 6, 7, 12, 5, 11, 1, 2, 14, 10, 13)$.

12. *Calculate the zig-zagginess for* $\pi_a = 13 - (2 + 2 + 1) = 8$.

13. *Calculate the zig-zaginess for* $\pi_b = 13 - (2 + 1 + 1) = 9$.

14. $\pi_b$ *is more zig-zaggy than* $\pi_a$. *Therefore return* $\pi_b$.

*End of example.*

The zig-zagginess of a permutation is calculated as follows. Given $n$ elements in $\pi$ there are $n - 1$ adjacent relations between the $n$ elements in $\pi$ such that either $\pi_i > \pi_{i+1}$ or $\pi_i < \pi_{i+1}$. Refer to the gap between two elements in $\pi$ as *spots in the permutation*. Thus, there are $n - 1$ spots in the permutation such that each spot can take on the $>$ or $<$ relation. If a permutation is maximally zig-zaggy, then for any two spots, $s_{j-1}$, $s_j$ if $s_{j-1}$ is $<$ then $s_j$ is $>$ and if $s_{j-1}$ is $>$ then $s_j$ is $<$. Therefore, a permutation with maximal zig-zagginess has a zig-zagginess value of $n-1$. Whenever there is an $s_{j-1}$ that has the same relation as $s_j$ subtract 1 from $n - 1$. I.E. if $s_{j-1}$ is $<$ and $s_j$ is $<$ then subtract 1 from $n - 1$. The resulting number is the value of the zig-zagginess of $\pi$. In the above example, $(n - 1) = 13$, $\pi_a$ has a zig-zagginess of $13 - 5 = 8$ and $\pi_b$ has a zig-zagginess of $13 - 4 = 9$. Therefore, $\pi_b$ is used as $\pi_2$ in $MinInvPi((4, 3, 9, 8, 7, 6, 12, 11, 5, 2, 1, 14, 13, 10))$.

**Lemma 4.2.4** *Given* $j$ *adjacent inversions in* $\pi_k$, *the more of these inversions that are pairwise disjoint, the more bars can be added to the* $k$*th row.*

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

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

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 $n = 1 = \lceil (j/2) \rceil$.
Base case 3: let $\pi = (3,2,1)/j = 2$. $(3,2) \cap (2,1) = \{2\}$ and $m = j+1 = 3$ and $n = 1 = \lfloor (j/2) \rfloor$.

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 $n = \lceil (j+1/2) \rceil / \lfloor (j+1/2) \rfloor$ 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 $\pi$. The reason being is that if the $j+1th$ inversion was formed by one more element in $\pi$ 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 $\pi$ then inversion $j+1$ cannot be pairwise disjoint. Let $inv(a,b)$ be the $jth$ inversion in $\pi$ 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, $n = j + 1$.

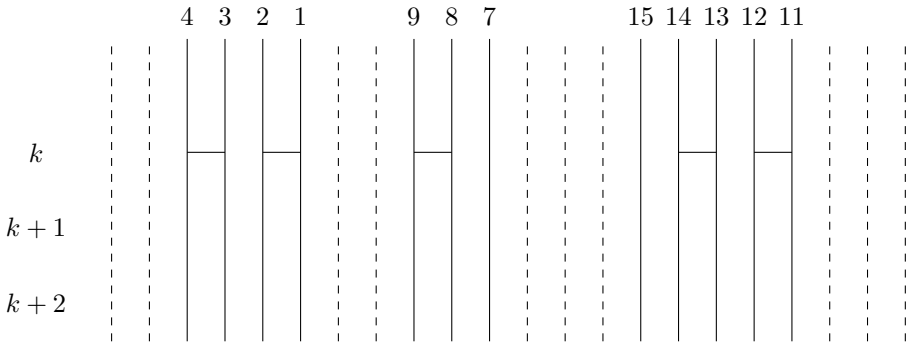
Suppose the $j + 1th$ adjacent inversion is not pairwise disjoint and suppose the first $j$ inversions are also not pairwise disjoint, then the $j + 1th$ adjacent inversion would require one more element to form an inversion in $\pi$. We shall do a direct proof to show that one more element is required. Let $inv(a, b)$ be the $jth$ inversion in $\pi$ where $a > b$. Let $x$ be the element to form the $j + 1th$ inversion in $\pi$. 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 $\pi$ 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 $n = \lfloor (j + 1/2) \rfloor$. 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 $n = \lceil (j + 1/2) \rceil$. 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 $\pi$ 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 $n = \lfloor (j + 1/2) \rfloor$. 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 $n = \lceil (j + 1/2) \rceil$.

Clearly $n = j > n = \lfloor (j/2) \rfloor / \lceil (j/2) \rceil$, 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 Figure 4.11.

□




| 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 4.11:** Figure demonstrating that the more pairwise disjoint adjacent inversions there are in $\pi_k$ the more bars can be added to *ladder* at $row = k$

From looking at Figure 4.11, 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 create $MinL(\pi)$, PrePROCESSROWONE performs the maximum number of transpositions of adjacent inversions in $\pi$. This leads to multiple candidate permutations. Once done, to determine which candidate permutation is the best option for $\pi_2$, determine which candidate is most zig-zaggy. This permutation is the permutation for $\pi_2$ in $MinInvPi(\pi)$. Then, from $\pi_2$, HEURISTICMINLADDER uninverts as many adjacent inversions in every subsequent $\pi_k$ 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. Please refer to Table 4.2 to see the cases where the heuristic algorithm created an incorrect ladder. In the analysis section, Table 4.2 will be analyzed and concluding remarks on the minimum height problem will be made.

| 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 $\pi_2$ 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 lead to a ladder with minimal height. Rather, the $\pi_2$ 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 Figure 4.12.

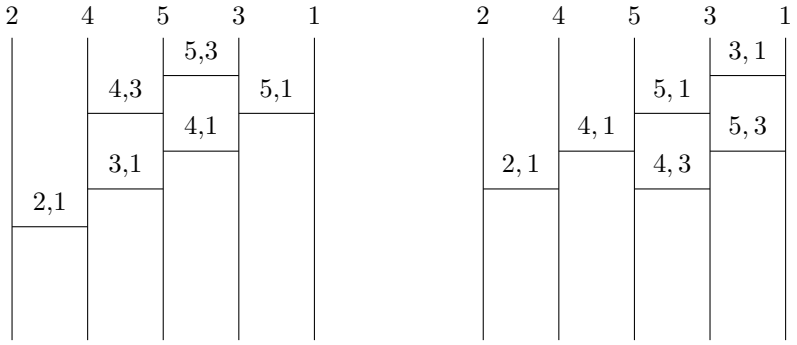



**Figure 4.12:** HEURISTICALGORITHM ladder on the left, shortest ladder is on the right.

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 $\pi_k$. This does not necessarily lead to a globally optimal solution 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 Figure A.1 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 $\pi$?

### 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.

# Bibliography


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

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

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

[4] N. Biggs and A. T. White. *Permutation Groups and Combinatorial Structures.* Number Vol. 33 in London Mathematical Society Lecture Note Series; 33. Cambridge University Press, 1979.

[5] P. J. Cameron. *Permutation Groups.* Number Vol. 45 in London Mathematical Society Student Texts. Cambridge University Press, 1999.

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

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

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

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

[10] M. T. Goodrich. *Zig-Zag Sort: A Simple Deterministic Data-Oblivious Sorting Algorithm Running in O(n Log n) Time.* STOC '14. Association for Computing Machinery, New York, NY, USA, 2014.

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

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

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

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

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

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

[17] W. Ledermann. *Introduction to the Theory of Finite Groups*. University Mathematical Texts. Oliver and Boyd, Edinburgh, Oliver and Boyd, 1961, (4th rev. ed.) edition, 1957.

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

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

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

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

[22] P. Singh. The So-called Fibonacci Numbers in Ancient and Medieval India. *Historia Mathematica*, 12(3):229 – 244, 1985.

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

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

[25] N. J. A. Sloane. *A Handbook of Integer Sequences*. Academic Press, 1993.

[26] N. J. A. Sloane. *Integer Sequences, Online Encyclopedia Of*, page 891892. John Wiley and Sons Ltd., GBR, 2003.

[27] F. Vega. One-In-Three 3sat is in P. *hal-01069057*, 2014.

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

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

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

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

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

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

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

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

# Appendix A

## Appendix

MinLadders.txt

```
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
```

---

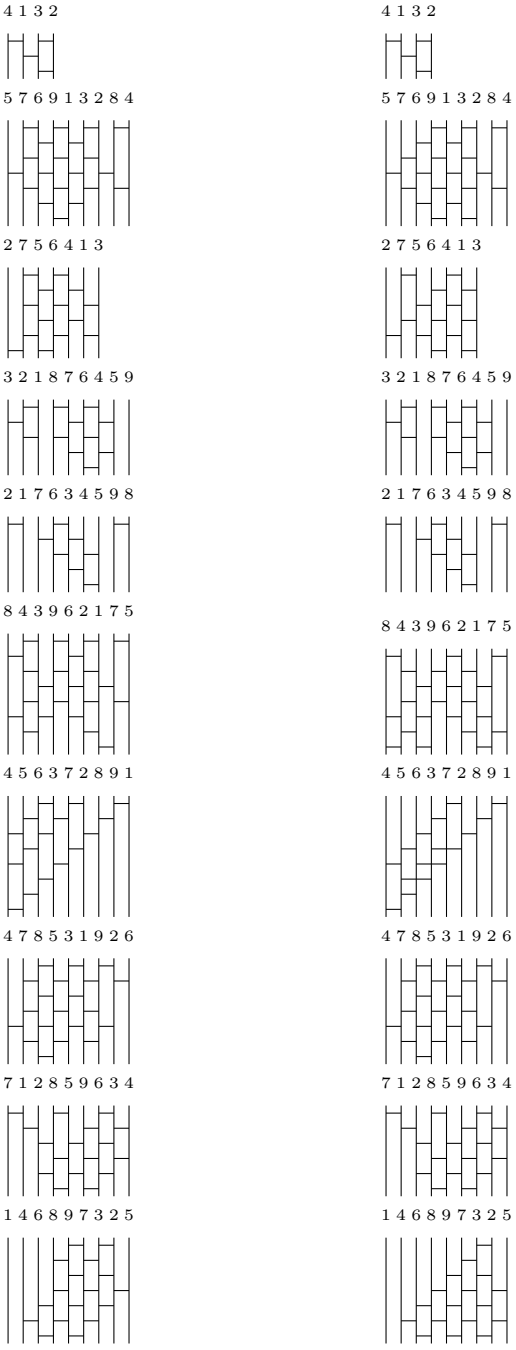



**Figure A.1:** Heuristic algorithm on the left, Brute Force on the right

| Permutation | number | k | $\lvert OptL\{\pi\}\rvert$ |
|---|---|---|---|
| 1 2 3 4 5 | 1 | 0 | 1 |
| 1 2 3 5 4 | 2 | 1 | 1 |
| 1 2 4 3 5 | 3 | 1 | 1 |
| 1 3 2 4 5 | 4 | 1 | 1 |
| 2 1 3 4 5 | 5 | 1 | 1 |
| 1 2 4 5 3 | 6 | 2 | 1 |
| 1 2 5 3 4 | 7 | 2 | 1 |
| 1 3 2 5 4 | 8 | 2 | 1 |
| 2 1 3 5 4 | 9 | 2 | 1 |
| 1 3 4 2 5 | 10 | 2 | 1 |
| 1 4 2 3 5 | 11 | 2 | 1 |
| 2 1 4 3 5 | 12 | 2 | 1 |
| 2 3 1 4 5 | 13 | 2 | 1 |
| 3 1 2 4 5 | 14 | 2 | 1 |
| 1 3 4 5 2 | 15 | 3 | 1 |
| 1 4 2 5 3 | 17 | 3 | 1 |
| 2 1 4 5 3 | 18 | 3 | 1 |
| 1 3 5 2 4 | 19 | 3 | 1 |
| 1 5 2 3 4 | 20 | 3 | 1 |
| 2 1 5 3 4 | 21 | 3 | 1 |
| 2 3 1 5 4 | 22 | 3 | 1 |
| 3 1 2 5 4 | 23 | 3 | 1 |
| 2 3 4 1 5 | 24 | 3 | 1 |
| 3 1 4 2 5 | 26 | 3 | 1 |
| 2 4 1 3 5 | 27 | 3 | 1 |
| 4 1 2 3 5 | 28 | 3 | 1 |
| 1 4 3 2 5 | 25 | 3 | 2 |
| 1 2 5 4 3 | 16 | 3 | 2 |
| 3 2 1 4 5 | 29 | 3 | 2 |
| 2 3 4 5 1 | 30 | 4 | 1 |
| 3 1 4 5 2 | 33 | 4 | 1 |
| 1 4 5 2 3 | 34 | 4 | 1 |
| 2 4 1 5 3 | 37 | 4 | 1 |
| 4 1 2 5 3 | 38 | 4 | 1 |
| 2 3 5 1 4 | 39 | 4 | 1 |
| 3 1 5 2 4 | 41 | 4 | 1 |
| 2 5 1 3 4 | 42 | 4 | 1 |
| 5 1 2 3 4 | 43 | 4 | 1 |
| 1 3 5 4 2 | 31 | 4 | 2 |
| 1 4 3 5 2 | 32 | 4 | 2 |
| 1 5 2 4 3 | 35 | 4 | 2 |
| 2 1 5 4 3 | 36 | 4 | 2 |
| 1 5 3 2 4 | 40 | 4 | 2 |
| 3 2 1 5 4 | 44 | 4 | 2 |
| 2 4 3 1 5 | 45 | 4 | 2 |
| 3 2 4 1 5 | 46 | 4 | 2 |
| 4 1 3 2 5 | 48 | 4 | 2 |
| 4 2 1 3 5 | 49 | 4 | 2 |
| 2 4 5 1 3 | 58 | 5 | 1 |
| 4 1 5 2 3 | 60 | 5 | 1 |
| 3 5 1 2 4 | 66 | 5 | 1 |
| 2 3 5 4 1 | 50 | 5 | 2 |
| 2 4 3 5 1 | 51 | 5 | 2 |
| 3 2 4 5 1 | 52 | 5 | 2 |
| 3 1 5 4 2 | 55 | 5 | 2 |
| 4 1 3 5 2 | 57 | 5 | 2 |
| 2 5 1 4 3 | 61 | 5 | 2 |
| 5 1 2 4 3 | 62 | 5 | 2 |
| 4 2 1 5 3 | 63 | 5 | 2 |

| | | | |
|---|---|---|---|
| 2 5 3 1 4 | 64 | 5 | 2 |
| 3 2 5 1 4 | 65 | 5 | 2 |
| 5 1 3 2 4 | 67 | 5 | 2 |
| 5 2 1 3 4 | 68 | 5 | 2 |
| 1 5 4 2 3 | 59 | 5 | 3 |
| 3 4 2 1 5 | 69 | 5 | 3 |
| 4 2 3 1 5 | 70 | 5 | 3 |
| 4 3 1 2 5 | 71 | 5 | 3 |
| 1 4 5 3 2 | 53 | 5 | 3 |
| 1 5 3 4 2 | 54 | 5 | 3 |
| 3 4 5 1 2 | 77 | 6 | 1 |
| 4 5 1 2 3 | 85 | 6 | 1 |
| 4 2 5 1 3 | 84 | 6 | 2 |
| 3 5 1 4 2 | 80 | 6 | 2 |
| 2 4 5 3 1 | 72 | 6 | 3 |
| 2 5 3 4 1 | 73 | 6 | 3 |
| 3 4 2 5 1 | 75 | 6 | 3 |
| 4 2 3 5 1 | 76 | 6 | 3 |
| 3 5 2 1 4 | 88 | 6 | 3 |
| 5 2 3 1 4 | 89 | 6 | 3 |
| 5 3 1 2 4 | 90 | 6 | 3 |
| 4 1 5 3 2 | 79 | 6 | 3 |
| 5 1 3 4 2 | 81 | 6 | 3 |
| 4 3 1 5 2 | 82 | 6 | 3 |
| 2 5 4 1 3 | 83 | 6 | 3 |
| 5 1 4 2 3 | 86 | 6 | 3 |
| 5 2 1 4 3 | 87 | 6 | 4 |
| 1 5 4 3 2 | 78 | 6 | 8 |
| 4 3 2 1 5 | 91 | 6 | 8 |
| 3 5 4 1 2 | 98 | 7 | 3 |
| 4 3 5 1 2 | 99 | 7 | 3 |
| 4 5 1 3 2 | 100 | 7 | 3 |
| 4 5 2 1 3 | 103 | 7 | 3 |
| 3 4 5 2 1 | 92 | 7 | 4 |
| 5 2 3 4 1 | 96 | 7 | 4 |
| 5 4 1 2 3 | 105 | 7 | 4 |
| 5 3 1 4 2 | 102 | 7 | 5 |
| 5 2 4 1 3 | 104 | 7 | 5 |
| 4 2 5 3 1 | 94 | 7 | 5 |
| 3 5 2 4 1 | 95 | 7 | 5 |
| 2 5 4 3 1 | 93 | 7 | 8 |
| 4 3 2 5 1 | 97 | 7 | 8 |
| 5 1 4 3 2 | 101 | 7 | 8 |
| 5 3 2 1 4 | 106 | 7 | 8 |
| 4 5 2 3 1 | 109 | 8 | 6 |
| 4 5 3 1 2 | 112 | 8 | 6 |
| 5 3 4 1 2 | 113 | 8 | 6 |
| 3 5 4 2 1 | 107 | 8 | 11 |
| 4 3 5 2 1 | 108 | 8 | 11 |
| 5 2 4 3 1 | 110 | 8 | 11 |
| 5 3 2 4 1 | 111 | 8 | 11 |
| 5 4 1 3 2 | 114 | 8 | 11 |
| 5 4 2 1 3 | 115 | 8 | 11 |
| 4 5 3 2 1 | 116 | 9 | 20 |
| 5 3 4 2 1 | 117 | 9 | 20 |
| 5 4 2 3 1 | 118 | 9 | 20 |
| 5 4 3 1 2 | 119 | 9 | 20 |
| 5 4 3 2 1 | 120 | 10 | 62 |

### A.0.1 Axillary Functions and Details for FindAllChildren

#### A.0.1.1 Local Swap Operation

The parent-child relation in the tree structure for FindAllChildren is based on a *local swap operation* which corresponds to a braid relation and is a local modification of *ladder* as shown in Figure A.2. 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 conditions are met [34].

- 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 [34].

- 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 A.2 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 A.2 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.

The following algorithms are used to perform a local swap operation: Algorithm 13, Algorithm 14 and Algorithm 15. Let *ladder* be initialized to an empty ladder. Let *bar* be the bar to be right or left swapped in the *ladder*. In Algorithm 15

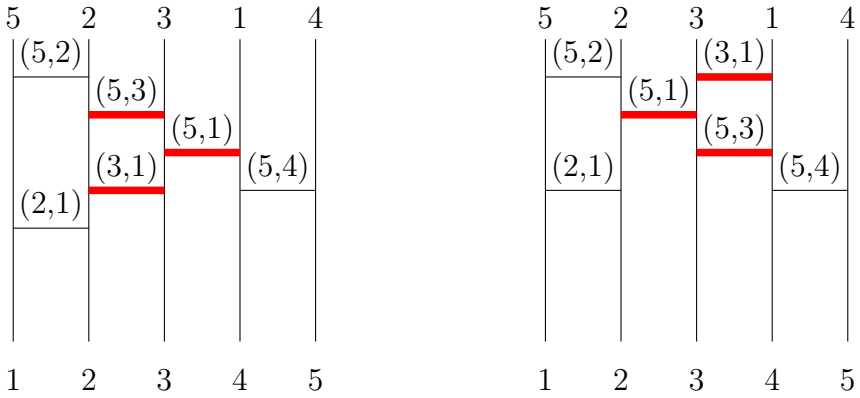



**Figure A.2:** Example of a local swap operation. When a right swap operation is performed on the left ladder, the result is the right ladder. When a left swap operation is performed on the right ladder, the result is the left ladder.

*offset* is initialized to 2 when right swapping and initialized to $-2$ when left swapping. In Algorithm 15 *index* is initialized 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 in the sub-ladder except the root of the sub-ladder. For an example of a sub-ladder please refer to Figure A.3.

**Algorithm 13** 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)
9:     end if
10:    Swap(upperNeighbor, ladder[row + 1][col + 1])
11:    Swap(bar, rightNeighbor)
12: end function
```

**Algorithm 14** 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:     leftNeighbor ← lfn(bar)
5:     Swap(bar, leftNeighbor)
6:     lowerNeighbor ← ln(bar)
7:     if col < n − 1 then
8:         subLadderRoot ← rc(lowerNeighbor)
9:         Swap(lowerNeighbor, ladder[row − 1][col − 1])
10:        ShiftSubLadder(ladder, subLadderRoot, −2)
11:    else
12:        Swap(lowerNeighbor, ladder[row − 1][col − 1])
13:    end if
14: end function
```

**Algorithm 15** Shifts the sub-ladder up or down the ladder data structure depending on if a right or left swap operation is being performed

```
1: function ShiftSubLadder(ladder, bar, offset)
2:     row ← bar's row in ladder
3:     col ← bar's column in ladder
4:     rightChild ← rc(bar)
5:     leftChild ← lc(bar)
6:     if rightChild = 0 and leftChild = 0 then
7:         Swap(ladder[row+offset][col], ladder[row][col])
8:         return
9:     else
10:        if offset== −2 indicating a left swap then
11:            Swap(ladder[row+offset][col], ladder[row][col])
12:            if rightChild ≠ 0 then
13:                ShiftSubLadder(ladder, rightChild, offset)
14:            end if
15:            if leftChild ≠ 0 then
16:                ShiftSubLadder(ladder, leftChild, offset)
17:            end if
18:        end if
19:        if offset== 2 indicating a right swap then
20:            if rightChild ≠ 0 then
21:                ShiftSubLadder(ladder, rightChild, offset)
22:            end if
23:            if leftChild ≠ 0 then
24:                ShiftSubLadder(ladder, leftChild, offset)
25:            end if
26:            Swap(ladder[row+offset][col], ladder[row][col])
27:        end if
28:    end if
29: end function
```

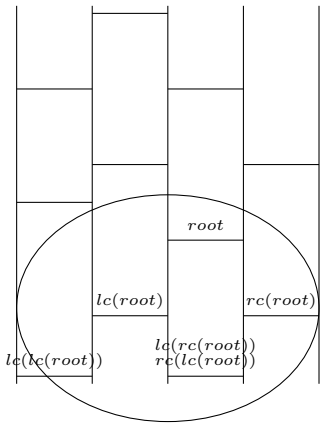



**Figure A.3:** The bars that are fully or partially in the circle make up the sub-ladder.

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 performing a right swap, ShiftSubLadder moves each bar in the sub-ladder two rows down the ladder. Since each bar in the sub-ladder is a left or right child of some other bar in the sub-ladder, with the exception of the root ladder, the index is set to 1 indicating the offset. 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; LeftSwap is the inverse of RightSwap. The *offset* value for ShiftSubLadder when performing a left swap is set to $-2$ to indicate the bars need to be moved up the ladder. When bars are moved in LeftSwap, the parent bar is shifted

upward before its children. This is unlike the RightSwap in which the children bars are shifted downward before their parent. This is done to ensure that for any two bars in the sub-ladder, $x, y$, $x$ will not be swapped with $y$, thus putting $y$ in the wrong position. This is why the $lowerNeighbor$ of $x$ in LeftSwap is swapped before ShiftSubLadder is called. Whereas in RightSwap the $upperNeighbor$ of $x$ is swapped after ShiftSubLadder is called. To see an example of all three algorithms performing a local swap operation please refer to Figure A.4.

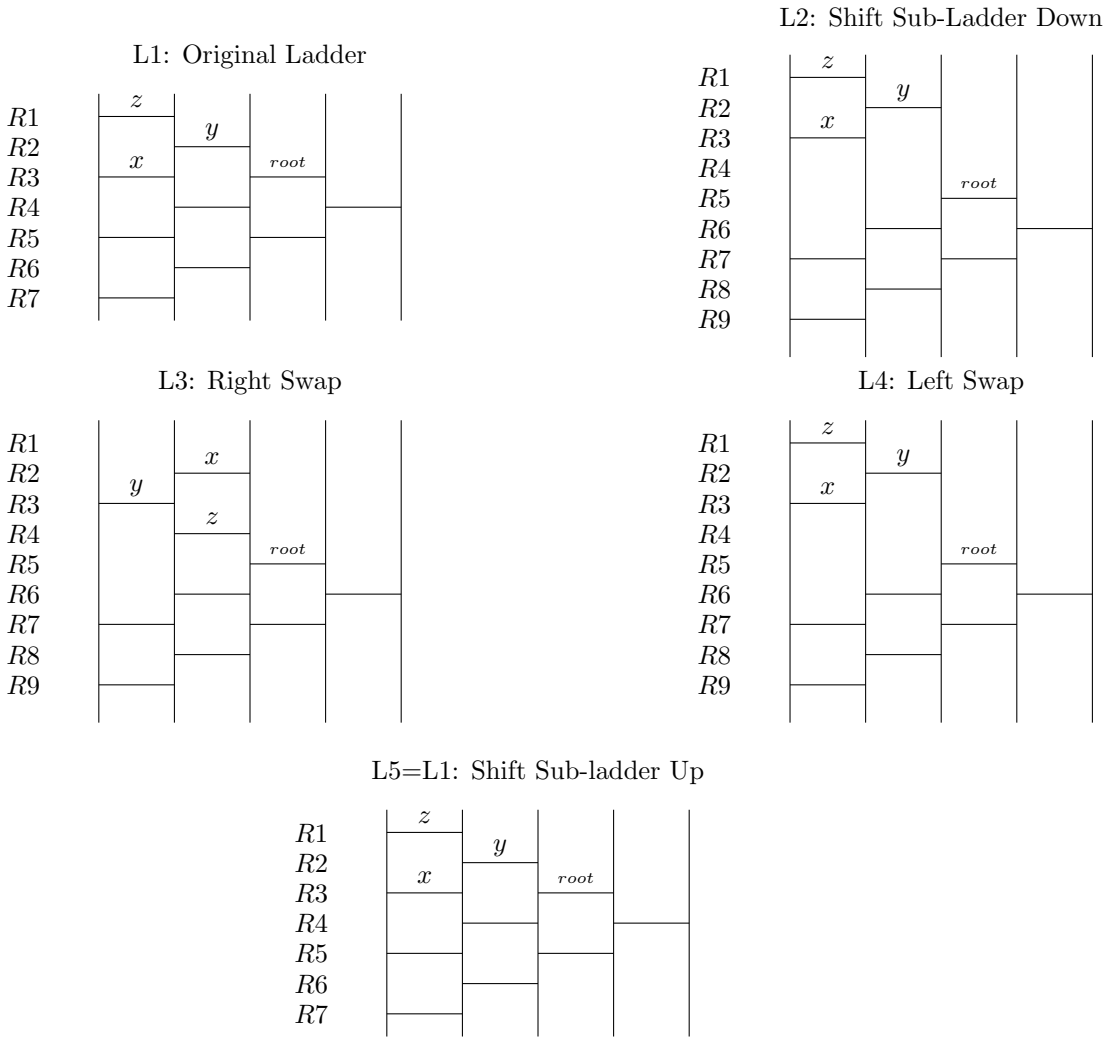



**Figure A.4:** $x, y, z$ to be locally swapped. *root* is the root of the sub-ladder.

**Lemma A.0.1 (h)** *The time complexity for performing a local swap operation is CAT, constant amortized time, per bar.*

*Proof.* In the LeftSwap and RightSwap algorithms, calculating the row and column for $x, y$ and $z$ along with the target row and column to swap each of these bars to is done in $O(1)$ time. In the algorithm ShiftSubLadder, each recursive call swaps one or two bars. The row and column for any given bar to be swapped is calculated in $O(1)$ time and the target row and column for any given bar is calculated in $O(1)$

time. □